Advances in Engineering Software 41 (2010) 92–98

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

PyPnetCDF: A high level framework for parallel access to netCDF files

Vicente Galiano [a], Héctor Migallón [a], Violeta Migallón [b], Jose Penadés [b,*]

[a] Departamento de Física y Arquitectura de Computadores, Universidad Miguel Hernández, E-03202 Elche, Alicante, Spain
[b] Departamento de Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante, E-03071 Alicante, Spain

ARTICLE INFO

Article history:
Available online 29 July 2009

Keywords:
Parallel distribution
Dataset
Performance
MPI
netCDF
Python interface

ABSTRACT

A Python tool for manipulating netCDF files in a parallel infrastructure is proposed. The parallel interface, PyPnetCDF, manages netCDF properties in a similar way to the serial version from ScientificPython, but hiding parallelism to the user. Implementations details and capabilities of the developed interfaces are given. Numerical experiments that show the friendly use of the interfaces and their behaviour compared with the native routines, are presented.

1. Introduction

In scientific and engineering applications, two obstacles hinder the full use of heterogeneous networks of powerful workstations: low-level sequential data access and data representation. Usually, data representations make it difficult to distribute applications across networks or to display output from programs running on different system architectures.

The network Common Data Form (netCDF) [1,2] is a data abstraction for storing and retrieving multidimensional data. NetCDF is distributed as a free software library that provides a concrete implementation of that abstraction. The library provides a machine-independent format for representing large datasets that are created and used by scientific applications. The netCDF software includes C and Fortran interfaces for accessing netCDF data. These libraries are available for many common computing platforms. Many organizations, including much of the climate community, rely on the netCDF data access standard for data storage (see, e.g., http://www.unidata.ucar.edu/packages/netcdf/usage.html).

On the other hand, there are available netCDF interfaces for high level languages that improve its ease of use from Matlab, Ruby, Java and particularly, Python [3]. Python is a dynamic object-oriented programming language that can be used for many kinds of software development. It offers strong support for integration with other languages (C, Fortran, ...) and comes with extensive standard libraries. At the moment, there are several netCDF interfaces for Python but the most popular is ScientificPython [4]. Also, the use of high level environments is common place in science and engineering to enable the development of custom applications, particularly during the early stages of new product or system modelling, simulation, and optimization. These very high level languages make it easy to manipulate high level objects (e.g., matrices), hiding many of the underlying low-level programming complexities from users. They also support rapid code iteration and refinement by enabling an interactive development and execution environment.

Today most scientific applications are programmed to run in parallel environments because of the increasing requirements of data amount and computational resources. It is highly desirable to develop a set of parallel APIs for accessing netCDF files that employs appropriate parallel I/O techniques for reading/writing from hard drive to computer memory. In this way, PnetCDF [5] provides a high-performance and parallel interface for accessing netCDF files from C using the MPI standard [6,7]. However, PnetCDF is only available for programming in C or Fortran. Our goal has been to provide an easy and powerful tool for accessing netCDF files from Python in a parallel programming environment. That is, the resulting interface, PyPnetCDF, enables scientists and engineers to manage netCDF files in a parallel application in the Python high level language, providing an easy-to-use parallel environment that hides the challenges of parallel programming.

This paper is organized as follows. Section 2 describes the format of a netCDF file. Section 3 introduces the main tool for sequential access to netCDF files from Python; this tool will be taken like reference point for the development of our parallel tool. Section 4 presents the PyPnetCDF module, that is, a Python distribution that allows the parallel access from several processes to a same origin of data in netCDF format. Section 5 gives experimental results and Section 6 concludes the paper with conclusions and some ideas for future research.

* Corresponding author.
E-mail address: jpenades@dccia.ua.es (J. Penadés).

0965-9978/$ - see front matter © 2009 Civil-Comp Ltd. and Elsevier Ltd. All rights reserved.
doi:10.1016/j.advengsoft.2009.06.005

2. NetCDF files

The purpose of the network Common Data Form (netCDF) interface is to allow to create, access, and share array-oriented data in a form that is self-describing and portable. "Self-describing" means that a dataset includes information defining the data it contains. "Portable" means that the data in a dataset is represented in a form that can be accessed by computers with different ways of storing integers, characters, and floating-point numbers. NetCDF files can provide a way to encapsulate structured scientific data for using among multiple application programs, and thus, these files can help to support high-level data access and shell-level application programming.

NetCDF is an abstraction that supports a view of data that can be accessed through a simple interface. Array values may be accessed directly, without knowing details of how the data are stored. Auxiliary information about the data, such as what units are used, may be stored with the data.

A netCDF dataset contains dimensions, variables, and attributes, which all have both a name and an ID number by which they are identified. These components can be used together to capture the meaning of data and relations among data fields in an array-oriented dataset. The netCDF library allows simultaneous access to multiple netCDF datasets which are identified by dataset ID numbers, in addition to ordinary file names.

A netCDF dimension is a named integer used to specify the shape of one or more of the variables and it may represent a real physical dimension, such as time, latitude, longitude, or atmospheric level. Dimensions may also be used to relate variables defined on a common grid and provide a natural way to specify coordinates. A netCDF dimension has both a name and a length. A dimension length is an arbitrary positive integer, except that one dimension in a netCDF dataset can have the length *UNLIMITED*.

Variables store the bulk of the data in a netCDF dataset and represent an array of values of the same type. A variable has a name, a data type, and a shape described by a list of dimensions. The header part describes each variable by its name, shape, named attributes, data type, array size, and data offset, while the data part stores the array values for one variable after another, in their defined order. A variable may also have associated attributes, which may be added, deleted or changed after the variable is created. NetCDF supports the most commonly needed variable types for scientific data: scalars and arrays of bytes, characters, integers, and floating-point numbers. In order to support variable-size arrays, netCDF introduces record variables and uses a special technique to store such data. All record variables share the same unlimited dimension as their most significant dimension and are expected to grow together along that dimension. The other, less significant dimensions all together define the shape for one record of the variable. For fixed-size arrays, each array is stored in a contiguous file space starting from a given offset. For variable-size arrays, netCDF first defines a record of an array as a subarray comprising all fixed dimensions; the records of all these arrays are stored interleaved in the arrays defined order. Fig. 1 illustrates the storage layouts for fixed and variable-size arrays in a netCDF file.

NetCDF attributes are used to store data about the dataset. Most attributes provide information about a specific variable and they are called variable attributes. Some attributes provide information about the dataset as a whole and they are called global attributes.

The netCDF API was designed for serial codes. In the netCDF library, a typical sequence of operations to write a new netCDF dataset is to create the dataset; define the dimensions, variables, and attributes; write variable data; and close the dataset. Reading an existing netCDF dataset involves first opening the dataset; inquiring about dimensions, variables, and attributes; reading variable data; and closing the dataset.

3. Accessing netCDF from high level languages

There are multiple references to software packages that may be used for manipulating or displaying netCDF data. The Unidata site [8] provides information about both freely-available and licensed (commercial) software that can be used with netCDF data. NetCDF files can be managed from Python by using, as we have mentioned, the corresponding package integrated with ScientificPython from Konrad Hinsen [4]. In this package, the structure of a netCDF file can be managed using object oriented programming. In this way, ScientificPython defines the *NetCDFFile* class with two standard attributes: "dimensions" and "variables". The values of both are dictionaries, mapping dimension names to their associated lengths, and variable names to variables, respectively. A variable in a *NetCDFFile* object is created using a new class *NetCDFVariable* which allows setting (*assignValue*(...)) or getting (*getValue*(...)) values to or from netCDF files. Also, a *NetCDFFile* class has methods to initialize a file, close it or create dimensions and variables (*createDimension*(...) and *createVariable*(...), respectively). Example 3.1 shows how we can access to netCDF files from Python. Lines 1 and 2 import the Python modules needed in this example. From line 3 to line 15, a netCDF file is created and defined. Lines 6 and 7 define two limited dimensions, while line 8 defines an unlimited one. Line 9 creates a variable and its values are assigned in lines 11–14. Finally, the file is closed in line 15. From line 16 to line 23, the same file is opened and the variables and their values are printed.

Example 3.1 (*NetCDF files management with ScientificPython.*).

```
1. from Numeric import *
2. from Scientific.IO.NetCDF import NetCDFFile
3. file = NetCDFFile(test.nc, w)
4. file.title = Just some useless junk
5. file.version = 42
6. file.createDimension(xyz, 3)
7. file.createDimension(n, 20)
8. file.createDimension(t, None)
9. foo = file.createVariable(foo, Float, (n, xyz))
10. foo.units = arbitrary
11. foo[:,:] = 1.
12. foo[0:3,:] = [42., 42., 42.]
13. foo[:,1] = 4.
14. foo[0,0] = 27.
15. file.close()
16. file2 = NetCDFFile(test.nc, r)
17. for varname in file2.variables.keys():
18.    varl = file2.variables[varname]
19.    print varname,:,varl.shape,;,varl.units
20.    foo = file2.variables[foo]
21.    datal = varl.getValue()
22.    print Data:,datal
23. file2.close()
```

As it has been shown in this example, accessing netCDF files from Python is very simple and intuitive. This tool expands the set of users that can use netCDF files and will be taken as reference point for the development of our parallel tool.

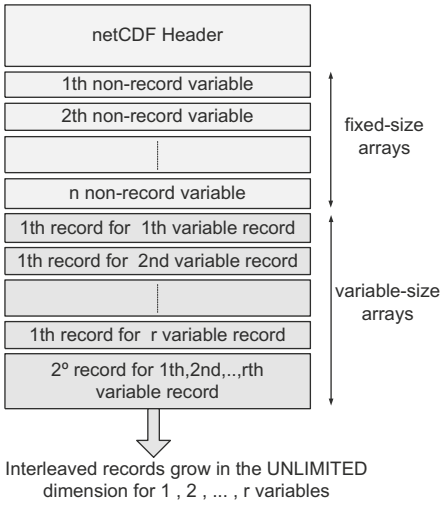


Fig. 1. NetCDF format file.

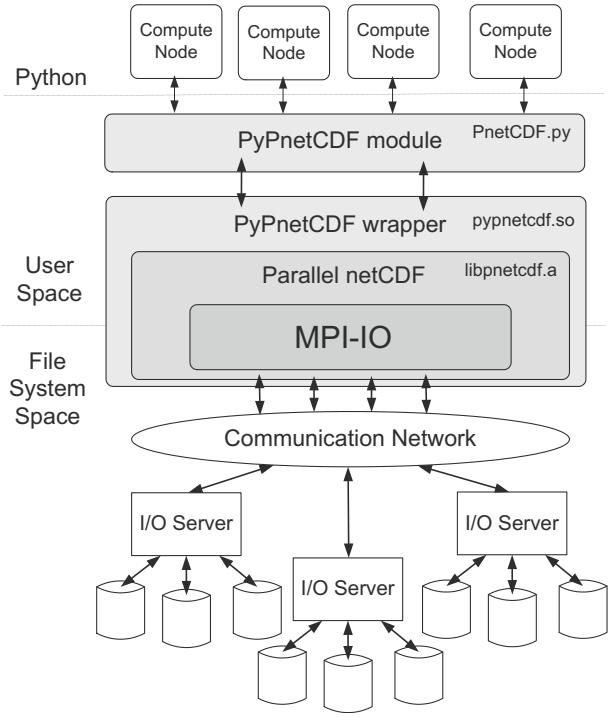


Fig. 2. PyPnetCDF structure.

4. The PyPnetCDF interface

With PnetCDF, the scientific community has a scalable tool for the parallel access to netCDF files. However, this tool is only available for programming in C or Fortran. Our goal is to create an easy and powerful tool for Python, that we have called PyPnetCDF, which would be able to manage netCDF properties in a similar way to the serial version from ScientificPython, but hiding parallelism to the user. For this purpose, a first step to build PyPnetCDF is to make an internal wrapper to the PnetCDF routines. The functionality of these internal wrappers remains unchanged, and they will be used internally to achieve a good interaction between the native routines of PnetCDF and the external wrappers (these are, strictly speaking, the high level user interfaces). These external wrappers were constructed such that parallel environment and data distribution are internally managed by PyPnetCDF. Moreover, since Python users are accustomed to use ScientificPython for managing netCDF files, the layout of the external wrappers follows that of the serial version from ScientificPython. For this reason we have created two PyPnetCDF classes very similar to their serial versions: *PNetCDFFile* and *PNetCDFVariable*. These objects are defined in the module *PnetCDF.py*, showed in Fig. 2, which presents the PyPnetCDF structure. This module acts as an intermediate layer between Python users and the shared objects library *pypnetcdf.so*, which is itself composed by the PnetCDF library and the Python internal wrappers. Following this structure, a Python script using PyPnetCDF is very similar to its serial version and users can easily convert their serial scripts and applications into parallel codes.

Writing Python internal wrappers for C routines can be a very tedious task, especially if a routine takes a lot of arguments but only few of them are relevant for the problems that they solve. For this reason, these internal wrappers have been built with the help of the SWIG wrapper generator [9]. SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages, in particular Python. There are other tools with similar purpose like F2PY [10], but in this case, this tool is devoted for building scripting language interfaces to Fortran programs. We note that while the shared objects library *pypnetcdf.so* allows Python to access low level routines, context, validation and automation are provided at a higher level in the *PnetCDF.py* module.

We want to point out the relationship between PyPnetCDF and PyACTS. PyACTS [11,12] is a collection of carefully designed and written software wrappers to the ACTS tools [13], it also includes other routines written in Python to provide high level users interfaces. These wrappers also provide us with the ability to transparently convert data types between PyACTS modules to support interoperability. Concretely, PyACTS provides some routines for reading and writing netCDF files using PyPnetCDF; the data distribution (or data recollection) is internally performed such that it follows the distribution schemes supported by PyACTS, currently, the two-dimensional block-cyclic distribution of PBLAS and ScaLAPACK [14]. These two libraries are a set of routines for performing basic vector and matrix operations, and solving some linear algebra problems for distributed memory message-passing computers. Hence, PyPnetCDF can also be used in a parallel Python framework in which these kinds of problems appear. Some numerical experiments showing the performance of PyPnetCDF inside PyACTS are presented in Section 5.

Example 4.1 shows how we can get a parallel access to a netCDF file using PyPnetCDF. As we can notice, the source code is very similar to the serial code presented in Example 3.1, and it is also divided into two parts. In the first one, we define and write in a netCDF file and in the second one, we read from that file. In this way, any serial script can be converted to a parallel code by changing a few lines. Concretely, in line 2 the parallel module instead of the serial one is imported and line 3 imports PyACTS; line 4 creates the file in writing mode by calling the constructor *PNetCDFFile*. The netCDF attributes, dimensions and variable creation follow the same structure that the serial example. In line 12, we finish header definition; the method of this line causes a synchronization between processes and assumes that no more header definition will be made in the netCDF file. Notice that the header has been made in collective mode because all processes have executed lines 4–12. From line 13 to line 16, we set some variable values and, in line 17, a hard drive writing is forced in all processes. Finally, the creation of the netCDF file is ended by calling the *close* method.

In the second part of the example, we create a new *PNetCDFFile* called *file2* in reading mode. From line 23 to line 26, all processes print, for each variable, its dimensions, its attributes and the data. The PyACTS package is used in line 20 to print two values: *iam* is an

integer which uniquely identifies each process, and *nprocs* which indicates the number of processes in the parallel execution. Both values are useful and let us to identify how the data distribution is performed.

Example 4.1 (*Example of accessing netCDF files from Python using PyPnetCDF.*).

```
1. from Numeric import *
2. from PyPnetCDF.PNetCDF import *
3. import PyACTS
4. file = PNetCDFFile(test.nc, w)
5. file.title =Just some useless junk
6. file.version = 42
7. file.createDimension(xyz, 3)
8. file.createDimension(n, 20)
9. file.createDimension(t, None)
10. foo = file.createVariable(foo, Float, (n, xyz))
11. foo.units =arbitrary
12. file.enddef()
13. foo[:,:] = 1.
14. foo[0:3,:] = [42., 42., 42.]
15. foo[:,1] = 4.
16. foo.data[0,0] = PyACTS.iam
17. foo.setValue()
18. file.close()
19. file2 = PNetCDFFile(test.nc, r)
20. print **,Process,PyACTS.iam,/,PyACTS.nprocs,**
21. print file2.variables.keys(),;,
    file2.dimensions.keys()
22. for varname in file2.variables.keys():
23.   varl=file2.variables[varname]
24.   print varname,:,varl.shape,;,varl.units
25.   datal=varl.getValue()
26.   print Data:,datal
27. file2.close()
```

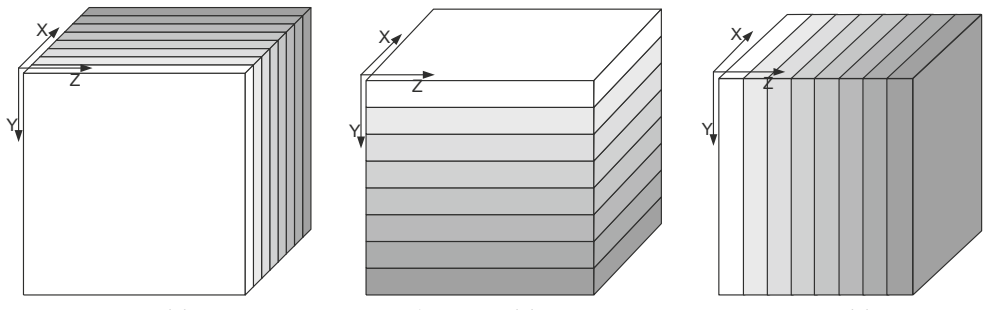
In order to show the parallelism, we center how the data distribution is managed by PyPnetCDF. We will illustrate it by executing this code with four processors. The obtained output is shown in Example 4.2. We must notice that pyMPI [15] is a functional Python interpreter that includes a large subset of MPI functions. PyMPI has extensive support for running parallel Python scripts and has been tested on a number of clusters and other scientific machines. The netCDF variable *foo* is of dimension 20×3 because it is defined using dimensions (*n*,*xyz*). However, when a variable is created or read, it is distributed among processes in partitions along first dimension (in this case, dimension *n*), which is the default option. The data distribution can be optionally specified when a *PNetCDFFile* or a *PNetCDFVariable* is created. Fig. 3 shows the 3D array distributions along each one of the three dimensions. If we want to specify another dimension for the data distribution (e.g., dimension *xyz*), we could do it by modifying line 19 and adding *dist* = (1). In Example 4.2, the default dimension is used for the data distribution and each process stores an array of dimension 5×3. In this way, we can see in line 14 that only rows 0–2 (row 3 is not included in range) of the global array are changed to 42. Notice that only those processes which store elements which have been referenced will modify the array elements; in this case, only process 0/4 modifies the array elements to 42. On the other hand, we can also reference the data by indicating local coordinates: in line 16, the process identifier *iam* is assigned to the element (0,0) of the local data. In summary, this is a simple and transparent way of distributed data management in which the programmer does not have to worry about data allocation.

Example 4.2 (*Output of executing Example 4.1 with four processors.*).

```
PyPNetCDF/test> mpirun -np 4 pyMPI example4.1.py
** Process 0/4 **
['foo']; ['xyz', 't', 'n']
Data: [[0.  4.  42.]
  [42.  4.  42.]
  [42.  4.  42.]
  [1.  4.  1.]
  [1.  4.  1.]]
** Process 3/4 **
['foo']; ['xyz', 't', 'n']
Data: [[3.  4.  1.]
  [1.  4.  1.]
  [1.  4.  1.]
  [1.  4.  1.]
  [1.  4.  1.]]
** Process 2/4 **
['foo']; ['xyz', 't', 'n']
Data: [[2.  4.  1.]
  [1.  4.  1.]
  [1.  4.  1.]
  [1.  4.  1.]
  [1.  4.  1.]]
** Process 1/4 **
['foo']; ['xyz', 't', 'n']
Data: [[1.  4.  1.]
  [1.  4.  1.]
  [1.  4.  1.]
  [1.  4.  1.]
  [1.  4.  1.]]
```

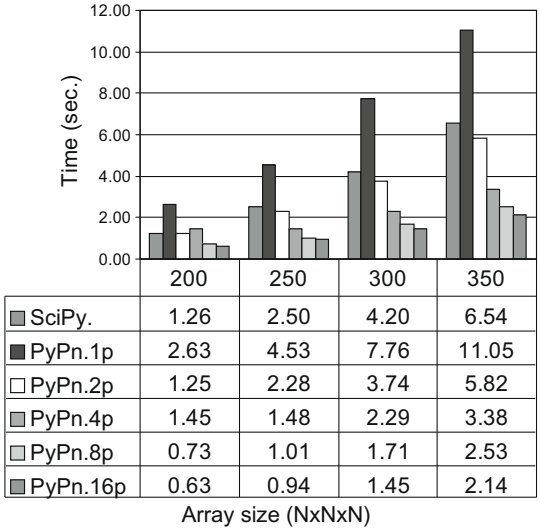
5. Numerical experiments

In the previous section, we have shown how distributed data can be managed from Python using netCDF files. This section evaluates the performance of the current implementation of PyPnetCDF compared to its serial version, included into ScientificPython, and the PnetCDF library. Fig. 4a and b compare, respectively, the reading and writing times of a netCDF file using both the serial version (indicated as "ScyPy.") and the parallel version from PyPnetCDF (indicated as "PyPn.") for different number of processors and array sizes. Basically, the test code reads or writes a



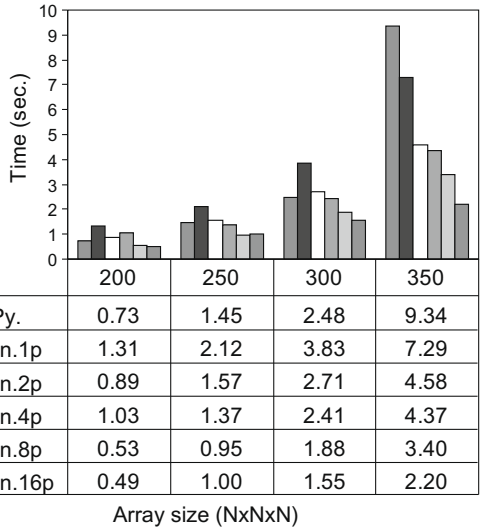
(a) X Partition (b) Y Partition (c) Z Partition

Fig. 3. Different 3D array partitions on eight processors.



	200	250	300	350
SciPy.	1.26	2.50	4.20	6.54
PyPn.1p	2.63	4.53	7.76	11.05
PyPn.2p	1.25	2.28	3.74	5.82
PyPn.4p	1.45	1.48	2.29	3.38
PyPn.8p	0.73	1.01	1.71	2.53
PyPn.16p	0.63	0.94	1.45	2.14

(a) Reading times



	200	250	300	350
SciPy.	0.73	1.45	2.48	9.34
PyPn.1p	1.31	2.12	3.83	7.29
PyPn.2p	0.89	1.57	2.71	4.58
PyPn.4p	1.03	1.37	2.41	4.37
PyPn.8p	0.53	0.95	1.88	3.40
PyPn.16p	0.49	1.00	1.55	2.20

(b) Writing times

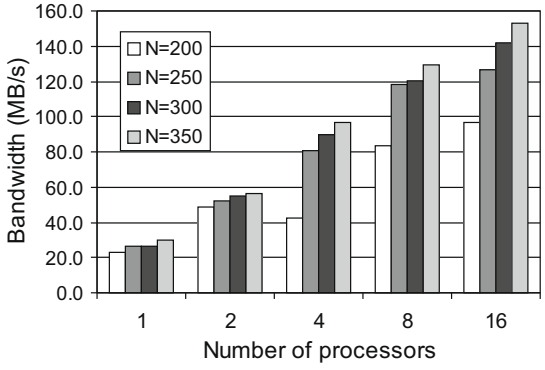
Fig. 4. Reading and writing times with ScientificPython and PyPnetCDF for different number of processors and array sizes.

three-dimensional array field (X,Y,Z) from or into a single netCDF file, where X is the most significant dimension and Z is the least significant dimension. In the parallel case, the test code partitions the three-dimensional array with the default distribution, that is, the data are distributed among processes along first dimension X, as it is illustrated in Fig. 3a.

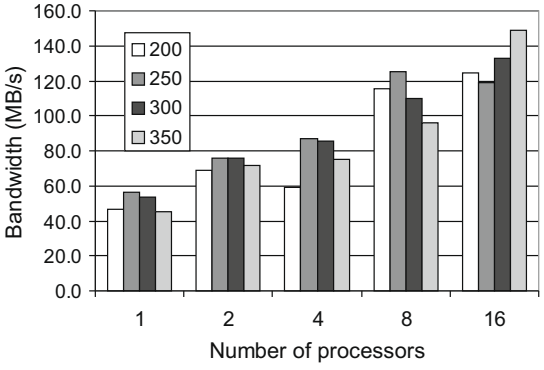
These tests were run in a distributed memory computer, named Seaborg, with 380 computing nodes with 16 processors per node. Each processor has a peak performance of 1.5 GFlops. The disk storage system is a distributed, parallel I/O system called GPFS. Additional nodes serve exclusively as GPFS servers.

Generally, the PyPnetCDF performance scales with the number of processors. In Fig. 4a, reading times with ScientificPython are superior to parallel reading times except when we execute the PyPnetCDF with only one process. This overhead involved in one process is due to additional callings to MPI functions. As expected, PyPnetCDF outperforms the serial netCDF as the number of processors increases. In writing times, PyPnetCDF also scales well with the number of processors but, in this case, the reduction of the parallel time is less than in reading times. The reason of this resides in the fact that, when a parallel writing is performed, all processes must synchronize the access to a unique resource. We would like to point out that the use of the serial version from ScientificPython implies that the data are locally stored, that is to say, there is no distribution of data. Consequently, if need be, an explicit data distribution (with its associated time) must be performed. In other words, the sequential and parallel times of Fig. 4 are not comparable at all because in the first case the data is not distributed among processors. The scalability of PyPnetCDF is also shown in Fig. 5. This figure shows the performance results, in the Seaborg multiprocessor, for reading and writing different datasets (arrays of size $N \times N \times N$) in terms of MB/s (I/O bandwidth) for different number of processors. We can see that the performance increases as the number of processors does.

On the other hand it is interesting to mention that the use of a parallel tool, like PyPnetCDF, may avoid some problems related to memory resources. Usually, many parallel implementations read on a single process from a file and it distributes the data to the rest of processes. If the global array size is bigger than the memory resources of the node, it will not be possible to run the application. With PyPnetCDF, we are not restricted to the memory size of the nodes, and if we want to solve bigger problems, we may add new nodes in order to achieve the needed resources. In this sense, we have integrated PyPnetCDF with PyACTS in such a way that



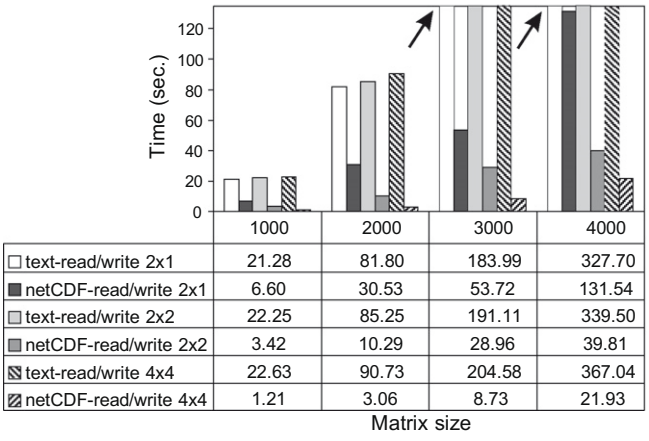
(a) Reading performance



(b) Writing performance

Fig. 5. Parallel performance of PyPnetCDF for different number of processors and datasets (arrays of size $N \times N \times N$).

netCDF files can be read or written from a PyACTS application (using *PnetCDF2PyACTS* and *PyACTS2PnetCDF*, respectively); this integration follows the distribution scheme currently supported by PyACTS (the two-dimensional block-cyclic distribution of ScaLAPACK), and it is performed in a user-transparent fashion, hiding details of the data distribution to the user. The other scalable option for reading or writing text files from PyACTS consists in a couple of routines that read or write a matrix stored as a text file following the communication pattern of the *pdlaread* and *pdlawrite* ScaLAPACK routines, respectively. These routines are called *Txt2PyACTS* and *PyACTS2Txt*, respectively. In order to compare these two scalable options of PyACTS, a distribution and collection test was programmed using both a text file and a netCDF file, for different square matrix sizes and processes grid configurations. Fig. 6 presents the results in the Seaborg multiprocessor. In this figure, "text-read/write" refers to the *Txt2PyACTS/PyACTS2Txt* execution and "netCDF-read/write" corresponds to the *PnetCDF2PyACTS/PyACTS2PnetCDF* test. Obviously, the conclusion is that the netCDF option is more efficient because with PyPnetCDF we get a parallel access to the file, while with the "text-read/write" option an explicit message passing between processes is needed. Note that in Fig. 6 the global matrix exists only as a collection of submatrices in the grid, in other words, no process in the grid ever has the whole global matrix as defined in the file, therefore the scalability is guaranteed.

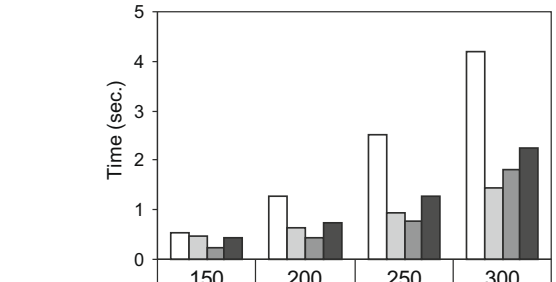


	1000	2000	3000	4000
text-read/write 2x1	21.28	81.80	183.99	327.70
netCDF-read/write 2x1	6.60	30.53	53.72	131.54
text-read/write 2x2	22.25	85.25	191.11	339.50
netCDF-read/write 2x2	3.42	10.29	28.96	39.81
text-read/write 4x4	22.63	90.73	204.58	367.04
netCDF-read/write 4x4	1.21	3.06	8.73	21.93

Fig. 6. Reading and writing times from a text file and from a netCDF file.

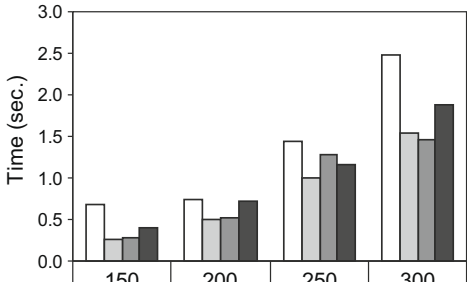
The results shown in Fig. 4 were obtained with the default distribution, that is, data were distributed among processes along the first dimension *X*. Partitioning in the *X* dimension generally performs better than in the *Z* dimension, since the continuity of stored data in memory is a significant parameter. As Fig. 3a shows, in the *X* partition each process only needs to access one time to the netCDF file; however, for the other partitions (Fig. 3b and c), each process needs multiple access to the netCDF file. Fig. 7 shows the performance results for reading and writing different datasets, with different data distribution axis. These tests are executed with 16 processors and we also show the serial times as reference. In Fig. 7a, the times are very similar for first and second data distribution axis, but when the size increases the *X* distribution gets lower times than the other distributions. In the writing tests shown in Fig. 7b, *X* and *Y* distributions are also similar but *X* distribution times are lightly lower. In these tests, differences between distributions are not very significant because the disk storage system has a parallel I/O architecture. Other similar tests were performed in a Linux cluster with 6 2.0 GHz Intel processors and 512MB memory per processor and connected through a 1 Gigabit network switch where the parallel disk storage system is located in one node that shares the hard drive with NFS (Network File System). In this architecture, collecting all I/O data on a single process can easily cause an I/O performance bottleneck and may overwhelm its memory capacity. Fig. 8 shows results on this cluster. Concretely, this figure presents the needed time for reading and writing an array with $200 \times 200 \times 200$ elements for different number of processors and using different data distribution axis (*X*, *Y* or *Z*); it also compares the times using PyPnetCDF from a Python script ("Py-") and using PnetCDF library from a C application ("C-"). Taking as reference of comparison the data distribution axis, it is observed that, in the reading times, the *X* distribution is better that the other distributions, as in the above platform. However, this conclusion changes when writing times are considered; in this case, the needed synchronization of each process waiting for all processes to finish their writing, causes an increase of time.

On the other hand, as we have mentioned, Fig. 8 also compares the times obtained from PyPnetCDF and PnetCDF. The obtained times with Python and C are very similar. In fact, in some cases the PyPnetCDF execution time is lower than that of PnetCDF, the reason being there that the differences between two consecutive executions are comparable to the overhead introduced by



	150	200	250	300
SciPy	0.54	1.26	2.50	4.20
PyPn. 16p x	0.48	0.63	0.94	1.45
PyPn. 16p y	0.23	0.44	0.78	1.81
PyPn. 16p z	0.45	0.74	1.26	2.24

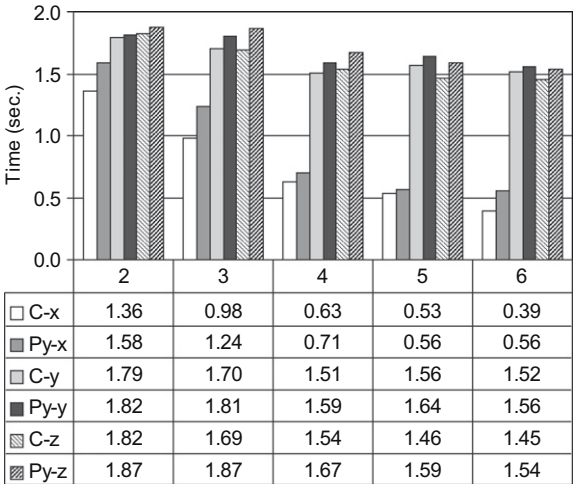
(a) Reading times



	150	200	250	300
SciPy.	0.68	0.73	1.45	2.48
PyPn. 16p x	0.27	0.49	1.00	1.55
PyPn. 16p y	0.28	0.53	1.28	1.46
PyPn. 16p z	0.41	0.73	1.17	1.89

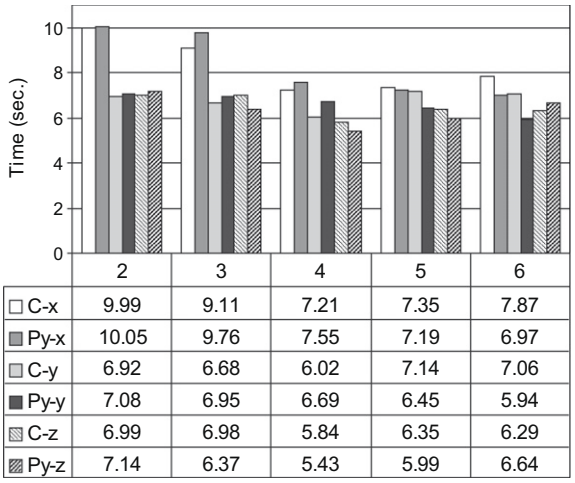
(b) Writing times

Fig. 7. Reading and writing times with ScientificPython and PyPnetCDF for different data distribution axis.



	2	3	4	5	6
C-x	1.36	0.98	0.63	0.53	0.39
Py-x	1.58	1.24	0.71	0.56	0.56
C-y	1.79	1.70	1.51	1.56	1.52
Py-y	1.82	1.81	1.59	1.64	1.56
C-z	1.82	1.69	1.54	1.46	1.45
Py-z	1.87	1.87	1.67	1.59	1.54

(a) Reading times



	2	3	4	5	6
C-x	9.99	9.11	7.21	7.35	7.87
Py-x	10.05	9.76	7.55	7.19	6.97
C-y	6.92	6.68	6.02	7.14	7.06
Py-y	7.08	6.95	6.69	6.45	5.94
C-z	6.99	6.98	5.84	6.35	6.29
Py-z	7.14	6.37	5.43	5.99	6.64

(b) Writing times

Fig. 8. PnetCDF and PyPnetCDF execution times.

PyPnetCDF. Therefore, the results in this figure demonstrate that the overhead introduced by the Python infrastructure is negligible.

6. Conclusions and future research

In this work we have presented a new Python package which provides a parallel access to netCDF files in a simple and intuitive mode. Python examples have demonstrated that PyPnetCDF can be used in a similar form to that given by ScientificPython. With a parallel file system architecture, PyPnetCDF can manage huge netCDF files without worrying about data distribution.

Performance tests prove that PyPnetCDF scales with the number of processors and the Python interface does not involve a penalty in performance. As summary, PyPnetCDF is an intuitive, handy, parallel and powerful tool to manage netCDF files from Python in a parallel architecture. PyPnetCDF is available at http://www.pyacts.org/pypnetcdf and it has been listed in the Unidata Software Page [8] as a useful software for manipulating netCDF data. Future work involves completing the production-quality parallel PyPnetCDF package and providing new functionalities.

Acknowledgements

This work was partially supported by the Spanish Ministry of Science and Innovation under Grant Number TIN2008-06570-C04-04 and FEDER, and by University of Alicante under Grant Number VIGROB-020.

References

[1] Rew R, Davis G, Emmerson S, Davies H. NetCDF user's guide for C; 1997. <http://www.unidata.ucar.edu/packages/netcdf/guidec>.

[2] Rew R, Davis G. The unidata netCDF: software for scientific data access. In: Proceedings of the sixth international conference on interactive information and processing systems for meteorology, oceanography and hydrology, Anaheim, CA; 2001.

[3] van Rossum G, Drake Jr FL. An introduction to python. Network Theory Ltd.; 2003.

[4] Hinsen K. ScientificPython user's guide. Grenoble, France: Centre de Biophysique Moleculaire CNRS; 2002.

[5] Li J, Liao W, Choudhary A, Ross R, Thakur R, Gropp W, et al. Parallel netCDF: a high-performance scientific I/O interface. In: Proceedings of SC2003: high performance networking and computing, Phoenix, AZ; 2003.

[6] Snir M, Otto S, Huss-Lederman S, Walker D, Dongarra J. MPI: the complete reference. Cambridge (MA): The MIT Press; 1998.

[7] Gropp W, Lusk E, Thakur R. Using MPI-2: advanced features of the message passing interface. Cambridge (MA): MIT Press; 1999.

[8] Unidata software page. Software for manipulating or displaying netCDF data. <http://www.unidata.ucar.edu/software/netcdf/software.html>.

[9] Beazley DM. SWIG: an easy to use tool for integrating scripting languages with C and C++. In: Proceedings of the fourth USENIX Tcl/Tk workshop, Monterey, CA; 1996.

[10] Peterson P. F2PY users guide and reference manual; 2005. <http://cens.ioc.ee/projects/f2py2e>.

[11] Drummond LA, Galiano V, Marques O, Migallón V, Penadés J. PyACTS: a high-level framework for fast development of high performance applications. Lect Notes Comput Sci 2007;4395:417–25.

[12] Drummond LA, Galiano V, Migallón V, Penadés J. High-level user interfaces for the DOE ACTS collection. Lect Notes Comput Sci 2007;4699:251–9.

[13] Drummond LA, Marques O. The ACTS collection. Robust and high-performance tools for scientific computing: guidelines for tool inclusion and retirement. Tech. Rep. LBNL/PUB-3175, Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley; 2002.

[14] Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel JW, Dhillon I, et al. ScaLAPACK user's guide. Philadelphia (PA): SIAM; 1997.

[15] Miller PJ. PyMPI – an introduction to parallel Python using MPI. Tech. Rep. UCRL-WEB-150152, Lawrence Livermore National Laboratory, Livermore; 2002. <http://www.llnl.gov/computing/develop/python/pyMPI.pdf>.