Parallel Statistics In Python Documentation

Joe Zuntz

Mar 24, 2023

CONTENTS:

This package collects tools which compute weighted statistics on parallel, incremental data, i.e. data being read by multiple processors, a chunk at a time, using MPI.

The ParallelMeanVariance tool will be much faster if numba is installed.

CHAPTER

ONE

API DOCUMENTATION

1.1 Parallel Mean Calculation

class parallel_statistics.**ParallelMean**(*size*, *sparse=False*)

`ParallelMean` is a parallel and incremental calculator for mean statistics. "Incremental" means that it does not need to read the entire data set at once, and requires only a single pass through the data.

The calculator is designed to work on data in a collection of different bins, for example a map (where the bins are pixels). The usual life-cycle of this class is:

- create an instance of the class (on each process if in parallel)
- repeatedly call `add_data` or `add_datum` on it to add new data points
- call `collect`, (supplying in MPI communicator if in parallel)

You can also call the `run` method with an iterator to combine these.

If only a few indices in the data are expected to be used, the sparse option can be set to change how data is represented and returned to a sparse form which will use less memory and be faster below a certain size.

Bins which have no objects in will be given weight=0 and mean=nan.

Methods

add_data(bin, values[, weights])	Add a chunk of data in the same bin to the sum.
add_datum(bin, value[, weight])	Add a single data point to the sum.
collect([comm, mode])	Finalize the sum and return the counts and the means.
run(iterator[, comm, mode])	Run the whole life cycle on an iterator returning data chunks.

add_data(*bin*, *values*, *weights=None*)

Add a chunk of data in the same bin to the sum.

Parameters

bin: int
Index of bin or pixel these value apply to

values: sequence
Values for this bin to accumulate

weights: sequence
Optional, weights per value

`add_datum`(*bin*, *value*, *weight=None*)

Add a single data point to the sum.

Parameters

bin: int
Index of bin or pixel these value apply to

value: float
Value for this bin to accumulate

`collect`(*comm=None*, *mode='gather'*)

Finalize the sum and return the counts and the means.

The `mode` decides whether all processes receive the results or just the root.

Parameters

comm: mpi communicator or None
If in parallel, supply this

mode: str, optional
"gather" or "allgather"

Returns

count: array or SparseArray
The number of values hitting each pixel

mean: array or SparseArray
The mean of values hitting each pixel

`run`(*iterator*, *comm=None*, *mode='gather'*)

Run the whole life cycle on an iterator returning data chunks.

This is equivalent to calling add_data repeatedly and then collect.

Parameters

iterator: iterator
Iterator yielding (pixel, values) pairs

comm: MPI comm or None
The comm, or None for serial

Returns

count: array or SparseArray
The number of values hitting each pixel

sum: array or SparseArray
The total of values hitting each pixel

1.2 Parallel Sum Calculation

class parallel_statistics.**ParallelSum**(*size*, *sparse=False*)

`ParallelMean` is a parallel and incremental calculator for sums. "Incremental" means that it does not need to read the entire data set at once, and requires only a single pass through the data.

The calculator is designed to work on data in a collection of different bins, for example a map (where the bins are pixels). The usual life-cycle of this class is:

- create an instance of the class (on each process if in parallel)
- repeatedly call `add_data` or `add_datum` on it to add new data points
- call `collect`, (supplying in MPI communicator if in parallel)

You can also call the `run` method with an iterator to combine these.

If only a few indices in the data are expected to be used, the sparse option can be set to change how data is represented and returned to a sparse form which will use less memory and be faster below a certain size.

Bins which have no objects in will be given weight=0 and sum=0.

Methods

add_data(bin, values[, weights])	Add a chunk of data in the same bin to the sum.
add_datum(bin, value[, weight])	Add a single data point to the sum.
collect([comm, mode])	Finalize the sum and return the counts and the sums.
run(iterator[, comm, mode])	Run the whole life cycle on an iterator returning data chunks.

add_data(*bin*, *values*, *weights=None*)

Add a chunk of data in the same bin to the sum.

Parameters

bin: int
Index of bin or pixel these value apply to

values: sequence
Values for this bin to accumulate

weights: sequence
Optional, weights per value

add_datum(*bin*, *value*, *weight=None*)

Add a single data point to the sum.

Parameters

bin: int
Index of bin or pixel these value apply to

value: float
Value for this bin to accumulate

collect(*comm=None*, *mode='gather'*)

Finalize the sum and return the counts and the sums.

The "mode" decides whether all processes receive the results or just the root.

Parameters

comm: mpi communicator or None
If in parallel, supply this

mode: str, optional
"gather" or "allgather"

Returns

count: array or SparseArray
The number of values hitting each pixel

sum: array or SparseArray
The total of values hitting each pixel

`run`(*iterator*, *comm=None*, *mode='gather'*)

Run the whole life cycle on an iterator returning data chunks.

This is equivalent to calling add_data repeatedly and then collect.

Parameters

iterator: iterator
Iterator yielding (pixel, values) pairs

comm: MPI comm or None
The comm, or None for serial

Returns

count: array or SparseArray
The number of values hitting each pixel

sum: array or SparseArray
The total of values hitting each pixel

1.3 Parallel Mean & Variance Calculation

`class` parallel_statistics.`ParallelMeanVariance`(*size*, *sparse=False*)

`ParallelMeanVariance` is a parallel and incremental calculator for mean and variance statistics. "Incremental" means that it does not need to read the entire data set at once, and requires only a single pass through the data.

The calculator is designed to work on data in a collection of different bins, for example a map (where the bins are pixels).

The usual life-cycle of this class is:

- create an instance of the class (on each process if in parallel)
- repeatedly call `add_data` or `add_datum` on it to add new data points
- call `collect`, (supplying in MPI communicator if in parallel)

You can also call the `run` method with an iterator to combine these.

If only a few indices in the data are expected to be used, the sparse option can be set to change how data is represented and returned to a sparse form which will use less memory and be faster below a certain size.

Bins which have no objects in will be given weight=0, mean=nan, and var=nan.

The algorithm here is basd on Schubert & Gertz 2018, Numerically Stable Parallel Computation of (Co-)Variance

By default the module looks for the package "Numba" and uses its just-in-time compilation to speed up this class. To disable this, export the environment variable PAR_STATS_NO_JIT=1

Attributes

size: int
number of pixels or bins

sparse: bool
whether are using sparse representations of arrays

Methods

add_data(bin, values[, weights])	Add a chunk of data in the same bin.
add_datum(bin, value[, weight])	Add a single data point to the sum.
collect([comm, mode])	Finalize the statistics calculation, collecting togther results from multiple processes.
run(iterator[, comm, mode])	Run the whole life cycle on an iterator returning data chunks.

`add_data`(*bin*, *values*, *weights=None*)

Add a chunk of data in the same bin.

Add a set of values assinged to a given bin or pixel. Weights may be supplied, and if they are not will be set to 1.

Parameters

bin: int
The bin or pixel for these values

values: sequence
A sequence (e.g. array or list) of values assigned to this bin

weights: sequence, optional
A sequence (e.g. array or list) of weights per value

`add_datum`(*bin*, *value*, *weight=1*)

Add a single data point to the sum.

Parameters

bin: int
Index of bin or pixel these value apply to

value: float
Value for this bin to accumulate

weight: float
Optional, default=1, a weight for this data point

`collect`(*comm=None*, *mode='gather'*)

Finalize the statistics calculation, collecting togther results from multiple processes.

If mode is set to "allgather" then every calling process will return the same data. Otherwise the non-root processes will return None for all the values.

You can only call this once, when you've finished calling add_data. After that internal data is deleted.

Parameters

comm: MPI Communicator, optional
mode: string, optional
'gather' (default), or 'allgather'

Returns

weight: array or SparseArray
The total weight or count in each bin

mean: array or SparseArray
An array of the computed mean for each bin

variance: array or SparseArray
An array of the computed variance for each bin

`run`(*iterator*, *comm=None*, *mode='gather'*)

Run the whole life cycle on an iterator returning data chunks.

This is equivalent to calling add_data repeatedly and then collect.

Parameters

iterator: iterator
Iterator yieding (bin, values) or (bin, values, weights)

comm: MPI comm, optional
The comm, or None for serial

mode: str, optional
"gather" or "allgather"

Returns

weight: array or SparseArray
The total weight or count in each bin

mean: array or SparseArray
An array of the computed mean for each bin

variance: array or SparseArray
An array of the computed variance for each bin

1.4 Parallel Histograms

`class` parallel_statistics.`ParallelHistogram`(*edges*)

ParallelHistogram is a parallel and incremental calculator histograms. "Incremental" means that it does not need to read the entire data set at once, and requires only a single pass through the data.

The usual life-cycle of this class is:

- create an instance of the class (on each process if in parallel)
- repeatedly call `add_data` or `add_datum` on it to add new data points
- call `collect`, (supplying in MPI communicator if in parallel)

You can also call the `run` method with an iterator to combine these.

Since histograms are usually relatively small, sparse arrays are not enabled for this class.

Bin edges must be pre-defined and values outside them will be ignored.

Methods

add_data(data[, weights])	Add a chunk of data to the histogram.
collect([comm])	Finalize and collect together histogram values
run(iterator[, comm])	Run the whole life cycle on an iterator returning data chunks.

add_data(*data*, *weights=None*)

Add a chunk of data to the histogram.

Parameters

data: sequence
Values to be histogrammed

weights: sequence, optional
Weights per value.

collect(*comm=None*)

Finalize and collect together histogram values

Parameters

comm: MPI comm or None
The comm, or None for serial

Returns

counts: array
Total counts/weights per bin

run(*iterator*, *comm=None*)

Run the whole life cycle on an iterator returning data chunks.

This is equivalent to calling add_data repeatedly and then collect.

Parameters

iterator: iterator
Iterator yieding values or (values, weights) pairs

comm: MPI comm or None
The comm, or None for serial

Returns

counts: array
Total counts/weights per bin

1.5 Sparse Arrays

class parallel_statistics.**SparseArray**(*size=None*, *dtype=<class 'numpy.float64'>*)

A sparse 1D array class.

This not complete, and is mainly designed to support the use case in this package. The scipy sparse classes are all focused on matrix applications and did not quite fit

These operations are defined:

- setting and getting indices
- Adding another by another `SparseArray`
- Subtracting to another by another `SparseArray`
- Multiplying by another `SparseArray` with the same indices
- Dividing by another `SparseArray` with the same indices
- Raising the array to a scalar power
- Comparing to another `SparseArray` with the same indices

Examples

```
>>> s = SparseArray()
>>> s[1000] = 1.0
>>> s[2000] = 2.0
>>> t = s + s
```

Attributes

d
[dict] The dictionary of set indices (keys) and values

Methods

count_nonzero()	The number of non-zero array elements
from_dense(dense)	Convert a standard (dense) 1D array into a sparse array, elements with value zero will not be set in the new array.
to_arrays()	Return the indices (keys) and values of elements that have been set.
to_dense()	Make a dense version of the array, just as a plain numpy array.

count_nonzero()
The number of non-zero array elements

Returns

int

classmethod from_dense(*dense*)
Convert a standard (dense) 1D array into a sparse array, elements with value zero will not be set in the new array.

Parameters

dense: array
1D numpy array to convert to sparse form

Returns

sparse: SparseArray

`to_arrays()`

Return the indices (keys) and values of elements that have been set.

Returns

indices: array
indices of elements that have been set.

values: array
values of elements that have been set.

`to_dense()`

Make a dense version of the array, just as a plain numpy array. Un-set values will be zero.

Returns

dense: array
Dense version of array

CHAPTER

TWO

EXAMPLE

This complete example shows how the use the `ParallelMeanVariance` calculator on chunks of data loaded from an HDF5 file.

```
import mpi4py.MPI
import h5py
import parallel_statistics
import numpy as np

# This data file is available at
# https://portal.nersc.gov/project/lsst/txpipe/tomo_challenge_data/ugrizy/mini_training.
↪hdf5
f = h5py.File("mini_training.hdf5", "r")
comm = mpi4py.MPI.COMM_WORLD

# We must divide up the data between the processes
# Choose the chunk sizes to use here
chunk_size = 1000
total_size = f['redshift_true'].size
nchunk = total_size // chunk_size
if nchunk * chunk_size < total_size:
    nchunk += 1

# Choose the binning in which to put values
nbin = 20
dz = 0.2

# Make our calculator
calc = parallel_statistics.ParallelMeanVariance(size=nbin)

# Loop through the data
for i in range(nchunk):
    # Each process only reads its assigned chunks,
    # otherwise, skip this chunk
    if i % comm.size != comm.rank:
        continue
    # work out the data range to read
    start = i * chunk_size
    end = start + chunk_size

    # read in the input data
```

(continues on next page)

(continued from previous page)

```
    z = f['redshift_true'][start:end]
    r = f['r_mag'][start:end]

    # Work out which bins to use for it
    b = (z / dz).astype(int)

    # add add each one
    for j in range(z.size):
        # skip inf, nan, and sentinel values
        if not r[j] < 30:
            continue
        # add each data point
        calc.add_datum(b[j], r[j])

# Finally, collect the results together
weight, mean, variance = calc.collect(comm)

# Print out results - only the root process gets the data, unless you pass
# mode=allreduce to collect.  Will print out NaNs for bins with no objects in.
if comm.rank == 0:
    for i in range(nbin):
        print(f"z = [{ dz * i :.1f} .. { dz * (i+1) :.1f}]    r = { mean[i] :.2f} ± {␣
↪variance[i] :.2f}")
```

CHAPTER

THREE

INSTALLATION

You can install with the command

```
pip install parallel_statistics
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX