Efficient NetCDF processing for big datasets

R.M. Singh [a], **J. Yu** [a] **and G. Podger** [a]

[a] *CSIRO Land and Water Flagship, GPO Box 1666, Canberra ACT 2601*
Email: ramneek.singh@csiro.au

Abstract: NetCDF (Network Common Data Form) is a data format used commonly to store scientific array-oriented data. A number of software tools exist that can be used to process and view NetCDF data. In some cases though, the existing tools cannot do the processing required and thus, we have to write code for some of the data processing operations. One such case is when we have to merge two NetCDF files. Although the core library for read-write access to NetCDF files is written in C, interfaces to the C library are available in a number of languages including Python, C#, Perl, R and others. Python with the Numpy package is widely used in scientific computing where the Numpy package provides for fast and sophisticated handling of the arrays in Python. NetCDF being an array-oriented data format, Python/Numpy combination is a good candidate for NetCDF processing. But in terms of performance, Python is not as fast as C. So if we are looping over the NetCDF data, as is the case when we merge two files, then as the dimension size of the arrays increases, so does the computing time. And if we have to merge 1000's of such files, then we have to look for opportunities to speed up the process. The processing time can be shortened by extracting the 'looping' code out to a custom C module which can then be called from the Python code. The C 'looping' code can then further be parallelised using OpenMP. This gives us best of both worlds, ease of development in Python and fast execution time in C. Problem setup can also reduce processing time if the files are sorted in such a way that the adjoining files show increasing overlap in the dimensions. And if one has access to a cluster of machines, then exploiting the parallelism at a coarser level by running multiple merge processes simultaneously will expedite the process more so than just parallelizing the loop given that the number of machines available is more than the cores in a single machine.

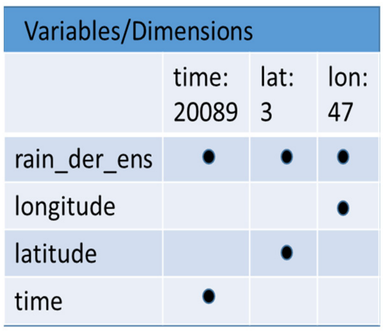
There also exist other use cases like where you have to provide aggregations and averages for NetCDF data. Existing third party tools like NCO and CDO, which themselves are written in C, cannot handle all the scenarios, like leap years or have other undesirable effects on the NetCDF files, like renaming the dimensions. For these cases as well, custom code has to be written to provide a coherent, portable, extensible and fast solution. C/C++ again is a good choice for writing such code keeping in mind the performance benefits it provides for processing large datasets. Modern C++ specially with STL, lambda functions support and smart pointers makes a compelling case to be used over plain C as it does away with issues like dangling pointers and memory allocation for dynamic arrays. Lambda functions combined with STL result in very expressive and concise code unlike the procedural C code.

Keywords: *netCDF, Python, C, OpenMP, high performance computing*

1. INTRODUCTION

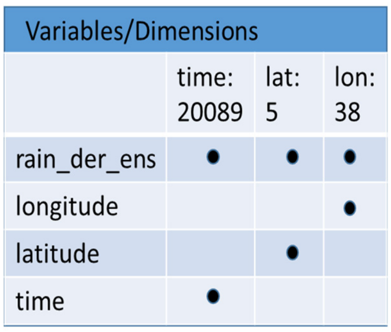
netCDF refers to a machine independent data format specification and a set of libraries for creating, accessing and manipulating data in that format (Rew and Davis, 1990). Along with the set of libraries that come with netCDF, there also exist third party software tools that can do user operations on netCDF files such as concatenation, manipulation of metadata, computing statistics, splitting, etc. These tools cover a range of standard operations that can be done on netCDF files and hence dispense with the need to write additional code for doing so. However, there are situations where certain operations are not handled by the existing tools. In such cases, to manipulate the data we have to either use the core C netCDF library or one of the interfaces built on top of it available in other languages such as Python and Java. Since the netCDF format is an array-oriented data format that is used in scientific environments, the Python-Numpy combination is good choice for doing the manipulations. Python is already used a lot in exploratory scientific computing due to its ease of use and the Numpy package (Van Der Walt et al., 2011) adds the functionality to deal with large multi-dimensional arrays.

The problem that we had was to build daily precipitation gridded data sets for Pakistan at 2.5km resolution over the Indus Basin and Pakistan for more than forty years. To handle the large amount of data the precipitation generation model created more than 2000 netCDF files that contained the daily time series gridded data for smaller parts of the entire area. To create a surface for the entire area files needed to be progressively combined together in space and time to create one large netCDF file. The structure of two such files, File1 and File2, can be seen in Figure 1 and Figure 2. Each netCDF file has three dimensions (*time, lat and lon*) and four variables (*time, latitude, longitude and rain_der_ens*). The *lat* and *lon* dimension length varies from one file to another. And some of the values in the *latitude* and *longitude* variables overlap. The *rain_der_ens* variable contains daily times series of precipitation ensembles for each location. Merging two such files would first involve removing the duplicate values from the *latitude* and *longitude* variables and then performing the merge. So merging the two files, File1 and File2, would produce a file, File3, with structure as shown in Figure 3. There currently exists no third party tools that are capable of doing this. Custom code had to be written to implement this merging process.



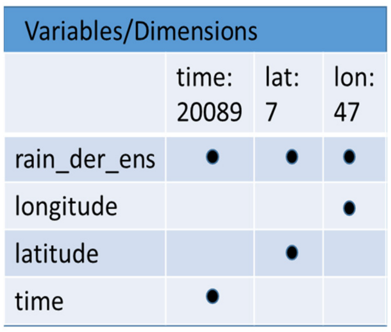
Variables/Dimensions			
	time: 20089	lat: 3	lon: 47
rain_der_ens	●	●	●
longitude			●
latitude		●	
time	●		

Figure 1. File1 structure



Variables/Dimensions			
	time: 20089	lat: 5	lon: 38
rain_der_ens	●	●	●
longitude			●
latitude		●	
time	●		

Figure 2. File2 structure



Variables/Dimensions			
	time: 20089	lat: 7	lon: 47
rain_der_ens	●	●	●
longitude			●
latitude		●	
time	●		

Figure 3. File3 merged file

Visually the process to merge together netCDF files is akin to stitching different smaller pieces of netCDF files, shown in Figures 4-6, representing an area being studied to produce a bigger netCDF file representing the combined area as shown in Figure 7.

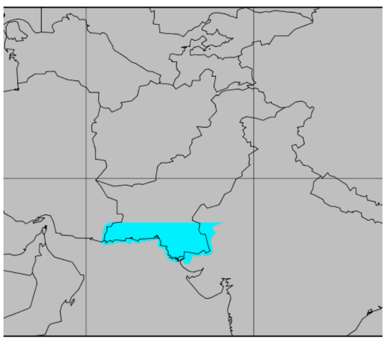


Figure 4. Tile 1

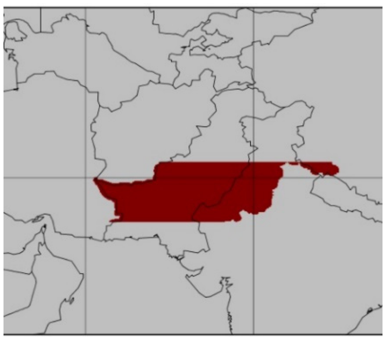


Figure 5. Tile 2

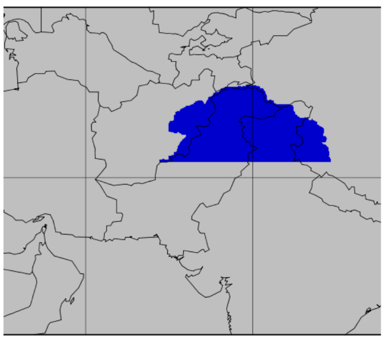


Figure 6. Tile 3

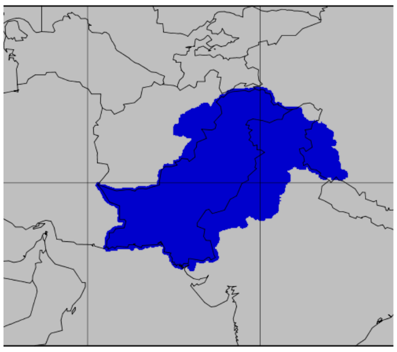


Figure 7. Merged tile

We used Python along with the Numpy package to test and create the merging code. Numpy adds a powerful array object to Python which itself is a language suitable for trying out new ideas quickly. The approach we followed was to code something that worked and then tried to optimize it later. Optimizations include exploiting parallelism at various levels and organizing the computation to speed up processing.

1.1. Initial Attempt Using Python

Following the approach that "Premature optimization is the root of all evil" (Don Knuth), we initially coded the merging process in Python. The netcdf4-python module was used to process the netCDF files.

The module took netCDF files of dimension [20089, 3, 47] (File1-Figure 1) and [20089, 5, 38] (File2-Figure 2) to produce the output netCDF file with dimensions [20089, 7, 47] (File3-Figure 3). For merging, the latitude variable arrays of dimensions 3 and 5 were first concatenated to produce an intermediate array of size 8 out of which the duplicates were then removed to reduce the combined latitude variable array to size 7. Similarly, the merged longitude variable array of size 47 was produced from variable arrays of size 47 and 38. Then the output netCDF file with dimensions [20089, 7, 47] is instantiated and the script makes use of 3 nested loops for iterating over the dimensions and checking whether a particular lat-lon combination has a valid value in any of the input files.

```
for i in range(0, iLength - 1):
    for j in range(0,jLength):
        for k in range(0,kLength):
            #traverse the data in arrays and create merged output
```

A valid value is any value other than the 'missing value'. If a valid value for a lat-lon combination is found in any of the input files, it is copied over to the output file. If a particular lat-lon combination does not exist in any of the inputs files, then the corresponding value for that lat-lon combination in the output file is flagged as a 'missing value' provided by the user when the output file was instantiated. To produce the merged file File3, the Python script took 40 minutes. Taking into account that we have to merge over 2000 files and that the dimensions of the merged file would increase with each completed merger, we needed to consider ways to improve the performance of looping code as it is where the majority of time is being spent.

2. OPTIONS AVAIABLE TO SPEED UP PYTHON:

Python code can be sped up by writing the time critical code in C (Oliphant et al., 2007) either:

a) Directly by writing an extension module in C and then calling the module from Python.
b) Or indirectly by using Cython (Behnel et al., 2011) to write an extension module in a Python like language which is compiled into C and then calling the module from Python.

The approach of writing the module directly in C has the advantage that the code can be distributed as a standalone library whereas Cython has a dependency on Python and as such can't be distributed without Python itself. Hence we chose to speed up our Python code by writing an extension module for it in C.

3. SOLUTION: PYTHON-C VERSION

To increase the performance of the Python code, the looping code that does the actual merger of netCDF files was re-written in C. This C code was then compiled as a Python extension module to make it callable from Python. Doing so reduced the merger time to 2 seconds from the 40 minutes it took the Python script to do the same. That is for a single merge operation. But since we have to merge around 2000 files, if we go along merging 2 files at a time and then using the merged file from the previous step as one of the input for next step, then the file dimensions become bigger and bigger and hence the computation time increases.

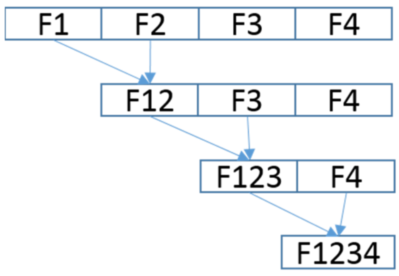


Figure 8. Sequential merge

Figure 8 shows this merging process. For merging 50 files, which are not sorted in any manner, it takes 1 hour and 22 minutes. An improvement can be made to this process by sorting the files such that two consecutive files show geographical proximity along either the latitude or the longitude dimensions. By doing so, we can ensure that the dimensions of the merged file will not increase dramatically at each merger as there would be a lot of overlap between either of those dimensions and hence computation time will remain minimal. So if the same 50 files were merged again using longitudinally adjoining splits (which was the order of splits in our case if the file names were sorted), the merge time was reduced to 56 minutes.

The Python-C version of code consists of two code files: one for Python and one for C (called ConcatenationEngine.c). The concatenation engine now contains the C code that has 3 nested loops to do the data traversal and merger:

```
for (int i = 0; i < iLength - 1; ++i) {
    for (int j = 0; j < jLength; ++j) {
        for (int k = 0; k < kLength; ++k) {
            //traverse the data in arrays and create merged output
```

To use the code written in C from Python, we have to compile the code for each new OS platform we want the code execute on. So for Linux, if we compile the code ConcatenationEngine.c, we get output compiled shared object file ConcatenationEngine.so. This has to be included in the Python script using the import statement:

```
from ConcatenationEngine import *
```

Now to merge 2 netCDF files we run the Python code file which initializes the job and then uses the compiled C shared object ConcatenationEngine.so to process the job by passing the data from the two files to be merged to do the concatenation.

3.1. Parallel Approaches

The Python-C version of merger code can be further improved by exploiting parallelism in the problem. Parallelism can be exploited at 3 levels in this case:

1) At a finer level, the nested loop which iterates over the merged file dimensions to copy over the value from either of the input files does not have any inter-iteration dependency. Hence it is a good candidate to parallelize with OpenMP (Dagum et al., 1998).
2) At a coarser level, the merging process itself can be parallelized by starting a number of these merger processes to merge more than 2 files in one pass.
3) Hybrid approach which combines coarse grain with fine grain parallelism.

In our case, since the size of the job of merging 2000 files is a big one computationally, we made use of a shared high performance cluster to run the job. To efficiently use shared clusters, resource usage requests should be kept to a minimum so that job is scheduled to run sooner. So using the hybrid approach was not preferable as doing so would have increased the resource usage request to cluster schedulers and hence delay the merge process to start on the shared clusters. So we can delay this option as long as not needed. The 'need' would be dictated to by computation time to complete a merger and cluster resource usage.

If we exploit loop level parallelism2 using OpenMP, then for a single merger we can use the n cores on an n core machine. Since OpenMP is a shared memory parallel programming model, we can't distribute the single merge process to other machines in a cluster and are restricted to using the cores on a single machine. If the cluster has more machines than the number of cores in a single machine, then it is advantageous to somehow distribute the problem over the machines in the cluster than to restrict the process to one machine. So for a start, the coarse grain approach of starting a number of merge processes simultaneously to merge more than 2 files in each pass seems reasonable. We can continue with this approach until the number of files to merge become equal to n*2 where n is the number of cores on a single machine in the cluster. In that case, the problem could either be parallelized at a finer or a coarser level or parallelized at both levels. Resource usage of the cluster and the way the job scheduler is setup would then dictate the way we proceed.

3.2. Parallel Solution

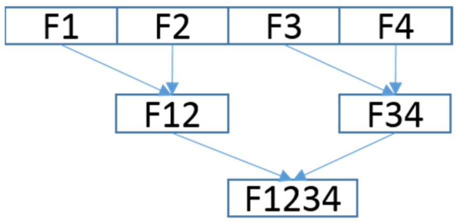


Figure 9. Parallel merge

Initially the coarse grain approach was used to merge the files which was later changed to the hybrid approach when the computation time taken by each merger became unacceptable. It involved setting up more than one merge for each pass. Figure 9 shows 2 passes of the merge process. In the first pass, 2 merge processes are started and produce files F12 and F34. And in the second pass the final output is produced. We can think about the problem setup in such a way that the individual files to be merged are the leaves in a binary tree and we perform the merger at each level of the tree in parallel. Given n files at the start to merge, the number of passes required to produce the final merged file is binary logarithm of n (floor ($\log_2 n$) + 1).

For exploiting loop level parallelism through OpenMP, we used pragma directive 'omp parallel for' and then specified which variables are supposed to be private to each thread and which variables can be shared across threads to run the computation. OpenMP is supported by all the major compilers on various platforms. Even if the compiler does not support OpenMP, the code will still compile and run correctly serially. The 3 nested loops from the function cConcat parallelized using OpenMP are shown below:

```
//set the number of threads desired for simulation
omp_set_num_threads(numberOfThreads);
int i, j, k;
#pragma omp parallel for default(none) shared(iLength,jLength,kLength,latArrayC
ombined,lonArrayCombined,latArray1,lonArray1,latArray2,lonArray2,rain_der_ensIn
1,rain_der_ensIn2,rain_der_ens_Concatenated) private(i,j,k)

for (i = 0; i < iLength - 1; ++i) {
    for (j = 0; j < jLength; ++j) {
        for (k = 0; k < kLength; ++k) {
            //traverse the data in arrays and create merged output
```

The number of threads to use to parallelize the loops can be set by calling the omp_set_num_threads function.

4. RESULTS

The merge process was run on the in house shared cluster where each node had dual 10 core Intel Xeon E5-2660 V3 processors running at 2.6 GHz with 25 MB cache. Available memory on compute nodes varied from 128GB to 3TB.The cluster was running Linux and supported SLURM job scheduling system.

Now in our case, we started the merge for more than 2000 files and noted the CPU time and elapsed time for each merger. A plot of elapsed time and CPU time for some of these merged files is shown in Figure 10.
As files are merged in each pass, their dimensions keep getting bigger as is the time taken to do the merge. For the ninth pass, the time taken to merge the files increases substantially. A total of 5 files were produced in the ninth pass. The graph shows that to produce one such file, it took 102 minutes.

From that pass onward, instead of using the coarser approach to parallelize the files, the hybrid approach of parallelizing the loop using OpenMP as well as starting more than one merge process was taken. This allowed us to allocate more cores for each individual merger. Hence the elapsed time drops down as compared to the CPU time. The final output file of dimensions [20089, 557,878] and size 36.5 GB was produced using 19 cores on a single machine and took 54 minutes of Wall time compared to 14 hours and 54 minutes of CPU time.

Figure 11 shows the memory usage for mergers as the file size increases. To produce the final merged file, 226 GB of RAM memory was used. Since the maximum memory supported by the cluster was 3TB, time was not spent on making the system memory efficient.

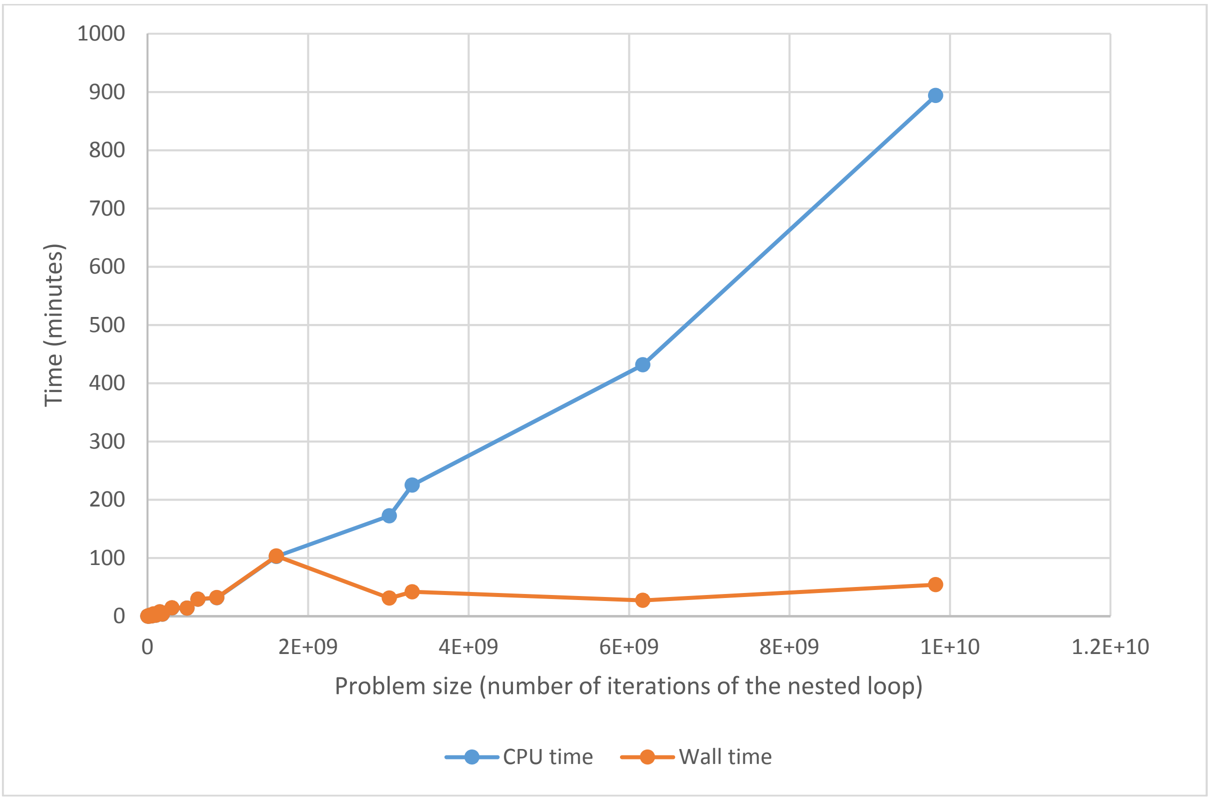


Figure 10. Time taken for mergers as file size increased

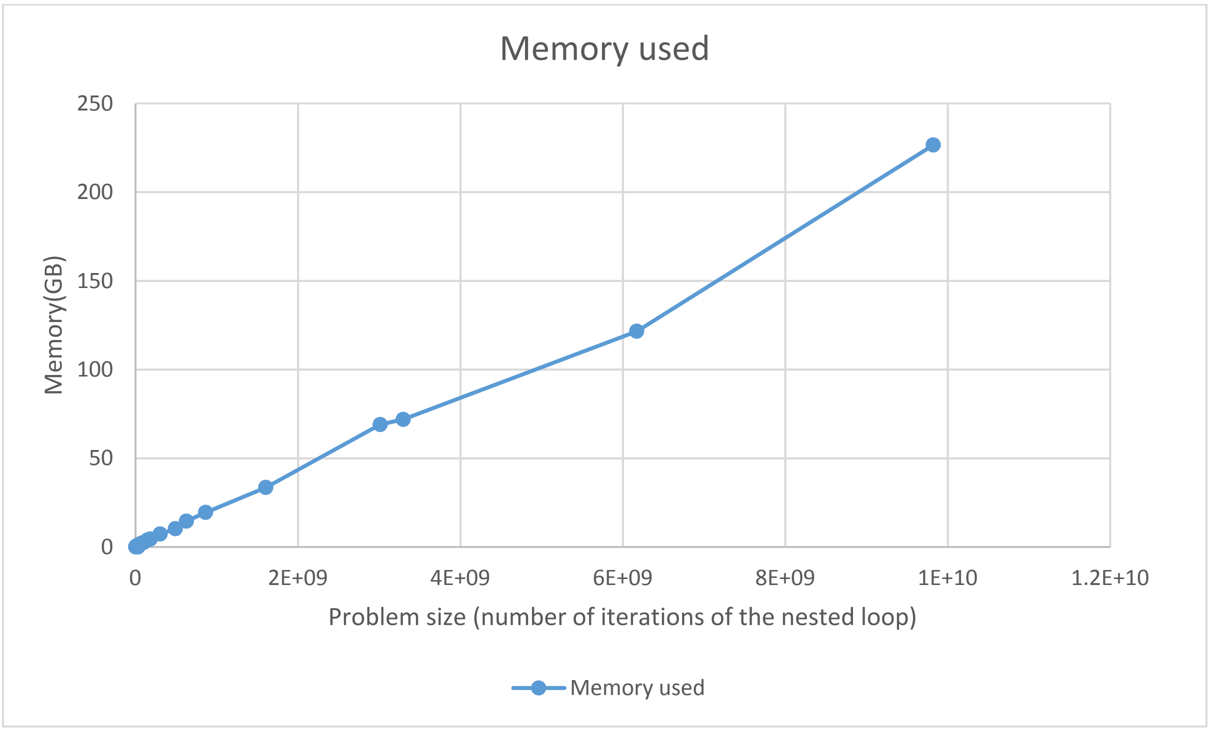


Figure 11. Memory used for mergers as file size increased

5. CONCLUSIONS AND RECOMMENDATIONS

Python and Numpy provide a good combination for testing an idea of processing netCDF files. But as the size of the processing increases, optimizations to reduce the run time need to be made. Computationally intensive code can be extracted to C as Python has the ability to interact with extension modules written in C. Smarter problem organization/set up can substantially reduce the processing time. To capitalise on this the user has to find correlations between the file structure and the type of processing being done. User also has to look for areas showing parallelism, be it fine grained or coarse grained, which can be exploited. Problem setup and choices around exploiting one particular type of parallelism also depend on the type of computer hardware at disposal for the execution and number of cores available, be it at the machine level or the cluster level and whether the hardware is shared or not. Memory usage is another concern that needs to be kept in mind while designing the solution. If not enough RAM memory resource is available for computation, then perhaps doing netCDF processing purely in C might be more efficient than using either Python or Python-C combination (Gouy, n.d.).

ACKNOWLEDGMENTS

The Sustainable Development Investment Portfolio (SDIP) project is funded by the Australian government's Department of Foreign Affairs and Trade (DFAT). OzNome project is funded by CSIRO strategic funding. This work was funded by SDIP and OzNome.

REFERENCES

Rew, R. and Davis, G. (1990). NetCDF: an interface for scientific data access. *IEEE computer graphics and applications*, *10*(4), pp.76-82.

Van Der Walt, S., Colbert, S.C. and Varoquaux, G. (2011). The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, *13*(2), pp.22-30.

Dagum, L. and Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering*, *5*(1), pp.46-55.

netcdf4-python module. *http://unidata.github.io/netcdf4-python/*, Accessed: 17, 1, 2017

Oliphant, T.E. (2007). Python for scientific computing. *Computing in Science & Engineering*, *9*(3), pp.10-20.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D.S. and Smith, K. (2011). Cython: The best of both worlds. *Computing in Science & Engineering*, *13*(2), pp.31-39.

Top of Form

Gouy, I., n.d. The Computer Language Benchmarks game. *http://benchmarksgame.alioth.debian.org*, Accessed: 17, 1, 2017