BigMPI4py: Python module for parallelization of Big Data objects

Alex M. Ascension and Marcos J. Araúzo-Bravo

Abstract—Big Data analysis is a powerful discipline due to the growing number of areas where technologies extract huge amounts of knowledge from data, thus increasing the demand for storage and computational resources. Python was one of the 5 most used programming languages in 2018 and is widely used in Big Data. Parallelization in Python integrates High Performance Computing (HPC) communication protocols like Message Passing Interface (MPI) via mpi4py module. However, mpi4py does not support parallelization of objects greater than 2^{31} bytes, common in Big Data projects. To overcome this limitation we developed BigMPI4py, a Python module that surpasses the parallelization capabilities of mpi4py, and supports object sizes beyond the 2^{31} boundary and up to the RAM limit of the computer. BigMPI4py automatically determines, taking into account the data type, the optimal object division strategy for parallelization, and uses vectorized methods for arrays of numeric types, achieving higher parallelization efficiency. Our module has simpler syntax than MPI4py and warrants "robustness" and seamless integration of complex data analysis pipelines. Thus, it facilitates the implementation of Python for Big Data applications by taking advantage of the computational power of multicore workstations and HPC systems.
BigMPI4py is available at https://gitlab.com/alexmascension/bigmpi4py.

Index Terms—Parallelization, Python, MPI, Big Data.

1 Introduction

The term Big Data has been known since the 1990s [1], and has gained popularity since 2010 due to the exponential amount of data from diverse sources. Until 2007, the volume of all available data was estimated to be slightly less than 300 EB [2]; in 2018, more than 2.5 EB of data were generated daily [3] due to the vast number of users of social networks, online business, messaging platforms, society administration [4], goverment sector aplications [5] and objects belonging to the Internet of Things (IoT) [6]. Big Data elements are also found in many branches of science such as physics, astronomy, omics (like genomics or epigenomics [7] [8]). Omics are currently tightly bound to the rising number of patient records in medical sciences and the application of Next Generation Sequencing (NGS) technologies. The cumulative amount of genomic datasets in the SRA (Sequence Read Archive) database increased to 9000 TB in 2017 [9]. Astronomy also produces vast amounts of data, with databases such as the Square Kilometer Array (SKA) or the Large Synoptic Survey Telescope (LSST), with weights of nearly 4.6 EB and 200 PB, respectively [10]. The quick evolution of Big Data resources has hindered the correct rooting of the discipline, resulting in even a lack of consensus on its definition. Andrea De Mauro et al. [11] considered Big Data as a mixture of three main types of definitions:

- Attributes of data: Big Data satisfy the Laney's "3 Vs" (Volume, Velocity and Variety) [12], and extends 2 more Vs: Veracity and Value. Size of Big Data generation follows an exponential growth associated to Moore's law [13]
- Technological needs: Big Data structures usually require HPC facilities [14].
- Social impact: Big Data is tightly linked to the advance of the society's technology, culture and scholarization.

Big Data is also defined as a consequence of three shifts in the way information is analyzed [15]: (1) All raw data is analyzed instead of a sample. (2) Data is usually inaccurate or incomplete. (3) General trends and correlations overtake analysis of focused characteristics and minute details.

There are multiple computer languages that can deal with the challenges of Big Data. Among them, Python is a dynamic-typed programming language commonly used in Big Data since

- Python is the top ranked language by IEEE Spectrum in 2018 [16] and is the second top ranked by number of active users in Github (14.69%) [17].
- Python is suited both for beginners and experienced programmers.
- Being dynamically typed considerable time is saved in code production at the expense of higher. This effect is reduced with the C-extensions for Python (Cython) and the numba modules integrate C architecture into Python code, drastically diminishing computational times.

- *Alex M. Ascensión and Marcos J. Araúzo-Bravo are with Computational Biology and Systems Biomedicine department, Biodonostia Health Research Institute; P/ Doctor Begiristain, Donostia, Spain, 20014.*
- *Marcos J. Araúzo-Bravo is with the Computational Biomedicine Data Analysis Platform, Biodonostia Health Research Institute; P/ Doctor Begiristain, Donostia, Spain, 20014.*
- *Marcos J. Araúzo-Bravo is with IKERBASQUE, Basque Foundation for Science, Bilbao, Spain, 48013.*
- *Marcos J. Araúzo-Bravo is the corresponding author. Email: mararabra@yahoo.co.uk*

Manuscript received XXX XX, XXXX; revised XXX XX, XXXX.

- Python integrates a vast amount of modules for data analysis, such as pandas, numpy, scipy or scikit-learn, partially implemented in C to overcome low computation times.

The requirements of Big Data have prompted the development of a set of tools for data processing and storage such as Hadoop [18], Spark, NoSQL databases, or Hierarchical Data Format (HDF) [19]. Spark is gaining popularity in Python with PySpark module, and there are other libraries like Dask, that implement routines for parallelization using common modules like numpy, pandas or scikit-learn, suiting them to apply Machine Learning in Big Data. Still, they do not fully implement all functions from these modules, or are limited to adapt to complex algorithms which require extra modules. MPI is still a common communication protocol used for parallelization, and Open MPI is one of the most commonly used implementations of MPI, since it is open source and constantly updated by an active community [20]. MPI4py [21] is the most used module that allows the application of MPI parallelization on Python syntax, allowing users to avoid adapting their pipelines to C++ language, supported by MPI, decreasing code production time. MPI4py allows the communication of *pickable* elements, like numpy arrays, in a faster manner than the common communication method.

Nonetheless, there is a limitation in MPI which impedes parallelization of data chunks with more than $2^{31} = 2147483648$ elements [22] [23] [24]. The MPI standard uses C `int` for element count, which has a maximum value of 2^{31} for positive values and, thus, any object bigger than that cannot be communicated by MPI. This limit is indeed an upper bound since many Python objects, e.g. numpy arrays or pandas dataframes, have a complex structure in which one element of the object corresponds to several C-type elements, decreasing that upper bound to around 10^6 to $2 \cdot 10^7$ elements. Communication of bigger objects to the cores throws an `OverflowError`. The object size problem is also encountered in the case of an object small enough for distribution to the cores that after computation it transforms into an object too large to be recovered to the original core, throwing an `OverflowError`. Thus, the size limitation problem does not only restrict the use of MPI but also negatively affects its "robustness" since there are algorithms whose final object has a undetermined object size (UOS), hampering to know in advance whether the size limitation will be fulfilled. This case leads to a long delay in code execution, since `OverflowError` occurs only after distributing the object and waiting its processing.

Due the importance of facilitating the development of Python solutions for Big Data applications, several solutions to the size limit problem were proposed. Hammond et al. [24] [25] developed BigMPI, that circumvents the C `int` problem using derived C types allowing up to $\sim 9 \cdot 10^9$ GB of data for parallelization. However, BigMPI developers acknowledge the limitation of BigMPI of supporting more complex datatypes, which must be derived to built-in types, posing a problem for users whose pipelines include Python-like objects, which would need to be transformed into C-type objects for parallelization. Another solution is to divide the parallelizable object into *chunks*, and parallelize each *chunk* independently. The main drawback of this method, implementable in Python, is that it has to be tailored to suit the object type, i.e., a pandas dataframe and a numpy array have to be divided using different syntax. Moreover, the number of chunks that have to be produced to parallelize without error is not straightforward to calculate.

We developed BigMPI4py to overcome these problems. BigMPI4py is a Python implementation which uses MPI4py as the base parallelization module, and applies the *chunking* strategy to automatically distribute large objects across processors. BigMPI4py distributes the most common Python data types (pandas dataframes, numpy arrays, simple lists, nested lists, or lists composed of previous elements), regardless of the size, and keeping an easy syntax. BigMPI4py currently allows collective communication methods `bcast()`, `scatter()`, `gather()`, or `allgather()`, as well as the point-to-point communication method `sendrecv()`, adapted from the homonym functions on MPI4py. Furthermore, BigMPI4py automatically communicates numpy arrays with vectorized `Scatterv()` and `Gatherv()` methods, providing a substantial time optimization. Thus, BigMPI4py seamlessly implements the most common parallelization pipelines with an easy and understandable syntax.

The remainder of this paper is organized as follows. Section 2 provides a system overview of the main functions implemented in the BigMPI4py module. Section 3 describes the two strategies of BigMPI4py to split the objects and their use by `scatter()` and `gather()` functions. Section 4 explains how other functions from this module were implemented. Section 5 explains the vectorization details of BigMPI4py. Section 6 presents numerical results on how BigMPI4py overcomes the object size limit of MPI4py, and performs faster in several computationally demanding applications. Section 7 includes several pieces of code illustrating the simplicity of the design of the parallelization task. Finally, Section 8 has the concluding remarks.

2 System Overview

BigMPI4py has 5 functions which mimic the actions of its MPI and MPI4py counterparts: `bcast()`, `scatter()`, `gather()` and `allgather()` belong to the collective communication category, and `sendrecv()` belongs to the point-to-point communication category.

- `bcast()` communicates replicates of an object from the source (root) core to the remaining cores.
- `scatter()` divides the object from the source core into n *chunks* (n = number of cores) and distributes them to the remaining cores.
- `gather()` combines the objects from all the cores into a unified object and sends it to a destination core.
- `allgather()` combines the objects from all the cores and distributes copies of the combined object to all the cores.
- `sendrecv()` communicates an object to a specific destination core.

Due to similarities in the algorithmic structure of these 5 functions, some of them are combined into 2 unified functions: `_general_scatter()` and `_general_gather()`.

`_general_scatter()` is called by `bcast()`, `scatter()` and `sendrecv()`, whereas `_general_gather()` is called by `gather()` and `allgather()`. `_general_scatter()` has 3 main arguments:

- `object type`: pandas dataframe, series, numpy arrays or lists. Lists are divided into "simple" lists when they contain integers, floats or strings; and "complex" lists when they contain dataframe, series, arrays or other lists. "Mixed" lists with "complex" and "simple" type of elements simultaneously are not currently supported.
- `optimize`: if `True` and when the object is a numeric numpy array it can be scattered using the `comm.Scatterv()` function from MPI4py. This function uses a vectorized implementation of the parallelization, drastically improving parallelization efficiency.
- `by`: columns referring to one or several categorical variables. E.g., if a table contains one column with months and another one with week days, choosing `by` with these two columns selects all combinations of months and week days and distributes tables so that no combination of both columns is distributed.

`_general_gather()` takes the same arguments as `_general_scatter()` although `by` is not considered. In both functions, the main structure has the following steps:

1) Determine the object type. Different partitioning strategies follow depending on the type.
2) Determine the positions to divide the object into n parts, or *chunks*, n being the number of processors.
3) Determine, for each *chunk* of the object, secondary parameters to further divide this object in case of memory limitations.
4) Perform the division of the object.
5) Merge all the communicated objects into a final object.

During the second step (Determine the division positions), a list of indexes by which the object is divided is created, where A, the input object, is split into n *chunks*. A will be divided into equally-sized *chunks* unless `by` argument is set; in that case the number of combinations will be equally distributed.

If A has length $|A|$ (number of rows in arrays, dataframes or series, and number of elements in lists), then the index positions of the division are

$$p_{n_i} = \left\lfloor \frac{|A| \cdot i}{n} \right\rfloor \qquad n_i \in \{1, \cdots, n\}$$

A_{n_i} is defined as the n_i-th *chunk* of A with $n_i \in \{1, \cdots, n\}$. Thus, the set of indexes would be $\{0, p_1, p_2, \cdots, p_{n-1}, p_n = |A|\}$, and the n_i-th *chunk* would start at position p_{n_i-1} and end at position p_{n_i}. If `by` argument is set to `True`, then p_{n_i-1} and p_{n_i} would be limited indexes between two categories.

During the third step (Determine secondary parameters), instead of dividing the object into n parts, or *chunks*, each of the *chunks* is further split into k *subchunks*, so that the object size restriction is overcome during the communication of A to the processors. To calculate k, two strategies are developed, hereby termed Strategy I and Strategy II. Those strategies are slightly different depending on whether an scattering or a gathering is being performed, since the final object will be distributed to all the cores, or will be communicated to a specific one.

3 Strategies to calculate the number of *subchunks*

3.1 Strategy I

Strategy I deals with "simple" and "complex" lists, arrays, dataframes and series. After A has been divided into $A_1, A_2, \cdots, A_n$, if the size (memory allocation) of any of those *chunks*, A_i^*, is greater than $L_n = L/n$, L being the memory limit (which can be assigned by the user), then k_I is defined as

$$k_I = \max\left\{ \left\lceil \frac{A^*_{n_i}}{L_n} \right\rceil \right\}$$

BigMPIpy defines A_{n_i,k_i} as the k_i-th *subchunk* of the n_i-th *chunk*. By choosing this k_I value, it is assured that any combination (n_i, k_i) will result in an A_{n_i,k_i} smaller than L_n. Therefore, A could be expressed as the following matrix of subchunks:

$$A = \begin{pmatrix} A_{1,1} & \cdots & A_{1,k_I} \\ \vdots & \ddots & \vdots \\ A_{n,1} & \cdots & A_{n,k_I} \end{pmatrix}$$

The positions by which the objects are divided are:

$$p_{n_i,k_i} = p_{n_i-1} \left\lfloor \frac{k_i \cdot (p_{n_i} - p_{n_i-1})}{k_I} \right\rfloor \qquad k_i \in \{1, \cdots, k_I\}$$

For each value k_i, all A_{n_i,k_i} are combined into a list, and communicated by MPI4py functions. The model of organization varies depending on whether the object is scattered or gathered, although the main strategy remains the same. The description of the scattering algorithm is described in Algorithm 1, where SCATTER() refers to `comm.scatter()` function from MPI4py whereas MERGE() is a function developed in the module that inputs a list with objects of the same type (e.g., a list of dataframes), and returns a concatenated object. idx_A refers to the list of indexes that divides A into n parts. During the scattering k_I is selected as the maximum k value of all *chunks*. Using this k_I all p_{n_i,k_i} positions are calculated. Then, for each k_i value, the list with A_{n_i,k_i} *subchunks* is created and scattered. Each core n_i will have its *subchunk* A_{n_i,k_i}. In the end, each core will have a list with subchunks `scatter_object=[` $A_{n_i,1}, \cdots, A_{n_i,k_I}$ `]`, which will be merged into the original *chunk* A_{n_i}. The gathering strategy is similar, although its initial step differs, since the object is now distributed across cores. In this procedure, for each core, its k_I value is calculated, and the final k_I is the highest across cores, which is then communicated. Afterwards, each A_{n_i} is divided into k_I subchunks $(A_{n_i,1}, \cdots, A_{n_i,k_I})$. For each $k_i \in \{1, \cdots, k_I\}$, A_{n_i,k_i} is gathered, and the destination core receives the list `gather_ki=` $[A_{1,k_i}, \cdots, A_{n,k_i}]$, which occupies the $k_i + n_i \cdot n$ positions of a return list `gather_list`, n_i being the position of each element in `gather_ki`. After the gathering loop, `gather_list=` $[A_{1,1}, \cdots, A_{1,k_I}, \cdots, A_{n,k_I}]$ is merged onto the final object A.

Algorithm 1 Strategy I for scattering

```
procedure STRATEGY_I_SCATTER(A, idx_A, L, n)
    if rank == root then
        k ← 0
        idx_k ← []
        scatter_A ← []
        for i in 0:len(idx_A) do
            k ← max(k, int(size(A[idx_A[i:i+1]]) / L) + 1)
        end for
        for i in 0:len(idx_A) - 1 do
            APPEND(idx_k, int(z) for z in np.linspace(idx_A[i], idx_A[i + 1], k + 1)[:-1])
        end for
        APPEND(idx_k, len(A))
        for i in 0:len(idx_k) - 1 do
            APPEND(scatter_A, A[idx_k[i]:idx_k[i + 1]])
        end for
    end if
    k ← SCATTER(k)
    scatter_object ← [None] * k
    for ki in 0:k do
        scatter_object[ki] ← SCATTER([scatter_A[i] for i in range(0, k * n, k)])
        for i in 0:k * n:k do        ▷ Deleting to free memory
            del scatter_A[i]
        end for
    end for
    return MERGE(scatter_object)
end procedure
```

Algorithm 2 Strategy I for gathering

```
procedure STRATEGY_I_GATHER(A, L, n)
    k ← 0
    divide_A ← []
    for i in 0:len(idx_A) do
        k ← int(size(A) / L) + 1
    end for
    k ← max(ALLGATHER(k))
    idx_k ← [int(z) for z in np.linspace(0, len(A), k + 1)]
    for i in 0:len(idx_k) - 1 do
        APPEND(divide_A, A[idx_k[i]:idx_k[i + 1]])
    end for
    if rank == root then
        gather_list ← [None] * (n * k)
    end if
    for ki in 0:k do
        gather_ki ← GATHER(divide_A[ki])
        for ni in 0:n do
            gather_list[ki + ni * k] = gather_ki[i]
        end for
    end for
    return MERGE(gather_list)
end procedure
```

3.1.1 Example of array scattering with Strategy I

A is distributed into $n = 5$ cores as illustrated in Fig. 1 where the object A is represented with a pentagon. Since $n = 5$ A is divided into 5 *chunks*: A_1, A_2, A_3, A_4, and A_5, represented with triangles. If $L = 3$, then $L_n = 3/5 = 0.6$. if the weights of the *chunks* are 1, 1.5, 1.2, 1.1 and 1 respectively, $k = \lceil 1.5/0.6 \rceil = 3$. Then, the distribution of subchunks is $A_{1,1}, A_{1,2}, A_{1,3}, A_{2,1}, \cdots ..., A_{5,2}, A_{5,3}$, which are combined in the list `scatter_A`. During the first iteration on k_I, all $A_{i,1}$ are combined, distributed and attached to `scatter_object`, i.e., the third core receives $A_{1,3}$. After all iterations, each core has the list `scatter_object` $= [A_{n_i,1}, A_{n_i,2}, A_{n_i,3}]$, which, after merging, will be transformed to A_{n_i}.

3.1.2 Example of list scattering with Strategy I

$A = [a_1, a_2, a_3, \cdots, a_9]$ is distributed into $n = 5$ cores, thus it is divided into 5 *chunks*: $A_1 = [a_1, a_2]$, $A_2 = [a_3, a_4]$, $\cdots$, $A_5 = [a_9]$. If $k_I = 2$, then the distribution of *subchunks* is $[[a_1], [a_2], [a_3], [a_4], \cdots, [a_9], []]$, i.e., $A_{1,1} = a_1, A_{1,2} = a_2, A_{1,3} = a_3, \cdots, A_{5,1} = a_9, A_{5,2} = []$. After scattering, each core receives the list $[A_{n_i,1}, A_{n_i,2}]$; for core 3 this list is $[[a_5], [a_6]]$, and for core 5 it is $[[a_9], []]$. After merging, core 3 has $[a_5, a_6]$ and core 5 has $[a_9]$.

3.1.3 Example of array gathering with Strategy I

Let's suppose that $n = 3$ and there are three arrays to be gathered: A_1, A_2, A_3; as illustrated in Fig. 2, where each array is represented with a rhomboid and the final gathered array with a triangle. If $L_n = 2$ and $A_1^* = 3$, $A_2^* = 3.5$, $A_3^* = 3.7$, then $k = \lceil \max\{3/2, 3.5/2, 4/2\} \rceil = 2$. Therefore, A_{n_i} is split into $A_{n_i,1}$ and $A_{n_i,2}$. In the receiver core `gather_object` is set to [`None`, `None`, `None`, `None`, `None`, `None`]; and after the first iteration on k_I, the receiver core receives `gather_ki`$= [A_{1,1}, A_{2,1}, A_{3,1}]$, and `gather_object` is [$A_{1,1}$, `None`, $A_{2,1}$, `None`, $A_{3,1}$, `None`]. After the second iteration, `gather_object` is [$A_{1,1}$, $A_{1,2}$, $A_{2,1}$, $A_{2,2}$, $A_{3,1}$, $A_{3,2}$], which is merged onto the object A represented with a triangle in Fig. 2. Gathering a list with Strategy I involves similar steps as gathering an array.

3.2 Strategy II

Strategy II deals with "complex" lists in which one or more elements individually exceeds the memory limit. In this case, the k_I value from strategy I is not suitable since the best partition of elements would be the one that makes each element of the list to be scattered or gathered individually. E.g., for $A = [a_1, a_2, \cdots, a_4, a_5]$, with $a_2^* = 4$ and $L_n = 2$, if $n = 2$, then the k value for this case would be $k_I = 3$, making each element to be scattered individually. However, since a_2^* alone is greater than L_n, the scattering would be impossible.

Given a list A with f elements $A = [a_1, \cdots, a_f]$, the main objective of Strategy II is to split each element of A into k_{II} sub-elements a_{i,k_i}, so that $\sum_i^f a_{i,k_i} < L \ \forall \ k_i \in \{1, \cdots, k_{II}\} \iff a_{i,k_i} < L_n \ \forall \ i, k_i$.

Firstly, k_{II} is obtained by sampling k_{II} for each element in the list, and then returning the highest integer round up value. Once k_{II} is fixed, each element a_i in A is divided into k_{II} sub-elements $a_{i,1}, a_{i,2}, \cdots, a_{i,k_{II}}$. Then, for each $k_i \in$

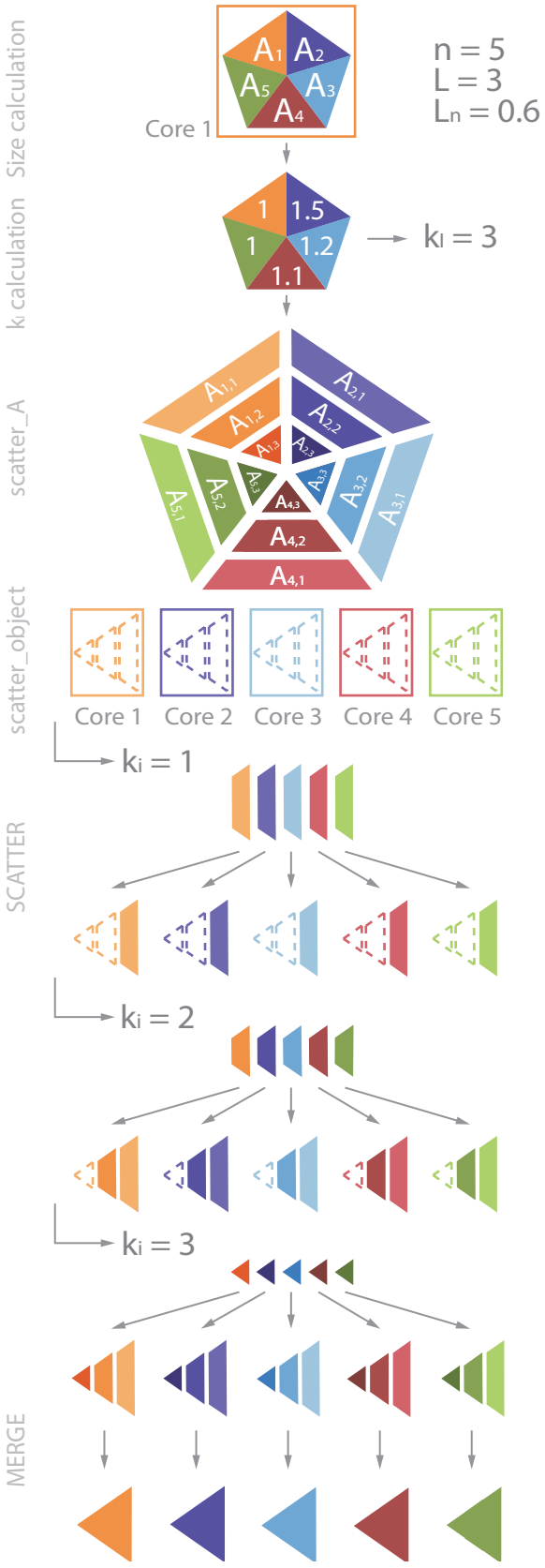

Fig. 1. Array scattering of example 3.1.1. n represents the number of cores, L is the memory limit (in GB) and $L_n = L/n$. For each colored piece, hue is the core that will process the *chunk* or *subchunk*, and luminosity represents the k_i loop in which the *subchunk* will be processed. Joined pieces belong to the same object, whereas separated pieces represent a list with *subchunks*. Dashed lines represent Nonetype objects within a list.

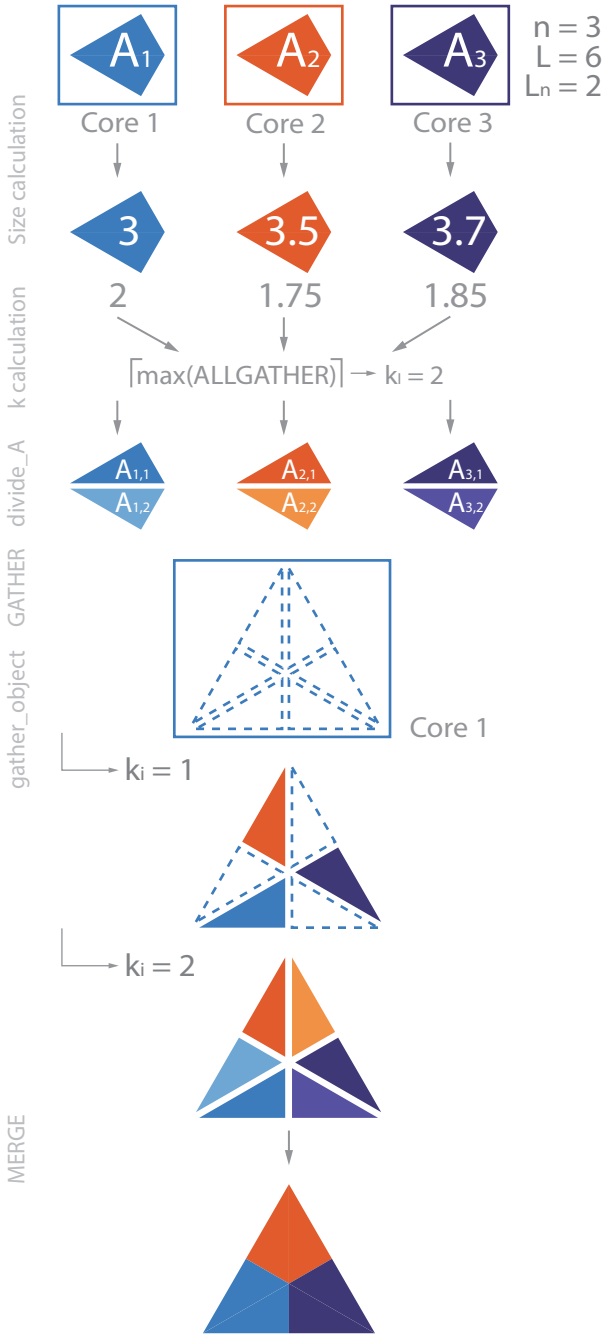

Fig. 2. Array gathering of example 3.1.3. n is the number of cores, L is the memory limit (in GB) and $L_n = L/n$. For each colored piece, hue represents the core that will process the *chunk* or *subchunk*, and luminosity represents the k_i loop in which the *subchunk* will be processed. Joined pieces belong to the same object, whereas separated pieces represent a list with *subchunks*. Dashed lines represent Nonetype objects within a list.

$\{1, \cdots, k_{II}\}$ the list with all a_{i,k_i} is scattered, and each core receives a respective set of sub-elements. At the end of the scattering loop, all a_{i,k_i} sub-elements are merged into their respective a_i elements.

The gathering strategy is similar, considering that k_{II} is the greatest value across cores. Since the number of elements for the list to be processed by each core may differ, all lists are filled with empty elements to match lengths. With this lengthening it is possible to iterate through all the elements of the list at the same time across cores.

After A elongation and k_{II} calculation, `gather_list` list with $n \cdot \text{len}(A)$ `None` elements is created to be filled with all a_i elements. For each $k_i \in \{1, \cdots, k_{II}\}$ a list `return_list` with all gathered a_{i,k_i} subelements is created, and the $k_i + n_i \cdot n$ elements are filled with `return_list`. After all n_i iterations, `gather_list` is complete. After gathering, `gather_list` might contain some `None` values, which are removed from the list.

Algorithm 3 Strategy II for scattering

```
1: procedure STRATEGY_II_SCATTER(A, idx_A, L, n)
2:     k ← 0
3:     for i in A do
4:         k ← max(k, int(size(i) / L) + 1)
5:     end for
       scatter_list ← [[[] for ki in k] for ni in n]
6:     for ni in n do
7:         A_ni ← A[idx_A[ni]:idx_A[ni+1]
8:         A_ni_k ← []
9:         for obj in A_ni do
10:            idx_k ← [int(i) for i in LINSPACE(0, len(obj), k+1)]
11:            obj_k ← [obj[idx_k[i]:idx_k[i+1] for i in 0:k]
12:            APPEND(A_ni_k, obj_k)
13:        end for
14:        scatter_list[ni] ← A_ni_k
15:    end for
       merge ← []
16:    for ki in 0:k do
17:        if rank == root then
18:            merge_ki ← []
19:            for sc_list_ni in sc_list do
20:                x ← []
21:                for l in 0:len(sc_list_ni) do
22:                    APPEND(x, sc_list_ni[l][ki])
23:                end for
24:                APPEND(merge_ki, x)
25:            end for
26:        end if
27:        merge_ki ← SCATTER(merge_ki)
28:        APPEND(merge, merge_ki)
29:    end for
30:    return_list ← []
31:    for l in 0:len(merge[0]) do
32:        return_ki ← [merge[ki][l] for ki in 0:k]
33:        APPEND(return_list, MERGE(return_ki))
34:    end for
35:    return return_list
36: end procedure
```

Algorithm 4 Strategy II for gathering

```
1: procedure STRATEGY_II_GATHER(A, L, n)
2:     k ← 0
3:     for i in A do
4:         k ← max(k, int(size(i) / L) + 1)
5:     end for
6:     k ← max(ALLGATHER(k))
7:     max_len_A ← len(A)
8:     max_len_A ← max(ALLGATHER(max_len_A))
9:     A ← A + [[] * (max_len_A - len(A))]
10:    gather_list ← [None] * (n * max_len_A)
11:    for i in 0:len(max_size_list) do
12:        cut_idx ← [int(x) for x in LINSPACE(0, len(i), k+1)]
13:        return_i ← []
14:        for ki in range(k) do
15:            x ← A[i][cut_idx[ki]: cut_idx[ki+1]
16:            gather_obj ← GATHER(x)
17:            APPEND(return_i, gather_obj)
18:        end for
19:        if return_i[0] then
20:            for ni in 0:n do
21:                list_return_i ← [return_i[ki][ni] for ki in range(k)]
22:                merged_i ← MERGE(list_return_i)
23:                if len(merged_i) > 0 then
24:                    gather_list[i + ni*max_len_A] ← merged_i
25:                end if
26:            end for
27:        end if
28:    end for
29:    return gather_list
30: end procedure
```

3.2.1 Example of scattering with Strategy II

$A = [a_1, a_2, a_3, a_4, a_5]$ is distributed into $n = 2$ cores, and $k_{II} = 3$. The list of distribution, `A_ni`, is $[[a_1, a_2, a_3], [a_4, a_5]]$. Then, each a_i element is split into $[a_{i,1}, a_{i,2}, a_{i,3}]$, and the previous list becomes `scatter_list` $= [[[a_{1,1}, a_{1,2}, a_{1,3}], [a_{2,1}, a_{2,2}, a_{2,3}], [a_{3,1}, a_{3,2}, a_{3,3}]], [[a_{4,1}, a_{4,2}, a_{4,3}], [a_{5,1}, a_{5,2}, a_{5,3}]]]$. Now, for each k_i (in this example, $k_i = 1$) the list `merge_ki` is created: $[[a_{1,1}, a_{2,1}, a_{3,1}], [a_{4,1}, a_{5,1}]]$, and after the scattering the first core receives $[a_{1,1}, a_{2,1}, a_{3,1}]$ while the second core receives $[a_{4,1}, a_{5,1}]$. This list is appended to the list `merge`. After all k_i values, `merge` is $[[a_{1,1}, a_{2,1}, a_{3,1}], \cdots, [a_{1,3}, a_{2,3}, a_{3,3}]]$ for core 1 and $[[a_{4,1}, a_{5,1}], \cdots, [a_{4,3}, a_{5,3}]]$ for core 2. Then, $[a_{n_i,1}, a_{n_i,2}, a_{n_i,3}]$ is merged to a_{n_i} for $n_i = 1, \cdots, 5$, and `return_list` is $[a_1, a_2, a_3]$ for core 1 and $[a_4, a_5]$ for core 2.

3.2.2 Example of gathering with Strategy II

The first core has the list $A_1 = [a_1, a_2, a_3]$, and the second core has the list $A_2 = [a_4, a_5]$. We obtain a k_{II} value of 2 for the second core, and of 4 for the first core, thus $k_{II} = 4$. A_2 is filled to $[a_4, a_5, []]$, and `gather_list` is [`None`, `None`, `None`, `None`, `None`, `None`]. BigMPI4py iterates

for `obj` in `0:3` and, for each iteration, gathers the i-th value of A_1 and A_2. The first iteration takes a_1 and a_4 respectively. Next, iterating for k_i, `x` = $a_{1,1}$ and $a_{4,1}$ for each core, and `gather_obj` = $[a_{1,1}, a_{4,1}]$. After all k_i iterations, `return_i` is $[[a_{1,1}, a_{4,1}], [a_{1,2}, a_{4,2}], \cdots, [a_{1,4}, a_{4,4}]]$. Then, iterating through n, $[a_{1,1}, \cdots, a_{1,4}]$ and $[a_{4,1}, \cdots, a_{4,4}]$ are merged to a_1 and a_4, and `gather_list` is filled to be $[a_1, \texttt{None}, \texttt{None}, a_4, \texttt{None}, \texttt{None}]$. After all iterations, `gather_list` is $[a_1, a_2, a_3, a_4, a_5, \texttt{None}]$, the `None` element is removed, and the list is returned.

4 SENDRECV(), ALLGATHER() AND BCAST() FUNCTIONS

`sendrecv()` function was implemented for point-to-point communication, whereas `allgather()` and `bcast()` were implemented for collective communication of whole objects across cores. `sendrecv()` was implemented together with `scatter()`, in `_general_scatter()` functions, since Strategy I and Strategy II are shared by the two functions. The main differences in the use of `scatter()` and `sendrecv()` are:

1) `sendrecv()` sends the object to a single core, thus $n = 1$ is set.
2) Since no scattering is performed, `by` argument is empty in `sendrecv()`.
3) `scatter()` calls `comm.scatter()` for scattering, whereas `sendrecv()` calls `comm.send(object, dest)` and `comm.recv(root)` for communication of the object.
4) Merging of objects with Strategy II in `sendrecv()` is simplified since $n = 1$.

`allgather()` was integrated together with `gather()` in `_general_gather()`, since Strategy I and Strategy II are also shared. The main differences in the use of `allgather()` and `gather()` are:

1) Instead of calling `comm.gather()`, `comm.allgather()` is called.
2) Combination of *subchunks* is performed for all cores instead of the destination core only.

`bcast()` function was implemented in a separate function, with the strategy developed in Algorithm 5. The first step in `bcast()` is to obtain the available memory of the computer (`mem`) and the memory allocation of the object (`size_A`). If $n \cdot$`size_A` $>$ `mem` a `MemoryError` is thrown before broadcasting is performed. Then, for each n_i in `1:n`, `bcast_list` is created, where the n_i^{th} element is A, and the rest are `None`.

5 VECTORIZED IMPLEMENTATION OF GATHER() AND SCATTER()

MPI4py includes vectorized functions `Scatterv()` and `Gatherv()` for numeric numpy arrays, in which the scattering and gathering are improved since they communicate the buffer of the *chunks*, instead of the values of the *chunks*. The original vectorized methods were implemented within `MPI_Gatherv()` and `MPI_Scatterv()` routines. BigMPI4py implements these functions within

Algorithm 5 bcast() algorithm

```
1:  procedure BCAST(A, L, n)
2:      mem ← GET_MEMORY()
3:      size_A ← GET_SIZE(A)
4:      for n_i in 0:n do
5:          bcast_list ← [None] * n
6:          bcast_list[n_i] ← A
7:          bcast_obj ← SCATTER(bcast_list, L)
8:          if rank == n_i then
9:              return_obj ← bcast_obj
10:         end if
11:     end for
12:     return return_obj
13: end procedure
```

`_general_scatter()` and `_general_gather()`. Thus, when called, BigMPI4py automatically checks whether the object is a numeric array and scatters or gathers the object using the corresponding function. During the vectorized implementation the array splitting positions, the number of rows, and the number of cells in each *chunk* are calculated, and these values are used with `Scatterv()` and `Gatherv()` methods from MPI4py, based on the original implementation from MPI using `sendbuf`, `sendcount` and `displs`. BigMPI4py adapts the vectorization also for large arrays, supporting values of k_I greater than 1. For this case, when using `scatterv()`, the `scatter_A` list for each k_i is merged into an array, and the counts for each array are considered. Although this step might be computationally expensive for big arrays, since `scatter_A` list must be transformed into an array, time performance is still improved, as shown in Results section.

6 RESULTS

To test the time performance of `scatter()` and `gather()` functions, we performed several simulations. For each function, random numpy arrays or pandas dataframes with 10 columns and row numbers ranging from 2^2 to 2^{26} (`gather()`) or 2^{30} (`scatter()`) at step 2 were generated. Additionally, different types (`int`, `float` and `string`) were used during random array generation. Then, scattering and gathering computational times were tested with $n = 10$, and with 5 repetitions for each state. The tests were performed on a Lenovo ThinkStation P910, Intel Xeon @ 2.3 GHz (28 cores) with 512 GB RAM. `cProfile` and `line_profiler` Python modules were used for monitoring time profiling. The simulations compare MPI4py, and BigMPI4py vectorized and non-vectorized implementations. The results of the times required for scattering and gathering are shown in Figs. 3 and 4, respectively.

Both vectorized and non-vectorized approaches process larger tables than the MPI4py approach, i.e., when scattering, `int` and `float` tables MPI4py can only process up to nearly $2^{24} * 10 \approx 167$ million cells, and `string` tables up to $2^{20} * 10 \approx 10$ million cells; whereas BigMPI4py reaches table sizes up to the maximum available memory in some situations. The objects that MPI4py cannot process due to its size limitation have sizes that are common for objects required to be processed in multiple Big Data projects.

Scattering and gathering processing times show a common trend across all tables and types: there is a quasilinear behavior until $2^{13} \cdot 10 \approx 80000$ to $2^{15} \cdot 10 \approx 300000$ cells, in which the processing times are around 10^{-3} for numpy arrays, and 10^{-2} for pandas dataframes. For larger numbers of cells, however, the times increase exponentially (linearly after log-log transformation), following a trend with almost no deviation from the expected exponential regression. For short processing times, MPI4py shows faster computation performance, up to an order of magnitude. This difference is due to the preprocessing done by BigMPI4py, where the main contributions are the communication of intermediate objects with MPI4py, like k_I values, consuming up to 60% of the total time. For longer processing times when the behavior is exponential, most of the time ($> 95\%$) is dedicated to the communication of the main object across cores, or the *merging* of a list of tables into a single table. For vectorized processing *merging* can take up to 70% of the processing time, whereas with non-vectorized processing *merging* takes up to 35% of the processing time. Interestingly, vectorized parallelization is still nearly 10 times faster, even considering that long time dedicated to *merging*. For string arrays the processing times are longer for the vectorized processing since a conversion between `string` and `float` must be performed for vectorized parallelization using `Scatterv()`, which consumes up to 80% of the total processing time. By default, string arrays are not processed by vectorized parallelization, which ought to be used for numeric arrays.

Vectorized parallelization is only available for arrays, and not for pandas dataframes, due to the large amount of time that requires to transform a numpy array to a pandas dataframe in case of big object sizes. Thus, processing times for vectorized and non-vectorized parallelization in pandas arrays are technically the same. Differences between vectorized and non-vectorized parallelization are shown in Figs. 5 and 6. Numpy float arrays show a great parallelization advantage (up to five times for 2^{18} rows) by vectorization both in scattering and gathering. For numeric types, vectorized scattering shows at least a two-fold time decrease even for number of rows that are not supported by MPI4py due to size limitations. Regarding vectorized gathering, numpy `float` arrays show improved computation times across the entire size spectrum, although for `int` arrays the processing time increases with respect to the non-vectorized processing. Profiling shows that $>95\%$ of time is associated to barrier synchronization, which implies a mismatch in processing rhythms across cores. This time profiling is highly dependent on the number of cores, obtaining better times with fewer processors, and cannot be always reproduced.

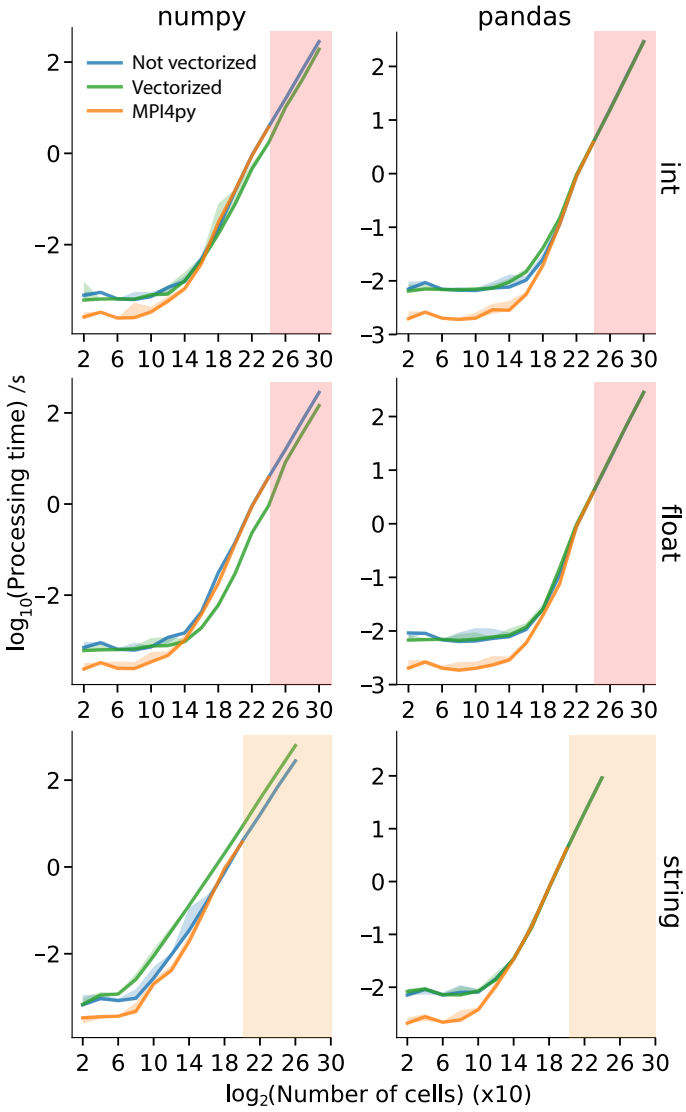

Fig. 3. Scattering times for numpy arrays and pandas dataframes for three data types. The number of cells that MPI4py is not able to process due to OverflowError is shadowed in red for int and float data types, and in yellow for string data type. Color filling between lines represents the 10^{th} and 90^{th} percentiles of the data.

7 Examples of use of BigMPI4py

Besides overcoming the object size limitation of MPI4py, BigMPI4py is designed to follow the philosophy of a simple syntax, only requiring the strictly necessary code to communicate with BigMPI4py. Any piece of code or attribute that is not essential for describing the parallelization is included inside BigMPI4py code, and is optional. BigMPI4py functions share some common attributes:

- `scatter_object` or similar: object to be communicated.
- `comm`: `MPI4py.MPI.COMM_WORLD` object.
- `size_limit`: limit of object size for k_I and k_{II}.
- `root`: in `gather()` function, the destination core where objects will be gathered; in the rest of functions, the core from which the object comes.
- `optimize`: apply *optimized* communication if possible.

Only the `scatter_object` and `comm` objects are required attributes, and the rest have default values. To use BigMPI4py the following code lines are required at the beginning of Algorithm 6.

Algorithm 7 shows array, dataframe and complex list scattering and broadcasting.

Firstly, all variables must be declared as `None` to do the communication (line 1). Then, for the root core, the object is created (lines 2-5). Finally, `scatter()` function is called

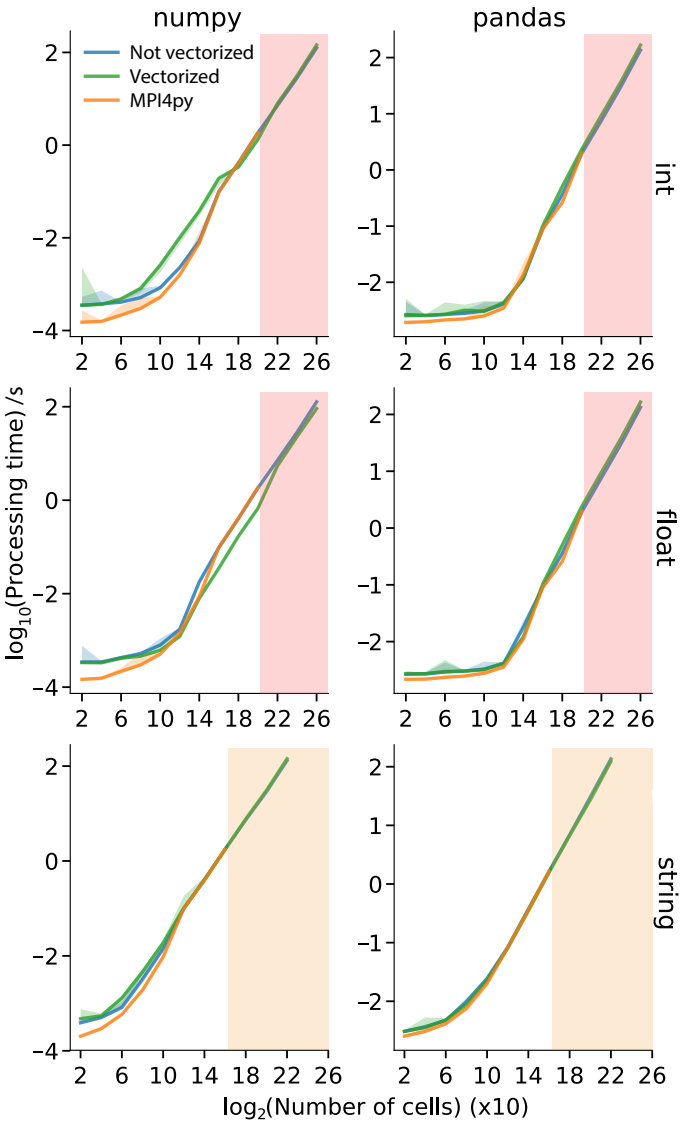

Fig. 4. Gathering times for numpy arrays and pandas dataframes in three data types. The number of cells that MPI4py is not able to process due to OverflowError is shadowed in red for int and float data types, and in yellow for string data type. Color filling between lines represents the 10th and 90th percentiles of the data.

Algorithm 6 Header of a Python file

```
from mpi4py import MPI
import BigMPI4py as BM
comm = MPI.COMM_WORLD
size, rank = comm.Get_size(), comm.Get_rank()
```

(lines 7-9). The same syntax is applied for object broadcasting (line 10). The code is simple, and with a working knowledge MPI4py, scattering is implemented in 4 lines of code.

If an object is distributed according to some categorical columns, `by` argument must be used. Algorithm 8 shows an example of this. In this example BigMPI4py extracts all pairs of values from the categorical columns (['A', 'Red'], ['B', 'Red'], ['B', 'Blue'], ['A', 'Blue'], ['C', 'Red'] and ['C', 'Blue']), and scatters the dataframe so that no dataframe with each pair of values is distributed across cores.

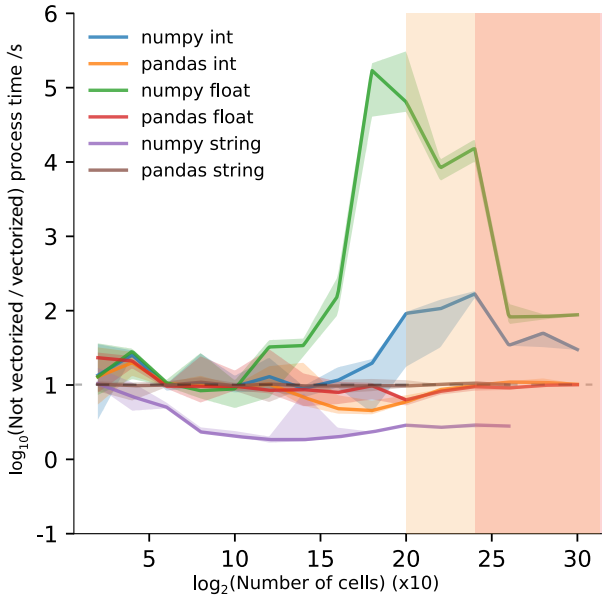

Fig. 5. Scattering time ratio between non-vectorized and vectorized approaches. The number of cells that MPI4py is not able to process due to OverflowError is shadowed in red for int and float data types, and in yellow for string data type. Color filling between lines represents the 10th and 90th percentiles of the data.

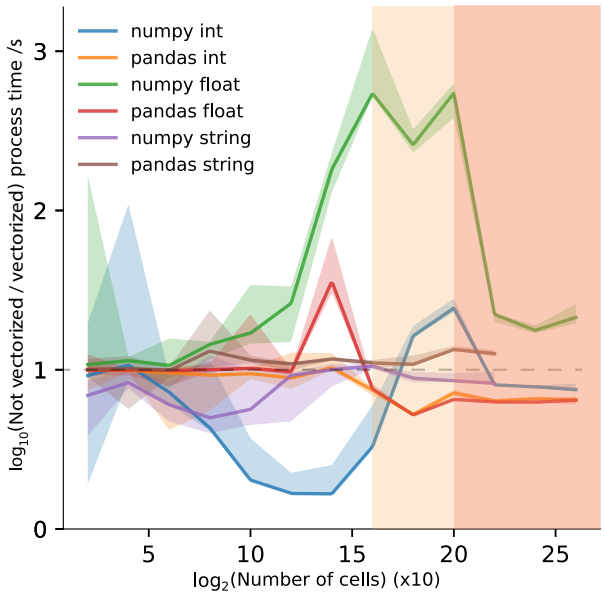

Fig. 6. Gathering time ratio between non-vectorized and vectorized approaches. The number of cells that MPI4py is not able to process due to OverflowError is shadowed in red for int and float data types, and in yellow for string data type. Color filling between lines represents to the 10th and 90th percentiles of the data.

Algorithm 7 Object scattering and broadcasting

```
arr, df, lst = None, None, None
if rank == 0:
arr = np.random.rand(2 ** 26, 10)
df = pd.DataFrame(arr)
lst = [arr, df, arr, df, arr, df]

scatter_arr = BM.scatter(arr, comm)
scatter_df = BM.scatter(df, comm)
scatter_lst = BM.scatter(lst, comm)

bcast_lst = BM.bcast(lst, comm)
```

Algorithm 8 Object scattering with categorical variables

```
arr = None
if rank == 0:
arr = pd.DataFrame(
[['A', 'Red', 0],
['A', 'Red', 1],
['B', 'Red', 0],
['B', 'Red', 0],
['B', 'Blue', 0],
['A', 'Blue', 1],
['C', 'Red', 0],
['C', 'Blue', 0]],
columns=['Letter', 'Color', 'Number'])

scatter_arr = BM.scatter(arr, comm,
by=['Letter', 'Color']])
```

The procedure to code object gathering and allgathering is similar to the procedure of scattering. Algorithm 9 shows the communication of an array for `gather` and `allgather`. The rest of object types follow the same syntax.

Algorithm 9 Object gathering and allgathering

```
arr = np.random.rand(2 ** 26, 10)
gather_arr = BM.gather(arr, comm)
allgather_arr = BM.allgather(arr, comm)
```

Gathering and allgathering of an object requires only two lines of code, making it even easier to program.

An example of array point-to-point communication is shown in Code 10.

Algorithm 10 Object point-to-point communication

```
arr = np.random.rand(2 ** 26, 10)
sendrecv_arr = BM.sendrecv(arr, comm,
   dest = 2)
```

The `dest` argument in line 2 is the processor that will receive the object.

Any code that uses MPI4py and BigMPI4py must be run externally using an MPI implementation, since it depends on MPI. Algorithm 11 shows an example using `mpirun` program from OpenMPI implementation.

More detailed examples of the simplicity of BigMPI4py are provided as a Jupyter notebook in the installation package.

Algorithm 11 Example of MPI launch

```
mpirun -np 4 python ~/mycode.py
```

8 Conclusion

BigMPI4py brings the possibility to take advantage of the parallelization implementing MPI in Python without any theoretical object size limitation, using all the computational power (number of cores and memory) of hardware (multicore PC, workstation or HPC) in Big Data projects with the only limitation being the resources in the system. The use of BigMPI4py increases the "robustness" of MPI4py, since it allows to overcome the `OverflowError` arising in MPI4py when the size of the output object exceeds the limit of MPI4py even when the size of the input object does not exceed such limit. Additionally, BigMPI4py simplifies the use of MPI4py by automatically splitting the object to be communicated across the cores and by automatically deciding whether to optimize the parallelization by performing vectorization of objects.

Acknowledgments

This work have been supported by grants DFG113/18 from Diputación Foral de Gipuzkoa, Spain, Ministry of Economy and Competitiveness, Spain, MINECO grant BFU2016-77987-P, Basque Government Predoctoral Grant PRE_2018_1_0008, Spain, and Instituto de Salud Carlos III (AC17/00012) co-funded by the European Union (Eracosysmed/H2020 Grant Agreement No. 643271).

The authors would like to thank Daniela Gerovska for fruitful discussion and comment during the preparation of this manuscript.

Conflict of Interest: none declared.

References

[1] S. Lohr, "The Origins of 'Big Data': An Etymological Detective Story," https://bits.blogs.nytimes.com/2013/02/01/the-origins-of-big-data-an-etymological-detective-story/.

[2] M. Hilbert and P. Lopez, "The World's Technological Capacity to Store, Communicate, and Compute Information," *Science*, vol. 332, no. 6025, pp. 60–65, 2011.

[3] B. Marr, "How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read," https://bit.ly/2FyrOrD, May 2018.

[4] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on Big Data," *Information Sciences*, vol. 275, pp. 314–347, August 2014.

[5] G.-H. Kim, S. Trimi, and J.-H. Chung, "Big-Data Applications in the Government Sector," *Communications of the ACM*, vol. 57, no. 3, pp. 78–85, March 2014.

[6] M. Gea, H. Bangui, and B. Buhnova, "Big Data for Internet of Things: A Survey," *Future Generation Computer Systems*, vol. 87, October 2018.

[7] P.-L. Luu, D. Gerovska, M. Arrospide-Elgarresta, S. Retegi-Carrion, H. R. Scholer, and M. J. Arauzo-Bravo, "P3BSseq: parallel processing pipeline software for automatic analysis of bisulfite sequencing data," *Bioinformatics*, vol. 33, no. 3, pp. 428–431, February 2017.

[8] A. M. Ascension, M. Arrospide-Elgarresta, A. Izeta, and M. J. Arauzo-Bravo, "NaviSE: superenhancer navigator integrating epigenomics signal algebra," *BMC Bioinformatics*, vol. 18, no. 296, June 2017.

[9] K. He, D. Ge, and M. He, "Big Data Analytics for Genomic Medicine," *International Journal of Molecular Sciences*, vol. 18, no. 2, p. 412, 2017.

[10] Y. Zhang and Y. Zhao, "Astronomy in the Big Data Era," *Data Science Journal*, vol. 14, no. 0, p. 11, 2015.
[11] A. D. Mauro, M. Greco, and M. Grimaldi, "A formal definition of Big Data based on its essential features," *Library Review*, vol. 65, no. 3, pp. 122–135, 2016.
[12] D. Laney, "3D Data Management: Controlling Data Volume, Velocity, and Variety," 2001.
[13] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, pp. 114–117, April 1965.
[14] T. Pearson and R. Wegener, ""NIST big data public working group", Draft of Big Data Definition," www.bain.com/images/bain_brief_big_data_the_organizational_challenge.pdf, 2013.
[15] V. Mayer-Schonberger and K. Cukier, "Big Data: A Revolution That Will Transform How We Live, Work and Think," in *Big Data: A Revolution That Will Transform How We Live, Work and Think*. John Murray, October 2013.
[16] S. Cass, "The 2018 Top Programming Languages," https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages, July 31.
[17] B. Frederickson, "Trending Programming Languages ranked by GitHub Users ," https://github.com/benfred/github-analysis, April 2018.
[18] A. Cheptsov, "HPC in Big Data Age: An Evaluation Report for Java-Based Data-Intensive Applications Implemented with Hadoop and OpenMPI," in *Proceedings of the 21st European MPI Users' Group Meeting*. New York, NY, USA: ACM, 2014, pp. 175:175–175:180.
[19] T. H. Group, "About us," https://www.hdfgroup.org/about-us/.
[20] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 97–104.
[21] L. Dalcín, R. Paz, and M. Storti, "Mpi for python," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 1108–1115, 2005.
[22] M. B. Nardelli and L. Dalcin, "OverflowError: integer 2768896564 does not fit in 'int'," *Bitbucket*.
[23] T. Lukinov and L. Dalcin, "OverflowError: integer 2559182040 does not fit in 'int'," https://groups.google.com/forum/#!topic/mpi4py/Ny-16HE3Aus.
[24] J. R. Hammond, A. Schäfer, and R. Latham, "To `int_max`... and beyond!: Exploring large-count support in mpi," in *Proceedings of the 2014 Workshop on Exascale MPI*, ser. ExaMPI '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 1–8.
[25] J. Hammond, "BigMPI," https://github.com/jeffhammond/BigMPI, 2018.

Alex M. Ascensión Graduated with honors of Biochemistry and Molecular Biology degree, University of the Basque Country, Spain (2017). MSc in Bioinformatics, Autonomous University of Barcelona, Spain (2018). He is currently completing a Degree in Mathematics as a part-time student at the National University of Distance Education, Spain, to complement his biological and computational knowledge with a solid mathematics basis. Currently doing his PhD in Computational Biology with Basque Government grant. He is currently working at Biodonostia Health Research Institute, Spain, at the Computational Biology group (leadered by Marcos J. Araúzo-Bravo) alongside with the Tissue Engineering group (leadered by Ander Izeta). His research is focused on several topics, such as epigenomic signaling or single-cell studies. He is interested in the development and application of computational techniques and mathematical methods to his topics of research. He has more than 3 years of experience in the computational field, his main programming languages are Python and R, as well as Matlab, Perl or C++ to a lesser extent. He is a member of the scientific committee of the European Project at the Eracosysmed JTC-2 (H2020) call 4D-Healing: Data-Driven Drug Discovery for Wound Healing. His main functions within the committee are the development of the web page and the computational analysis of the generated single-cell RNA-seq data.

Marcos J. Araúzo-Bravo Graduated as an Electronic and Control Engineer, University of Valladolid, Spain (1996). In 2001 he earned a Ph.D. in industrial technologies from the University of Cartagena, developing neuro-fuzzy algorithms for monitoring penicillin production. From 1998 to 2004 he was an Associate Professor in electrical engineering at Burgos University. In 2000 he received a scholarship from the Japanese Ministry of Education to work in the field of Metabolic Engineering at Kyushu Institute of Technology, Japan. In 2002 he earned a Ph.D. in Information Technology and Biotechnology from the Kyushu Institute of Technology, Iizuka, Japan. From 2004 to 2006 he was a Japan-Society-for-the-Promotion-of-Science postdoctoral research fellow at the Kyushu Institute of Technology, where he worked on the synergetic control of genetic networks through transcriptional regulators. From 2006 to 2014 he led the laboratory of Computational Biology and Bioinformatics at the Max Planck Institute for Molecular Biomedicine in Mnster, Germany, developing tools for deciphering cellular reprogramming, methods to study transcription regulation, and algorithms for high-throughput data analysis. Since 2014 he is an Ikerbasque Research Professor, head of the group of Computational Biology and Systems Biomedicine and head of the Computational Biomedicine Data Analysis Platform at the Biodonostia Health Research Institute, San Sebastin, Spain. He develops Big Data approaches for integrating omics, image and clinical history data to study the interaction of biological networks in terms of their topology, dynamics, and perturbations to interpret complex biological systems associated with neurodegenerative diseases, cancer, aging, stem cells and regenerative medicine. His vast experience analyzing and deriving models from omics data produced more than 120 publications, some of them very high impact such as Science, Nature, Nature, Cell. He is the leader of the European Project at the Eracosysmed JTC-2 (H2020) call 4D-Healing: Data-Driven Drug Discovery for Wound Healing.