Enabling System Wide Shared Memory for Performance Improvement in PyCOMPSs Applications

Clément Foyer*§
*HPE HPC/AI EMEA Research Lab
Bristol, United Kingdom
clement.foyer@hpe.com

Javier Conejero†, Jorge Ejarque†, Rosa M. Badia†
†Barcelona Supercomputing Center
Barcelona, Spain
{francisco.conejero,jorge.ejarque,rosa.m.badia}@bsc.es

Adrian Tate‡
‡Numerical Algorithms Group Ltd. (NAG)
Oxford, United Kingdom
adrian.tate@nag.co.uk

Simon McIntosh-Smith§
§High Performance Computing Research Group
Department of Computer Science, University of Bristol
Bristol, United Kingdom
{clement.foyer,s.mcintosh-smith}@bristol.ac.uk

***Abstract*—Python has been gaining some traction for years in the world of scientific applications. However, the high-level abstraction it provides may not allow the developer to use the machines to their peak performance. To address this, multiple strategies, sometimes complementary, have been developed to enrich the software ecosystem either by relying on additional libraries dedicated to efficient computation (e.g., NumPy) or by providing a framework to better use HPC scale infrastructures (e.g., PyCOMPSs).**

In this paper, we present a Python extension based on SharedArray that enables the support of system-provided shared memory and its integration into the PyCOMPSs programming model as an example of integration to a complex Python environment. We also evaluate the impact such a tool may have on performance in two types of distributed execution-flows, one for linear algebra with a blocked matrix multiplication application and the other in the context of data-clustering with a k-means application. We show that with very little modification of the original decorator (3 lines of code to be modified) of the task-based application the gain in performance can rise above $40\,\%$ for tasks relying heavily on data reuse on a distributed environment, especially when loading the data is prominent in the execution time.

***Index Terms*—Memory, Shared Memory, Task, Python, Parallel Programming, Distributed Memory, NumPy, Data Management**

I. Introduction

Through the convergence between High Performance Computing (HPC), Artificial Intelligence (AI) and Big Data, one area of focus is the availability of common tools that can bring the performance of the former to the techniques and algorithms used by the latter two. One big actor of this evolution is the development of the Python ecosystem, which provides a large set of tools and libraries while being designed for code readability, maintainability and enhancement over time. However, the ease of use comes at the cost of a complex memory management behind the scene, and complex data structures that get abstracted for the user. The computation intensity required by scientific applications cannot suffer such an overhead to be able to provide performance. Hence, we mitigate it by using dedicated data structures that provide contiguous buffers to make the best of modern processors and architectures.

In addition, frameworks like COMPSs [1] provide a seamless way to parallelize workloads across large scale infrastructures, notably for Python with the binding provided by PyCOMPSs [2]. However, because of the complexity of Python's internal structures, the communication between nodes and between processes is more complicated compared to using languages such as C or Fortran. Python class internal structure rely heavily on pointers to numerous structures, which could not be easily shared between processes. In order to benefit from Python memory allocation mitigation strategies, the fine grain management of memory can depend only on the Python interpreter, but care has to be shown when dealing with internal reference counters due to the risk of early deallocation done by the garbage collector.

In this paper, we present a Python extension based on SharedArray [3] that enables the support of system-provided shared memory for Python arrays, and its integration into the PyCOMPSs programming model. This demonstrates how the CPython interface in conjunction with the NumPy library can provide tools for memory management outside of the Python interpreter, and how to integrate it in a complex framework. We will also show how these capabilities can optionally be manually tuned by the user with Python metaprogramming features. Related work is presented in Section II, while the design and implementation details will be explained in Section III. Section IV will present the performance evaluation of our solution over the original PyCOMPSs version and we will conclude and present further work in Section V.

II. Related Work

Multiple approaches for shared memory have been proposed in the literature. OpenMP [4], [5] supports multi-platform shared-memory parallel programming in C/C++ and Fortran. The OpenMP API defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer. Since OpenMP is a shared memory programming model, most variables in OpenMP code are visible to all threads by default. OpenMP was extended to support tasks and its data dependencies. The tasks, as other OpenMP constructs, can operate in private arguments, but has also access to the shared variables.

OpenSHMEM [6] is an effort to create a specification for a standardized API aiming at unifying the different SHMEM libraries available. SHMEM is a communications library that adopts the Partitioned Global Address Space (PGAS) programming models. The key features of SHMEM include one-sided, point-to-point and collective communication, a shared memory view, and atomic operations that operate on globally visible, or "symmetric" variables in the program.

However, previous approaches do not support the Python programming language. In Python there are two main issues to overcome when accessing data from parallel tasks: one is the Global Interpreter Lock (GIL), a mutex that protects accesses to Python objects, preventing multiple threads from executing Python bytecodes at once [7]. The second issue is related to the impossibility of accessing Python objects with a reference address. This is only possible with NumPy objects, which can be accessed through a link in a C memory space.

The Python 3.8 multiprocessing module provides the SharedMemory class for the allocation and management of shared memory to be accessed by one or more processes on a multicore or symmetric multiprocessor (SMP) machine[1]. The class permits distinct processes to potentially read and write to a common (or shared) region of volatile memory. The design is similar to the solution we used, both for the shared memory and the integration with NumPy. However, we wanted fine control over the memory management (authorizing read-on-write with no synchronization) and the sharing mechanism (System-V or file backed-up with a customizable path). We also wanted portability with Python 2, hence our custom approach with a python extension.

Python 3.8 also provides other means of data sharing between processes and their children. Although all workers are spawned from a common *parent* process, using it to check the type of variables in order to add them in shared memory would have been a breach of abstraction. It also would have been needlessly complex and inefficient as it would have added an extra access to the serialized version on file to check on the types.

The Plasma In-Memory Object Store[2] is another approach that supports holding immutable objects in shared memory so that they can be accessed efficiently by many clients across process boundaries. Plasma supports two APIs for creating and accessing objects: a high level API that allows storing and retrieving Python objects and a low level API that allows creating, writing and sealing buffers and operating on the binary data directly. A drawback of Plasma is that it does not support user defined objects, and our tests with PyCOMPSs did not succeed.

[1]https://docs.python.org/3/library/multiprocessing.shared_memory.html

[2]https://arrow.apache.org/docs/python/plasma.html

III. Design and Implementation

This section will present the design that drove the implementation decisions and the preexisting framework on top of which the solution was developed.

A. PyCOMPSs

PyCOMPSs is the Python binding for COMPSs, a task-based parallel programming model for distributed computing platforms. In such paradigm, the unit for parallelism is the *task*. Tasks are identified by the application programmer, who also indicates the *directionality* of the task parameters and, if required, other metadata such as the type or collection size when applicable. The parameters directions can be *IN*, *OUT* or *INOUT*. Some of the metadata can be inferred by the static analysis of the code, but sometimes it is mandatory for the user to make it explicit. With this information, the COMPSs runtime builds at execution time a task graph, where nodes denote task instances and edges data dependencies between tasks. From this task graph, the COMPSs runtime is able to decide which tasks can be executed in parallel and which ones sequentially. It then performs all the actions required to execute the application tasks in the computing platform. The COMPSs runtime is deployed as a master-worker application, by instantiating a master process in one node and multiple worker processes in different nodes. The master orchestrates the application execution and makes decisions while the application tasks are executed by the worker processes.

COMPSs runtime is written in Java, but expose layers of binding for C and Python. The Python specific binding is called PyCOMPSs. As previously stated for COMPSs, PyCOMPSs applications can be executed in distributed computing platforms (i.e., clusters or clouds). Yet, the COMPSs runtime gives the programmer a virtual single memory space. To this end, the COMPSs runtime transfers the data from one node to another when needed. In order to reduce the number of transfers, the locality is exploited as much as possible. However, when the application is being run across multiple nodes, they do not share a common shared memory space.

While in Java we deploy a single process per node with multiple threads, in the case of Python applications the process of transferring data between tasks is especially delicate. Due to the Global Interpreter Lock (GIL), which prevents good parallelization schemes when using threads as the access to the critical resource serializes the executions, we opted to start multiple Python processes in each of the worker nodes. Hence, the different Python processes do not share the same memory space. As a result, in order to be able to transfer data between

two processes, the object to be sent from one task to another is serialized and written into a file. The task that needs the object, will read the file and deserialize the object. Additionally, if the processes are in different nodes, the file containing the serialized object needs to be transferred from one node to the other.

Thus, any optimization to sharing data within one node is critical to the PyCOMPSs worker performance, and providing node-local, system-wide shared memory support could be greatly beneficial for the application's performances.

The task themselves are distributed to the different workers by the Java master. The tasks details, such as function name, parameters' value or filename, are sent to one specific python instance on each nodes that is responsible for distributing the tasks to the members of the Python multi-process environment. This process is also responsible for the synchronization and the termination of the task executing processes. The scheduling decisions depend on availability of workers and on the data already held by them from previous tasks. However, currently there is no weighting of the cached data depending on their size. Hence, a single-entry array weight the same as a 1 GB array in the decision making process.

1) Objects as files: As the COMPSs model requires communication channels across nodes and across languages and software stacks, it needs a portable way to exchange data. In addition, the synchronization between tasks is done based on data, which may need to be buffered after being generated and before getting used. To solve this issue, the framework serialize the data into files, and name them following a naming scheme which reflects both the unique data identifier and its *version*. The unique data identifier is unique across the whole application. The *version* of data is a counter that get incremented every time the data is modified. That way, tasks depending on the new state of the data can still be spawned while not all tasks depending on previous data version have started yet.

Although providing a lot of flexibility, this management also limits the possibility to use shared memory based on the data identifier and version unique key. In order not to overwrite data, any memory sharing capabilities would be have to maintain this versioning capability, or would need to limit it to read-only variables.

One other constraint to the shared memory extension is the capacity to release the memory when it is not needed anymore. As scratch space is limited, so is system-provided shared memories. If the data is mapped by the system and backed by the file, it may need some costly input-output (I/O) operation from memory to disk when physical memory is necessary. If the data is mapped using the SHM POSIX API, the data is pinned and cannot be swapped. In addition, in the latter case, because of the impossibility to apply swap operations, the total amount of shared memory available is limited (although configurable by the system administrator). Whether for costly operation avoidance or for resource freeing, the need to release memory on demand or when the data is not to be used anymore is essential. COMPSs uses the

```
@task(c=INOUT)
def multiply(a,b,c):
    import numpy
    c += a*b
```

(a) Task decorator before. The type and directionality for `a` and `b` are automatically inferred.

```
@task(a={Type:IN,RRO:True},
      b={Type:IN,RRO:True},
      c=INOUT)
def multiply(a,b,c):
    import numpy
    c += a*b
```

(b) Task decorator with *recurrent read-only* flag enabled. `RRO` can be replaced with the string "`recurrent_read_only`".

Fig. 1: Example of usage of the new decorator.

versioning of data and the task scheduler information to release unnecessary memory and space scratch-memory. PyCOMPSs also expose an API to explicitly request the deleting of a file or an object, effectively releasing the resources held.

2) Python Decorator: As part of its model, PyCOMPSs is using Python decorators in order to annotate functions and describe some characteristics that cannot be inferred. The **`task`** decorator selects the methods that will become tasks at execution time and gives hints about the parameters, leading to a better management of these. Describing the directionality of data, the availability or the type of data, or the data structures are examples of metadata to be added using the decorator. This metadata collection has been extended to request the data to be marked as *recurrent read-only* (see Fig. 1). The annotated Python code is said *taskified*, as the selected method will be distributed to the different workers at run-time.

This flag is only applied to objects which are instances of NumPy class *ndarray* or whose class inherits from NumPy class *ndarray*. Task arguments marked as such will be loaded into the system shared memory or retrieved from it if previously added. This provides much better performance to the whole system compared to adding every array-based object, as the loading to the shared memory adds an overhead to the inevitable deserialization. Details are provided in Section III-B1.

This extra cost, however, can be mitigated by a high reuse of the data. From the original PyCOMPSs interface, each task parameter has to be deserialized when starting a new task, potentially read from file. Applications that rely on multiple reads of the same data through successive iterations (e.g., for data mining applications) can gain in performance when run as tasks using PyCOMPSs as shown in Section IV. Moreover, the addition of a simple decorator keeps the high productivity provided by the programming-model [8].

3) Internal Dictionary: In order to keep track of each shared memory segments, each worker has its own lookup-table to keep track of what data are already mapped in memory. Each worker being its own Python process, there is no need for synchronization as the memory address space is not shared. Following the Pythonic way, the testing for an existing reference is done by first looking in the dictionary if the entry exists. If not, we try to load from shared memory the requested entry. The shared memory allows us to access data that would have been deserialized and shared by an other process. If the target entry does not exist, an exception is raised

by the runtime. This exception triggers the deserialization of the requested array and the creation of the proper shared memory segment. There is no synchronization across the different workers on one node. However, as the data deserialized is necessarily the same because of the naming scheme, a read-on-write operation would not risk creating incoherent state. If the array had to be either loaded from memory or deserialized from file, then the corresponding entry is added to the lookup-table. Each entry is indexed by its runtime-defined file name, based on its identifier and its version. This name is ensured to be unique by the runtime, and kept across tasks if the variable is only read during concurrent or previous tasks. If the variable is modified, the name of the variable is ensured to be different as explained in Section III-A1. This dictionary is also used in order to deregister all memory segments on request or once the application terminates.

B. SharedArray, a Python extension

In order to provide the persistence on the working nodes and the sharing capabilities between processes, an external library was required to interface with the operating system. The library SharedArray provided most of the characteristics we could hope for. This Python's module provides the interface to create arrays shared either via the POSIX SHM API, or with the memory mapping of a file. The library uses the NumPy [9] library with a CPython interface. However, some limitations made it unsuitable in its current state.

NumPy is a library widely used in order to improve the performance of Python applications. It provides an interface to interact with the data without requiring any copy to memory in addition of using a contiguous buffer. This buffer can be externally provided using the C API of the library. The CPython part gives a native interface between C and Python, and is the entry point to any C-based extension library for Python.

1) Module API extension: In order to give access to system's shared memory to Python applications, an extension to the base language was necessary. While [3] was quite thorough, it only allows the creation of zero-initialized arrays. A decision was made to extend this library in order to provide an enriched API that adds a copy constructor to NumPy based arrays.[3] The CPython library provides access to the internal state of the variables, objects and arrays inside the Python interpreter, and allow the library to tempered with them. This allowed us to create a new object of class *ndarray* with its contiguous buffer pointer referencing our newly allocated shared memory region instead of the original buffer. The shared buffer is initialized with a copy of the values contained in the original array whose internal reference counter is decremented. However, this implies that the original array used for the deserialization from file still requires to be freed with its corresponding internal data structures.

In addition to the copy constructor modification, the behaviour for preexisting names had to be changed as well. Previously, an exception was being raised when a name was already taken while registering an array. Avoiding this exception to be raised would have required an external synchronization on the Python side and the addition of a global lock at the node level to ensure mutual exclusion when accessing and loading objects to memory. Moreover, as the shared memory can be based on the mapping of a file, which could belong to a parallel file system, the synchronization would have required to be done across all Python workers, potentially across multiple nodes. This would have added unnecessary complexity to the framework which would have suffer a decrease in parallelism. As presented in Section III-A3, and because the read-on-write risk being limited, we decided to keep the race condition on deserialization. The same file being deserialized, data coherency is ensured although the location is shared, so we did not enforce unnecessary synchronization.

2) Shared Memory Model: There are many standard ways of using system's shared memory. We focused our choice between two. The first method is using `mmap(2)` to create a file-backed up shared segment. The processes are mirroring the file, mapping it into their virtual memory space. Any modification to the memory is eventually propagated to the file to keep coherency between processes. The second method is using `shm_open(2)` to create a memory segment held by the system that is persistent after program ending if not freed. The newly memory page created by the system can then be mapped into multiple processes' memory space.

Although the `shm_open(2)` shared memory implementation exposes limitations unlike with `mmap(2)`, the higher performance allowed by the absence of disk operations inclined us to choose the former. It also has the advantage of resolving issues due to potential parallel file system name conflicts across cooperating nodes and high latency induced by the environment. Moreover, using the `shm_open(2)` shared memory API imposes restrictions on the maximum number of segments that can be kept at one time, and the maximum size of one segment. The values can be usually found in the `/etc/sysctl.conf` system file, and default to $2\,097\,152$ pages of 4096 bytes which limits it to 8 gigabytes (GB), across 4096 segments of shared memory. But these limitations, either imposed by the operating system configuration or by the environment, can be overcome by having the settings changed by the system administrators.

IV. Results

The tests were executed in the MareNostrum III cluster [10], hosted at the Barcelona Supercomputing Center. The codes ran on two nodes. One is the master node, run in exclusive mode in order to avoid sharing resources with any worker, while the second node executes the tasks with 16 processes (workers), one process per core. Each node features two Intel SandyBridge–EP 20M E5–2670/1600 8-cores at $2.6\,\mathrm{GHz}$. As the design relies on the system shared memory, the decision was made to restrict the execution to one worker with 16 processes on a single node as all cores would be used without over-subscription and it would show best the limitation of

[3]The extended version of the SharedArray is publicly available at https://gitlab.com/cerl/third-party-contributions/shared-array.

the system if attainable. The objective of the experiments is to show an improvement in execution time by reducing the overall time required for deserialization. In order to limit the influence of external parameters that would add noise to the time measured, all data are stored locally on the nodes disks, without using the parallel file system.

The first application to be tested was k-means, as the memory access is simple and quickly shows the improvement that can be achieved with careful selection of data for reuse. The second test-case is a task-based blocked 2D matrix multiplication, as it presents some advantages to reuse data but with a more complicated data access pattern.

A. K-means clustering application

K-means is a widely used clustering algorithm often used by machine learning applications. This algorithm is a numerical, unsupervised, non-deterministic, iterative method that partitions a set of points into k clusters, centered on one point. Each point belongs to exactly one cluster and contributes to the *center*'s position.

K-means applications have many parameters that influence the behavior of the application. For the purpose of testing, we have limited to four variables. These variables were chosen as the were expected to present the most significance in showing the effect of our contribution. We decided to run the application with a fixed number of *fragments*, corresponding to the number of processes acting as workers. Each *fragment* represents a subset of the full set of input points. It is an arbitrary selection of $\frac{\text{number of points}}{\text{number of } \textit{fragments}}$ points to distribute evenly the work-load between tasks. Further work could involve testing the influence of increasing the number of *fragments* without modifying the number of workers and the effect of loading from the system shared-memory compared to loading from the file-system. However, this test requires proper management of the task scheduling as it would require the task to be loaded to a new worker in order to expose the need for deserializing the corresponding subset of points before its addition to the internal dictionary.

The details of the different test-cases and their timings, both with and without our extension, are gathered in Table I. We tested separately the variation of the number of points, of the maximum number of iterations, of the number of dimensions and of the number of centers. We present in the table the average timings, but also the $95\,\%$ confidence interval of the difference in mean, along with the p-value, calculated with the Welch's t-test.

The default number of *centers* is set to 4. This parameter influences the amount of computations and the time spent in each task, as each point has to be compared to each *center* in order to define its cluster. Increasing the number of *centers* increases the time required per iteration but has very little influence on the time required for serializing and deserializing the data. Although, the *centers* also need serialization and deserialization; their number is usually negligible compared to the number of points per *fragment*. In addition, the *centers* being modified between two iterations, the framework would not allow any gain from reuse of memory and the array would always need to be deserialized every time. However, diminishing the number of *centers* increases too much the risk of early termination because of the convergence criterion. The dummy test-case 1 from Table I that only requests one *center* shows this effect, as the convergence happens in two iterations, as shown in the traces gathered (Fig. 2a). Our study does not show much influence of the number of *centers* in percentage of improvement (cases 1 to 5) as the number of *centers* is orders of magnitude lower than the number of points. We expect the performance improvement to be degraded for a smaller $\frac{\text{number of } \textit{point}}{\text{number of } \textit{centers}}$ ratio as the fraction part of total time dedicated to I/O will decrease as well.

The maximum number of iterations influences the overall time of the application. For a data-loading/unloading dominated application such as k-means with PyCOMPSs, this parameter would artificially increase the performance gain. In our case, the increase of the maximum number of iterations (case 14) only increases the application running time, with little difference in performance compared to the reference (case 4). The difference in means changed from 9–$10\,\%$ for case 4 to 10–$11\,\%$ for case 14.

Finally, the last two parameters are the number of points and the number of dimensions. These parameters have a direct influence on the performance gain as they both define the amount of data to be loaded, and thus, the potential improvement by keeping this data in memory instead of loading and unloading it at each iteration. *point* and *center* are vectors of doubles. Each vectors' length equals the number of dimensions. The number of dimensions usually represents different variables influencing the points being clustered. The number of dimension only influences the amount of data in the *fragment* and *centers*. However, the amount of points also influences the amount of data to be loaded when returning the *labels* array which contains the affiliation of each point to one of the *centers*.

The points used for the tests were generated randomly, with a constant seed shared across cases. The points were generated using uniform random number generator. The affiliation criterion was computed by finding the minimum Frobenius norm between a point and each of the *centers*. All the points were generated with the same seed across cases and runs. The epsilon distance used to evaluate the convergence criterion of the centers was set to 1×10^{-9}.

The algorithm used to taskify and parallelize the k-mean application is the same as the one used in [8] and shortly presented in a simplified version in Algorithm 1 for the record. The distributed part of the algorithm is the call to the **`cluster_partial_sum`** function. The reduction of the array of *centers* into the accumulation variable $\texttt{centers}_{acc}$ is serialized on the master side, as well as the concatenation of the *label* lists. The function returns the computed *centers* along with the association between *points* and *centers*, data carried by *labels*.

The algorithm execute three main steps. First, **`generate_fragments`** randomly generates the points of all

TABLE I: Raw results for K-Means clustering application

ID	number of points	max. iter.	dims	centers	time with (in seconds)	time without (in seconds)	95 % confidence interval inferior and superior in seconds		in percentages (%)		p-value
1	4 194 304	20	64	1	79.420	86.542	−8.228	−6.017	−9.51 %	−6.95 %	1.167×10^{-20}
2	4 194 304	20	64	2	627.472	733.311	−112.697	−98.981	−15.37 %	−13.50 %	1.700×10^{-23}
3	4 194 304	20	64	3	760.308	870.277	−114.208	−105.730	−13.12 %	−12.15 %	2.596×10^{-70}
▷ 4	4 194 304	20	64	4	890.667	987.643	−102.498	−91.455	−10.38 %	−9.26 %	1.233×10^{-50}
5	4 194 304	20	64	5	1027.569	1138.324	−117.625	−103.885	−10.33 %	−9.13 %	1.612×10^{-53}
6	4 194 304	20	8	4	862.735	858.778	−3.297	11.211	−0.38 %	1.31 %	0.271 527 7
7	4 194 304	20	16	4	859.134	871.523	−18.621	−6.156	−2.14 %	−0.71 %	1.891×10^{-4}
8	4 194 304	20	32	4	873.129	872.752	−4.720	5.473	−0.54 %	0.63 %	0.883 804 2
▷ 4	4 194 304	20	64	4	890.667	987.643	−102.498	−91.455	−10.38 %	−9.26 %	1.233×10^{-50}
9	4 194 304	20	96	4	904.989	1331.601	−434.343	−418.880	−32.62 %	−31.46 %	1.950×10^{-89}
10	4 194 304	20	128	4	941.375	1585.822	−651.557	−637.337	−41.09 %	−40.19 %	2.134×10^{-86}
11	4 194 304	10	64	4	456.969	513.037	−59.443	−52.693	−11.59 %	−10.27 %	3.769×10^{-52}
▷ 4	4 194 304	20	64	4	890.667	987.643	−102.498	−91.455	−10.38 %	−9.26 %	1.233×10^{-50}
12	4 194 304	30	64	4	1332.570	1489.920	−180.864	−133.836	−12.14 %	−8.98 %	2.337×10^{-19}
13	4 194 304	40	64	4	1733.928	1955.808	−233.903	−209.855	−11.96 %	−10.73 %	5.183×10^{-56}
14	4 194 304	50	64	4	2181.225	2443.273	−276.564	−247.531	−11.32 %	−10.13 %	8.564×10^{-55}
15	2048	20	64	4	6.647	6.684	−0.189	0.116	−2.83 %	1.73 %	0.631 266 2
16	32 768	20	64	4	13.852	14.161	−0.726	0.108	−5.12 %	0.76 %	0.143 467 4
17	262 144	20	64	4	60.330	60.468	−0.525	0.249	−0.87 %	0.41 %	0.480 262 9
18	524 288	20	64	4	117.406	120.133	−12.519	7.064	−10.42 %	5.88 %	0.580 506 7
19	1 048 576	20	64	4	229.888	225.582	2.992	5.619	1.33 %	2.49 %	3.280×10^{-9}
20	2 097 152	20	64	4	448.796	447.682	−0.669	2.895	−0.15 %	0.65 %	0.219 859 5
▷ 4	4 194 304	20	64	4	890.667	987.643	−102.498	−91.455	−10.38 %	−9.26 %	1.233×10^{-50}
21	8 388 608	20	64	4	1800.803	2408.625	−619.544	−596.100	−25.72 %	−24.75 %	4.740×10^{-94}
22	16 777 216	20	64	4	3491.839	4835.420	−1367.719	−1319.443	−28.29 %	−27.29 %	7.419×10^{-121}
23	33 554 432	20	64	4	7076.171	9737.459	−2710.143	−2612.435	−27.90 %	−26.76 %	8.803×10^{-64}

The maximum number of iterations is the number of iterations of the algorithm if no convergence between the centers happens first. The confidence interval correspond to the 95 % confidence interval of the difference in mean of each subgroup (*time with* or *time without*). The difference in percentage is relative to the base time, i.e., time without the usage of shared memory.

fragments which will be distributed to the different tasks. The centers are initially common for all fragments, although each application of **`cluster_partial_sum`** will modify them. **`cluster_partial_sum`** is used in order to compute the *labels* for each point and the *center*'s position based on the clustered points. The position of one cluster *center* is the barycenter of the *fragment*'s points belonging to this cluster. Finally, the *centers* positions are reduced by calculating the means of the *centers* coordinates, for each *center*. Hence, $\texttt{centers}_{acc}$ contains the mean of the means of each fragment clusters. If between two iterations, the difference between the previous and the new position of all *centers* is below the threshold ε, the function returns without executing the remaining iterations.

The function **`cluster_partial_sum`** takes as parameters one *fragment*, the corresponding *labels* and the redefined *centers*. Only the *fragment* is defined as *recurrent read-only* and hence is loaded into shared memory. Contrary to the algorithm presented, the *labels* are modified by side effect while the updated *centers* are returned by the function. However, because the execution of the function is realized by a worker, and as *labels* and *centers* are collections, their input and output are handled with the data being serialized to disk, waiting for the master to deserialize them, due to COMPSs behavior. The timer starts just after the generation of the *fragments* and finishes when the main loop finishes (either by running out of iteration or by meeting the criterion of convergence), after all the *labels* have been gathered.

Table I presents the raw results from the application. Each cases were run at least 50 times both with and without the use of shared memory. Grey lines (cases 6, 8, 15 to 18 and 20) present test-cases where the difference in mean of the timing in the method that uses the shared memory is not significantly different from 0, meaning that there are no statistically significant difference between the timings due to standard variability (p-value <0.05). Red lines (case 19) are for cases where the difference in means is significantly greater than 0, meaning that there is a negative effect in performance when using the shared memory that is likely not due to standard variability. Green lines show cases where the performance improvement is statistically significant.

Cases 11 to 14 show that changing the number of iteration does not affect the gain in performance in proportion. The benchmark execution time is already dominated by the execution time of the iteration, which performance is improved proportionally to the improvement of the data deserialization part. Even the dummy-case 1 which finishes early after only 2 iterations shows a difference in mean close to 10 %. The 2–5 %

Algorithm 1 Main loop for distributed k-means application

Input

N	Total number of points to cluster
k	Number of clusters
f	Number of fragments
max_iterations	Maximum number of iteration
ε	Convergence criterion

Output

$\texttt{centers}_{acc}$	Vector of centers
labels	Association between points and centers

1: $(\texttt{ctrs}, \texttt{lbls}) \leftarrow$ **generate_centers**(k)
2: fragments $\leftarrow$ **generate_fragments**(N, f)
3: **for** iter $\leftarrow 1$ **to** max_iterations **do**
4: $\quad \texttt{centers}_{acc} \leftarrow \overrightarrow{0}$
5: $\quad$ labels $\leftarrow$ **empty_list**()
6: $\quad \texttt{centers}_{old} \leftarrow \texttt{ctrs}$
7: $\quad$ **for all** frg $\in$ fragments **do**
8: $\quad\quad$ **cluster_partial_sum**(frg, ctrs, lbls)
9: $\quad\quad$ **list_append**(labels, lbls)
10: $\quad\quad \texttt{centers}_{acc} \leftarrow \texttt{centers}_{acc} + {}^{1}/_{\texttt{f}} \cdot \texttt{ctrs}$
11: $\quad\quad \texttt{ctrs} \leftarrow \texttt{centers}_{old}$
12: $\quad$ **end for**
13: $\quad$ **if** $\|\texttt{centers}_{acc} - \texttt{centers}_{old}\| > \varepsilon$ **then**
14: $\quad\quad \texttt{ctrs} \leftarrow \texttt{centers}_{acc}$
15: $\quad$ **else**
$\quad\quad$ ▷ Early exit when convergence criterion is met.
16: $\quad\quad$ **break**
17: $\quad$ **end if**
18: **end for**
19: **return** $(\texttt{centers}_{acc}, \texttt{lbls})$

difference can be explained by the original sending of data to the workers which cost is no longer negligible compared to the time to run the iterations.

The cases 6 to 10 and 15 to 23 show the impact of the amount of data to be loaded (by increasing either the number of points or the number of dimensions). In most of cases past some threshold, the more data, the more improvement can be achieved. Also, the performance improvement seems to have a stronger scale with the number of dimension than with the number of points. As a matter of fact, doubling the amount of data by doubling the number of points boosted the gain in performance from 9–10 % to 24–25 % while doubling the number of dimensions increased the gain up to 40–41 %. One reason is that increasing the number of points also increases the number of *labels* to serialize and deserialize for each iteration, which impact negatively the performance. It also seems to appear that the gain in performance can be bounded as cases 22 and 23 both expose a gain around 25–28 %. However, we could not verify this as it would require further testing.

For cases 6, 8 and 15 to 20 the method to be used to load data does not seems to be very influential as the amount of data (below 64 MB per task) is too small to see any impactful improvement by reuse of memory. Hence the difference in mean would only increase the variance of the timing distribution, participating in the noise of the measures. However, the test case 19 shows that it can be disadvantageous to use the shared memory. The shared memory execution path efficiency relies on the condition that the look-up through one table or the access to the system shared memory is more efficient than accessing data on disk through standard NumPy deserialization. Although it may be an artifact in the execution of the benches, this test case shows that this condition is not necessarily always met, but we have not manage to determine the reason for this behavior for this set of parameters. In this underperforming case, the overhead still stays very low, below 2.5 %. The comparison between cases 6 to 10 and cases 15 to 20 are showing that the amount of data, especially the ratio of *read-only* memory over total memory to load is critical to finding cases where improvement can be reach using shared memory. This ratio limits the grows in performance to be expected, hence, multiplying by a factor 4 the threshold before seeing substantial gain in performance.

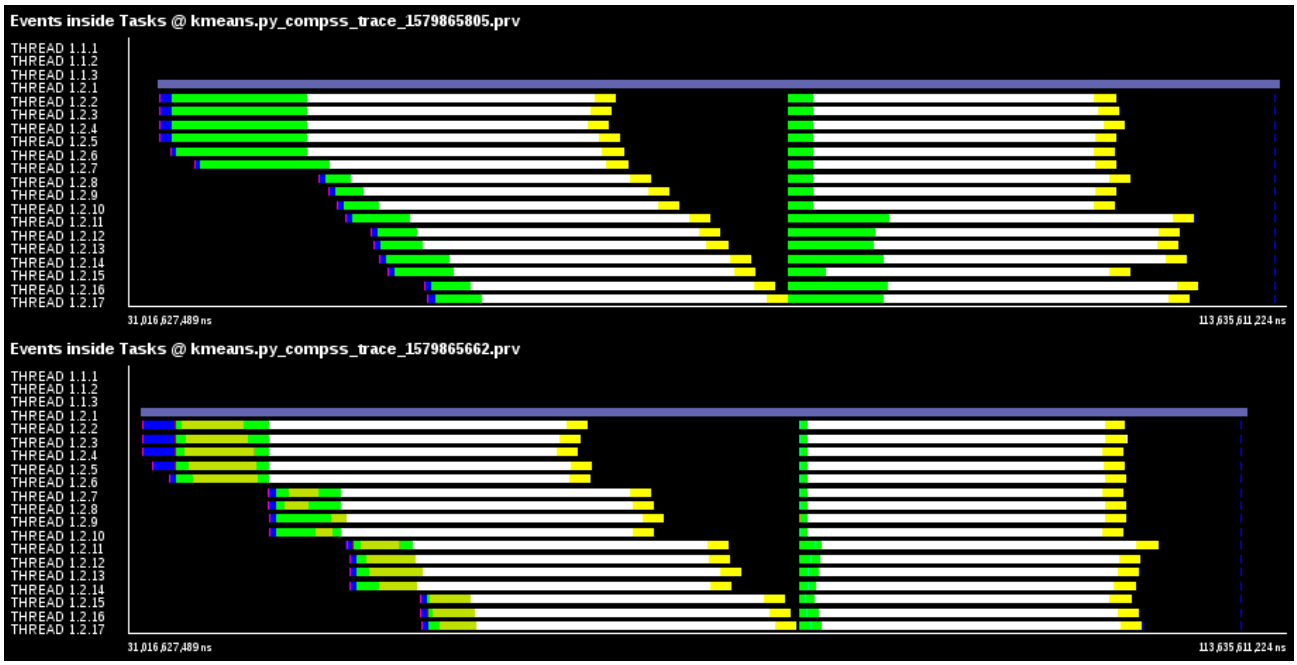
Fig. 2 presents screenshots of postmortem visualization of k-means application traces using the software PARAVER [11]. Fig. 2a shows that although the first iteration deserialization gets slightly longer because of the loading of data to the shared memory (light green at the origin of each segment), from the second iteration onward the deserialization is greatly reduced. The darker shade of green on the lower part of Fig. 2a at the beginning of each segment of the first column corresponds to the time spent charging the data into shared memory.

As shown in Fig. 2b, the overhead that was required on the first iteration gets evened out by quick data recovery in shared memory, and by even quicker data load from the internal dictionary.

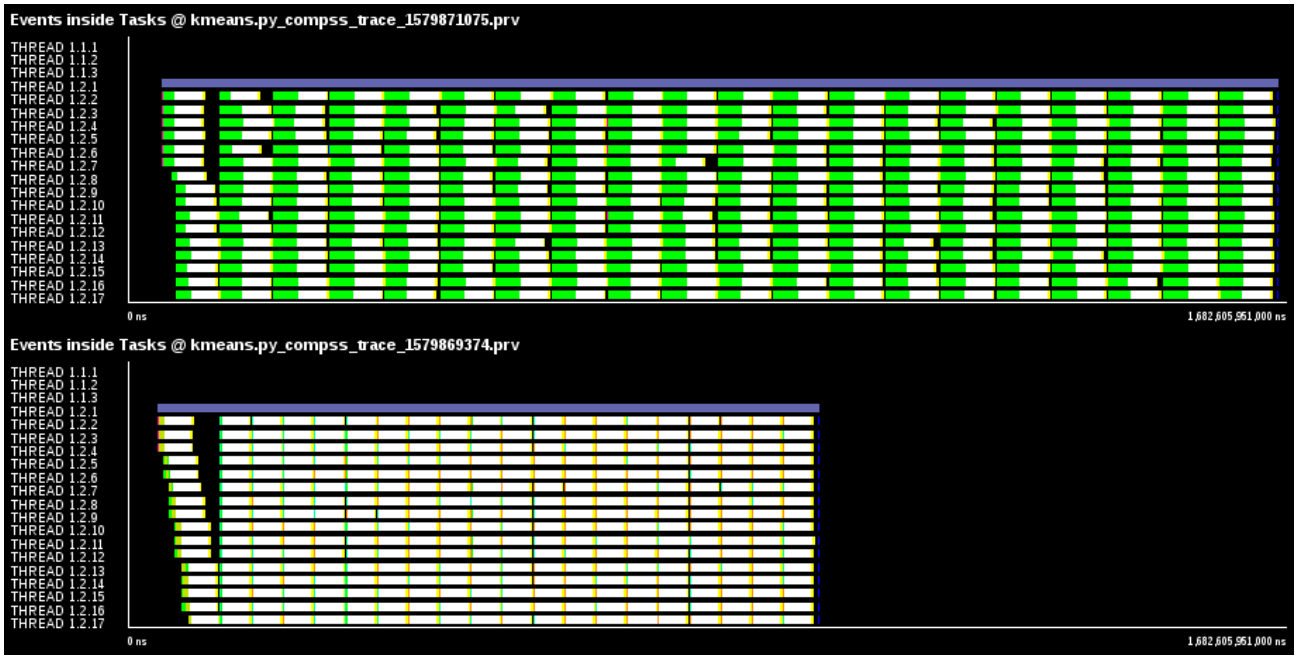
B. Blocked Matrix Multiplication

The second case considered is a task-based version of the blocked 2D-matrix multiplication. The main algorithm is depicted in Algorithm 2. The data dependency expressed on the resulting matrix C is detected automatically by PyCOMPSs and managed with the COMPSs scheduler. Each matrix is composed by sub matrices indexed by their row and column numbers, starting at 0. All blocks are square and share the same dimensions, expressed in amount of double precision floating point numerical values. During the experiments, the timer was started after the initialization of the matrices, before the main multiplication loop. The call to the **matmul_block** function triggers a task creation that is executed by some worker process on the second node. We stopped the timer once all tasks were finished, after a synchronization barrier. As expressed in the input parameters of Algorithm 2, only the two read-only input matrices A and B are using the shared memory capabilities.

For these experiments, we had two parameters to vary. The number of blocks per matrix dimension and the number of elements per dimension of each block (reported as block size).



(a) Parameters similar to test-case 1, convergence in 2 iterations.



(b) Over the time of the application, reuse of data makes the cost of data serialization insignificant. Parameters are similar to test-case 10.

Trace shown are from worker cores only. The colors are given chronologically, from left to right. Green segments represent the deserialization of objects, darker green being the creation and population of the shared memory segments. The large white segments in the middle represent the user code execution. Yellow segments represent the serialization of objects. As shown, not all objects are added to shared memory, hence the deserializations timing hardly appear on the second traces shown in Figure 2b.

Fig. 2: Traces of k-means PyCOMPSs application.

Algorithm 2 Main algorithm for blocked matrix multiplication

Input
- $\texttt{dim}$ Matrices dimension
- A, B Square 2D matrices, shared
- C Square 2D matrix, not shared, zero initialized

Output
- C Result matrix

```
for r ← 0 to dim − 1 do
    for c ← 0 to dim − 1 do
        for i ← 0 to dim − 1 do
            C_rc ← matmul_block(A_ri, B_ic, C_rc)
        end for
    end for
end for
```

The former influences the number of tasks generated in the application, while the latter modifies the amount of data to be loaded when deserializing a block. It also have a direct impact on the ratio $\frac{\text{user code execution time}}{\text{data and task management time}}$, which limits the proportion of code that can be improved with our addition to PyCOMPSs. As we are trying to show an improvement in the overhead induced by the programming model, increasing the number of elements too much can make the improvement disappear in comparison to the overall application time.

The preliminary parameter exploration showed us that the most interesting performance test-cases included 8, 12, 16, 20 and 24 blocks in each dimension, with 128, 512, 1024, 2048 and 4096 doubles per dimension for each block. Each case was ran 50 times with and 50 times without the usage of shared memory. The different parameters tested and their results are reported in Table II. The results exposed are similar as in Table I, namely average timings with and without the extension, $95\,\%$ confidence interval of difference in mean and p-value. For the sake of brevity and clarity, only the statistically significant results are shown (p-value <0.05). For a block size of 4096 elements, only the 8 blocks case gave results, as memory exhaustion lead Python to rise exceptions when trying to execute larger cases. Thus, we were unable to run comparative tests, but we believe finer control over the memory allocation would make it possible to run larger cases.

From the 21 test-cases, 9 do not show any statistical difference in means when compared with a Welsh's t-test. $58\,\%$ of the remaining cases show an improvement in performance, ranging from $<1\,\%$ to $16.65\,\%$ for improved cases. The worst penalty measured is an overhead of $14.07\,\%$ in case 5. For cases 1 to 5, the differences in mean are showing respectively $3.64\,\%$, $5.4\,\%$, $5.3\,\%$, $7.9\,\%$ and $10\,\%$ of timing increase when the application is being run with the shared memory.

The variability in the results is to be explained from a lack of fine grain control over the tasks attribution. This reduce the direct reuse of previously deserialized matrices that are stored in the internal dictionary. The creation of a shared-memory array requires to execute the memory allocation twice, once from file to memory, and once from memory to shared memory, and needs the data to be written both times. We analyse that this overhead can only overcome its cost with sufficient reuse of memory pages. It appears that bigger sizes lead to better results, as the loading from the disk can throttle the performance compared to memory mapping.

We are planning on future work to include testing with regard to fine grain task scheduling in order to both reduce the variability of measurements and to try to define an optimal scheduling. As a matter of fact, a fine grain management of task scheduling that would prioritize tasks with disjoints data sets would improve performance as fewer shared memory blocks would be rewritten in case of data names conflict. In the case of matrix multiplication, all blocks have to be used multiple times. On a single node, one optimized algorithm could be to first compute the blocks along the diagonal to maximize the number of array ready to be reused, and to minimize the number of array loaded twice to shared memory. The workers could be execute computing operations following either the row in order to reuse the blocks deserialized from matrix A and then compute blocks from the column in order to reuse the blocks deserialized from matrix B.

Although two thirds of the cases are showing an improvement when using shared memory, the selection of good candidates is a difficult issue for applications with a complex pattern of memory accesses. As an example, increasing the number of blocks increases the parallelism of the application and the number of times the data have to be used. But it also increases the number of times the output matrices have to be serialized and deserialized. As shown in cases 6 to 8 and cases 9 to 11, for one given block size, the increase of number of blocks decreases the gain of performance, from a maximum of $11.45\,\%$ (case 6) to a minimum of $2.15\,\%$ (case 8) on average for a block size of 1024 and from a maximum of $15.20\,\%$ (case 9) to a minimum of $8.81\,\%$ (case 10) on average for a block size of 2048, respectively.

V. Conclusion

This work presented the integration of an extension to the Python language features into the PyCOMPSs framework. Analogously to the original work, the code modification required to use this new trait is kept to a minimum in order to make code adaptation as easy as possible and provide a seamless integration into the parallel framework. For algorithms based on a high number of reuse of data, such as k-means, performance have been shown to be improved by at least $\approx 10\,\%$ or unaffected by using shared-memory, in our condition of experiment. For improved cases, the effect is amplified as the amount of data increased, proportionally to the total amount of I/O, with an improvement of $\approx 40\,\%$ when increasing the number of point dimensions in the case of a k-means application. However, for a too small task granularity there can be an antagonistic, yet relatively limited, effect. Nonetheless, the promising results based on read-only memory may lead to a possible evolution of the framework allowing a better usage of shared-memory when distributing tasks to different workers sharing one same node, to reduce even more the number of serializations required, for application behaving

TABLE II: Raw results for blocked matrix multiplication.

ID	number of blocks	block size	time with shared mem.	time without shared mem.	95 % confidence interval inferior & superior in seconds		in percentage		p-value
1	16	128	43.624	42.093	0.034	3.027	0.08 %	7.19 %	0.045 101 7
2	8	512	18.287	17.343	0.088	1.800	0.51 %	10.38 %	0.031 097 6
3	12	512	51.527	48.920	0.717	4.496	1.46 %	9.19 %	7.366×10^{-3}
4	16	512	121.165	112.291	3.964	13.784	3.53 %	12.28 %	6.928×10^{-4}
5	20	512	233.181	211.867	12.811	29.818	6.05 %	14.07 %	2.969×10^{-6}
6	8	1024	46.353	52.348	−8.472	−3.519	−16.18 %	−6.72 %	8.376×10^{-6}
7	12	1024	151.787	160.587	−16.390	−1.210	−10.21 %	−0.75 %	0.023 540 3
8	20	1024	741.622	757.913	−25.440	−7.143	−3.36 %	−0.94 %	5.524×10^{-4}
9	8	2048	180.111	204.291	−31.049	−17.311	−15.20 %	−8.47 %	3.446×10^{-10}
10	12	2048	581.257	637.424	−76.866	−35.468	−12.06 %	−5.56 %	5.083×10^{-7}
11	16	2048	1404.155	1551.578	−203.842	−91.005	−13.14 %	−5.87 %	2.381×10^{-6}
12	8	4096	786.623	925.552	−154.124	−123.735	−16.65 %	−13.37 %	8.347×10^{-33}

The number of blocks and the block sizes are the same in each dimension as the matrices are square. Times are given in seconds. The boundaries of the 95 % confidence interval are for the difference in mean of each subgroup (time *with* or *without* shared memory). The difference in percentage is relative to the base time, i.e., time without the usage of shared memory.

like the blocked matrix multiplication benchmark. Larger scale testing of the k-means application could be done for a very large dataset of points to see the integration with storage solutions such as Redis (as presented in [12]) to evaluate the behaviour in case of unsupervized distributed applications and verify whether the solution still provides an improvement of the general performance.

An additional outcome of this work is to display the eligibility of Python based applications and workflows to be executed with memory placement tools and for their performance to be improved with standard memory-oriented optimizations. This second noteworthy result will lead to further research on integration with low level memory management on the strength of the assumptions provided by a higher-level language.

Acknowledgement

This work was partly funded by the EXPERTISE project (http://www.msca-expertise.eu/), which has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 721865.

BSC authors have also been supported by the Spanish Government through contracts SEV2015-0493 and TIN2015-65316-P, and by Generalitat de Catalunya through contract 2014-SGR-1051.

References

[1] F. Lordan, E. Tejedor, J. Ejarque, R. Rafanell, J. Álvarez, F. Marozzo, D. Lezzi, R. Sirvent, D. Talia, and R. M. Badia, "Servicess: An interoperable programming framework for the cloud," *Journal of grid computing*, vol. 12, no. 1, pp. 67–91, 2014.

[2] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta, "Pycompss: Parallel computational workflows in python," *The International Journal of High Performance Computing Applications*, vol. 31, no. 1, pp. 66–82, 2017.

[3] M. Mirmont, "Python extension: Sharedarray," https://pypi.org/project/SharedArray/, 2019.

[4] OpenMP Architecture Review Board, "OpenMP Application Programming Interface Specification," Web page at http://www.openmp.org/specifications/, (Date of last access: 3rd May, 2017).

[5] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[6] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, 2010, pp. 1–3.

[7] M. Driscoll, A. Kamil, S. Kamil, Y. Zheng, and K. Yelick, "PyGAS: A partitioned global address space extension for python," Poster abstract, 2012. [Online]. Available: http://web.eecs.umich.edu/~akamil/papers/pgas12.pdf

[8] J. Álvarez Cid-Fuentes, P. Álvarez, R. Amela, K. Ishii, R. K. Morizawa, and R. M. Badia, "Efficient development of high performance data analytics in python," *Future Generation Computer Systems*, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X18321393

[9] S. J. van der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: A structure for efficient numerical computation," *Computing in Science Engineering*, vol. 13, no. 2, pp. 22–30, March 2011.

[10] BSC - CNS. (2012) MareNostrum 3. [Online]. Available: https://www.bsc.es/marenostrum/marenostrum/mn3

[11] D. Computadors, V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," *WoTUG-18*, vol. 44, 03 1995.

[12] BSC - Workflows and Distributed Computing, "K-means with Redis," https://github.com/bsc-wdc/apps/tree/stable/python/examples/kmeans_redis, 2019.