Light-Weight Parallel Python Tools for Earth System Modeling Workflows

Kevin Paul*, Sheri Mickelson†, John M. Dennis‡, Haiying Xu§, and David Brown¶
National Center for Atmospheric Research
1850 Table Mesa Drive
Boulder, Colorado 80305
Email: *kpaul@ucar.edu, †mickelso@ucar.edu, ‡dennis@ucar.edu, §haiyingx@ucar.edu, ¶dbrown@ucar.edu

***Abstract*—In the last 30 years, earth system modeling has become increasingly data-intensive. The Community Earth System Model (CESM) response to the next Intergovernmental Panel on Climate Change (IPCC) assessment report (AR6) may require close to 1 Billion CPU hours of computation and generate up to 12 PB of raw data for post-processing. Existing post-processing tools are serial-only and impossibly slow with this much data. To improve the post-processing performance, our team has adopted a strategy of targeted replacement of the "bottleneck software" with light-weight parallel Python alternatives. This allows maximum impact with the least disruption to the CESM community and the shortest delivery time. We developed two light-weight parallel Python tools: one to convert model output from time-slice to time-series format, and one to perform fast time-averaging of time-series data. We present the motivation, approach, and results of these two tools, and our plans for future research and development.**

***Keywords*-Scientific computing; Data processing; High performance computing; Parallel processing**

I. Introduction

Climate modeling is a massively data-intensive area of research. Substantial improvements to computing power have resulted in climate modeling codes that are capable of spatial and temporal resolutions that were only figments of scientists' imaginations 30 years ago. With this dramatic increase in resolution, the size of the raw computational output has increased proportionally. The enormous increase in computing power—evidenced by the increase in single-processor computing speeds, system memory, and parallel computing platforms—has seriously outpaced I/O performance. As a result, these raw data files have become so large in the last decade that previously effective serial post-processing analysis and diagnostics software have become crippled under the massive amount of raw data. In this paper, we focus our attention on just one of these climate modeling codes, the data-intensive problems it is currently facing, and the targeted parallel Python-based solutions that we are proposing to address this code's problems.

The Coupled Model Intercomparison Project Phase 5 (CMIP5) [1] assembled and compared data produced by 20 climate modeling groups from around the world in preparation for the Intergovernmental Panel on Climate Change (IPCC) [2] Assessment Report 5 (AR5) [3]. One of the models participating in the CMIP5 project was the Community Earth System Model (CESM) [4], with development centered at the National Center for Atmospheric Research (NCAR) [5]. CESM is a fully-coupled model comprised of a *coupler* code and four *component* codes: the Community Atmosphere Model (CAM) [6], the Community Land Model (CLM) [7], the Parallel Ocean Program (POP) [8], and the Community Ice CodE (CICE) [9]. CESM produced close to 2.5 PB of raw data, of which 170 TB were submitted to the Earth System Grid (ESG) [10] as part of the CMIP5 project. For CMIP6 [11], estimates of 12 PB of raw data output are expected from CESM's contribution.

The 2.5 PB of CESM data generated for the CMIP5 project was produced in NetCDF-formatted [12] *time-slice* files. A single time-slice file contains all of the data generated by one of CESM's component codes for a "slice" (*i.e.*, a short duration) of time, nominally a single (or small number) of simulation time steps. Historically, these files have been the *de facto* standard output format for all of CESM's component codes. However, as both spatial and temporal resolution have increased, and as model improvements result in more time-dependent variables being written to output (currently between 30 and 300, depending upon the component and simulation case), the size of these time-slice files have grown and become unwieldy for scientists to analyze after the data has been generated and stored to disk (*i.e.*, post-processing).

During most post-processing analyses, scientists need to extract only a few variables spanning a relatively long duration of time (*i.e.*, many time-slices). To accomplish this with archived time-slice files, scientists must download all of the time-slice files that contain the necessary variables and that span the required duration of time. This data can be massive in size, depending upon the duration of time spanned by the analysis and the number and size of variables in each time-slice file. Alternatively, one can *subset* the data on the server side and extract only the necessary data needed for analysis. The Research Data Archive (RDA) [13] at the National Center for Atmospheric Research's (NCAR) is one such service for efficient online subsetting. Unfortunately, this subsetting service is not sufficient for frequent day-to-day use.

To solve this problem, CESM developers and scientists have recently switched from storing time-slice files to storing *time-series* files. This time-series format is required for submission of the CMIP data to the Earth System Grid (ESG). Each *time-*

978-1-4799-9926-2/15/$31.00 ©2015 IEEE

TABLE I
DATASETS USED FOR TIME-SLICE TO TIME-SERIES TRANSFORMATION TESTING

Component	Component Code Name	Resolution	TS Size (GB)	MD Size (MB)	TS Variables
Atmosphere	CAM-FV	1 deg	28	< 1	122
	CAM-SE	1 deg	30	1	132
		0.25 deg	1050	18	198
Ice	CICE	1 deg	6	15	117
		0.1 deg	433	1055	112
Land	CLM-SE	1 deg	8	18	297
		0.25 deg	80	288	150
Ocean	POP	1 deg	188	18	114
		0.1 deg	2958	1285	34

series file contains only one time-dependent variable produced by the component code, but it contains the variable data spanning a much longer duration of time. This means that an analysis that requires only a few variables spanning a long duration of time can be done without the scientist downloading significantly more data than is absolutely necessary and/or subsetting the data before downloading it. Unfortunately, this switch to time-series files for long-term storage of CESM data has presented a number of new challenges that are currently being addressed by NCAR scientists.

The first new challenge is the actual conversion from time-slice files—which are still currently the standard output from CESM component codes—to time-series files for long-term storage and submission to the ESG. For the CMIP5 project, it took close to 15 months to post-process model simulation data, which is comparable to the time it took to generate the raw data with CESM itself. A large part of the 15 month conversion time was due to the use of serial tools to perform the necessary formatting and data transformations. With current forecasts of data volume for the next CMIP6 suggesting a 5$\times$ increase in data volume in the 2017-2018 timeframe, it is clear that the current set of serial tools used for converting the data for the CMIP5 project are insufficient. To overcome this hurdle, a new solution must be found soon for the time-slice to time-series transformation.

Another new challenge for the CESM workflow is the production of climatology (*i.e.*, averaged) data from the time-series files themselves. Climatology files typically store multiple variables, averaged over space or time, in one file for diagnostic purposes such as generating plots. The computation of these averages from time-slice files has not been considered a significant bottleneck in the CESM workflow, even though such computations with the current set of serial tools can take close to 7 hours for some high-resolution datasets. Unfortunately, the same canonical serial averaging tools, when acting on time-series files instead of time-slice files, can perform up to 20$\times$ slower, suddenly making the averaging operation a serious bottleneck. Therefore, a new solution is required to provide efficient averaging.

In this paper, we present our approach to solving the two specific problems described above. Due to the need for solutions that are in place before CMIP6 in 2017, we have chosen targeted replacement of the specific "bottleneck software" with light-weight parallel Python alternatives. Our contributions, detailed in the following sections of this paper, are listed below.

- We have chosen a design methodology that is consistent with the programming community's "Principle of Least Astonishment" (POLA) [14], [15]: implementing incremental solutions that target bottleneck software and result in *maximum impact* with the *least disruption* to the CESM community. An added benefit of this approach is that the solutions are *deliverable in the shortest possible timeframe*.
- We have created the PYRESHAPER [16]: a light-weight parallel Python tool to efficiently solve the CESM workflow problem of *transforming time-slice files into time-series files*.
- We have created the PYAVERAGER [17]: a light-weight parallel Python tool to efficiently solve the CESM workflow problem of *computing climatology averages directly from time-series files*.

II. APPROACH & DESIGN

There are many ways to approach the solution to the problems discussed in the introduction (§I). When a software solution already exists but is problematic (as in our case with the CESM workflow), these various approaches typically lie somewhere along the spectrum from *minimal transformation* of the existing solution to *maximal transformation*. Maximally transformative change is *required* when the existing solution no longer works for *any* use case. In contrast, *zero* transformative change is usually desired when there is no perceived problem with the existing solution.

While the existing serial solutions are well understood to be outdated by both the development and scientific user communities, a fully transformative solution that implements the most advanced techniques and best software practices would require more development time and resources than are available. More transformative solutions have been proposed that have taken many years to develop and still do not provide a comprehensive solution (see §VII). With CMIP6 starting in the 2017–2018 timeframe, a solution must be in the hands

of scientists well before the start of the CMIP6 runtime, roughly in mid-2016. Even if a fully transformative solution could be delivered into the scientists hands in this time-frame, the additional cost to learn the new user interface would most likely hamper CESM productivity. Therefore, we have determined that a more transformative solution—while admittedly desirable—is not a practical option.

Under these constraints, and borrowing from the Principle of Least Astonishment, we have opted for a design strategy that targets the individual "bottleneck" scripts in the CESM workflow and replaces them with light-weight parallel Python tools. The advantages of this approach, as compared to a more transformative approach, are listed below.

- **Incremental.** The greater problem can be broken into small "sub-problems" to which a very small, individual software solution can be devised. Smaller, incremental solutions can be easily replaced later with minimal disruption if a solution proves insufficient in the future.
- **High impact.** Solutions can be targeted to extremely problematic or underperforming sections of the existing solution, so that change on the scale of wholesale transformation can be achieved with a small, incremental change.
- **Rapidly deployed.** Each incremental solution can be small enough to allow very rapid prototyping and productization.
- **User-centric.** Overall usage changes can be kept to a minimum, if required at all.
- **Minimal maintenance.** Again, related to the incrementalism of the approach, solutions are small enough to be easily maintained by as few people as possible. We aim for maintenance to be only a fraction of a single developer's time.
- **Hierarchical.** More complex tasks and solutions can be built on top other incremental solutions.

III. New Products

Conforming to the design approach described in the previous section (§II), we have developed two light-weight Python tools as solutions to the specific problems described in the introduction:

1) the transformation of time-slice files into time-series files, and
2) the computation of climatology (time-averaged) files from time-series files.

To address each of these problems, we have created the PyReshaper and the PyAverager, respectively.

The original serial scripts, which these two new tools were created to replace, are Unix shell scripts that make command-line calls to the NetCDF Operators (NCO) [18]. We chose to write their replacements in Python for a number of reasons that address the priorities of the design methodology described in the previous section.

1) Python lends itself well to rapid prototyping. This is partly due to the large and growing scientific Python development community, but this is also partly because Python is a very high-level language.
2) Python scripting seamlessly integrates into the existing script-driven CESM workflow.
3) Python code can be easily extended, or shared with future code, to provide new solutions.
4) Python scripts (using the mpi4py package) can be easily parallelized and injected into the existing MPI batch-job environment used with CESM.
5) Python code, with minimal dependencies, can be easily ported to other supercomputing systems.

Our new light-weight Python replacements are very small in size. The PyReshaper is roughly 900 lines of code (excluding comments, blank lines, and documentation), and the PyAverager is roughly 2000 lines of code (excluding comments, blank lines, and documentation). Both depend upon a common module containing simple utility classes and functions, called the asaptools, that is roughly 1100 lines of code (excluding comments, blank lines, and documentation). It would be easy for a single developer to support and maintain both tools. The PyReshaper was prototyped in a day, and it was first released a few months later. The PyAverager was prototyped in a week, and was first released in early 2015. Both tools leverage the capabilities of the numpy [19], mpi4py [20]–[23], and PyNIO [24] Python packages.

IV. Testing Methods & Datasets

Both of the light-weight Python tools described in the previous section (§III) were tested for correctness and performance on NCAR's Yellowstone IBM iDataPlex cluster [25], [26], the machine on which much of CESM's contributions to CMIP6 will be generated. The Yellowstone cluster features 4,536 Intel Sandy Bridge computational nodes with 16 processor cores per node and 2 GB of memory per processor core (*i.e.*, 32 GB per node). Each Yellowstone computational node has a theoretical maximum I/O read/write speed of roughly 1.25 GB/sec, with 90 GB/sec peak from the filesystem. Yellowstone uses a centralized file service called the *GLobally Accessible Data Environment* (GLADE) with the high-performance GPFS and 16 PB of usable capacity to provide common access to the Yellowstone storage system from any of the NCAR computing resource systems.

We evaluate both tools using the same NetCDF datasets. These datasets are representative output of various component codes in the CESM model: the Community Atmosphere Model (CAM), the Community Land Model (CLM), the Parallel Ocean Program (POP), and the Community Ice CodE (CICE). All of these sample datasets from their respective CESM components span 10 years of monthly *time-slice* files. The detailed properties of each component dataset can be found in Table I. The total size of the time-series data (TS Size) and metadata (MD Size) are shown in separate columns, indicating the total sizes of each kind of data in the entire dataset (*i.e.*, all 10 years). The number of time-series variables (TS Variables) is shown in the sixth column.

Within these datasets, there can be 2D (*e.g.*, latitude $\times$ longitude), 3D (*e.g.*, latitude $\times$ longitude $\times$ level), or even higher-dimensional variables. A 2D double-precision (*i.e.*, 8 byte) variable on a 0.1 degree grid (*i.e.*, $3600 \times 1800 = 6480000$ values) measures about 50 MB of data per time step. For 10 years of monthly data (*i.e.*, 120 months), this measures 5.9 GB. A 3D double-precision variable on the same horizontal (*i.e.*, latitude $\times$ longitude) grid with 30 vertical levels would then be 30$\times$ larger per time step (about 1.5 GB), and the total amount of data for 10 years of monthly data would be 175 GB. Assuming about 100 3D time-series variables per time-slice, this suggests an average time-slice file size for such a dataset to be about 150 GB. A time-series file, spanning all 10 years of a 3D variable's data, would be about 175 GB.

We make an important distinction between *time-series data* and *metadata*. Not all of the data output from the CESM component codes is time-series data. To a varying degree, dependent upon the individual component code itself, a fraction of the variables in each dataset are considered "metadata." This metadata is used to describe the other data in the dataset, and it can consist of both time-dependent and time-independent data. Coordinate data (*i.e.*, the data describing each dimension) is one common example of metadata, though not the only example. According to the CF Conventions [27], this metadata must accompany each component variable in the file, thus making the file entirely "self-describing." However, this means that each time-slice file must duplicate every *time-independent* metadata variable across all time-slice files, inefficiently using disk space. Alternately, each time-series file must contain every piece of metadata, potentially resulting in an even larger disk-space inefficiency. The amount and kinds of metadata in each dataset can influence the performance of both the PyReshaper and PyAverager operations.

As mentioned, the canonical tools that have existed to perform the same operations for which the PyReshaper and the PyAverager were designed are serial shell scripts using the NetCDF Operators (NCO). We compare the results generated by our light-weight Python tools with the results generated by the NCO scripts for each dataset. If the results of the new tools are the same as the results of the NCO scripts, the tool's results are deemed "correct." After correctness was verified for each dataset, we then compared performance of the new tools against the performance of the NCO scripts. The performance results are described in the *Technical Results* sub-sections of the PyReshaper and PyAverager sections (§V-B and §VI-B). Performance is measured in terms of both *duration* (the total time to run the tool and produce correct output) and *throughput* (the total *input* data size divided by the duration).

In our testing, the PyReshaper was used to convert the datasets shown in Table I from time-slice format to time-series format, and the PyAverager was used to generate identical climatology files from both the original time-slice data and the time-series data. In practice, the PyAverager will act on the time-series data generated by the PyReshaper, but for these testing purposes the PyAverager tests compared the performance and accuracy of the climatology generation step from both the time-slice and time-series data. In the past, the original serial NCO scripts to generate the climatology files acted directly on the time-slice data itself.

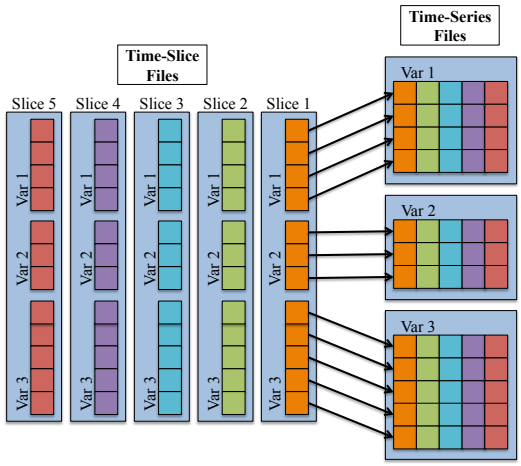


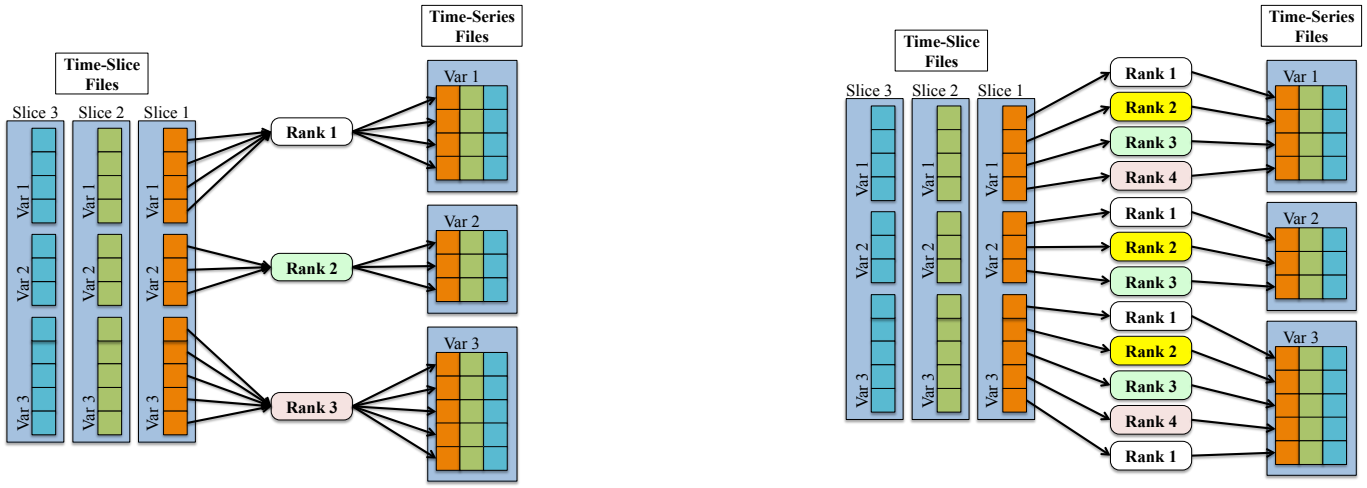
Fig. 1. A graphical representation of the *time-slice* to *time-series* ("slice-to-series") data transformation operation, where each *slice* box represents a time-slice file and each *series* box represents a time-series file. The smaller boxes inside each time-slice box represent individual data elements of each variable at the time associated with the time-slice.

V. The PyReshaper

As mentioned in the introduction (§I), the desire to adopt time-series file formats for long-term data storage has resulted in a number of challenges, one of which is the challenge of efficiently converting the raw NetCDF time-slice files, generated by the individual CESM component codes, into time-series format. This is the task for which the PyReshaper was designed.

A graphical representation of this transformation can be seen in Figure 1. Each *slice* box in the Figure 1 represents a time-slice file and its contents, and each *series* box represents a time-series file and its contents. This transformation of the data is akin to a transpose operation, where the axes being transposed are *variable* and *time*. In the *time-slice* format, the *time* axis spans multiple files, and in the *time-series* format, the *variable* axis spans multiple files.

Also mentioned in the introduction, the "slice-to-series" transformation using NCO took close to 15 months with 170 TB of CESM data, suggesting an average operational speed of 4.5 MB/sec. This is only slightly faster than typical broadband download speeds and less than 20% of typical hard-drive write speeds, and a supercomputer with a good parallel filesystem can perform more than an order of magnitude better than a single hard drive. Some of the inefficiency associated with this transformation comes from simple human inefficiency, because the transformation operation was not automated as part of the CESM workflow.



(a) Task Parallelism

(b) Data Parallelism

Fig. 2. A graphical representations of the *time-slice* to *time-series* ("slice-to-series") data transformation operation, as performed with (a) *task-parallel* or (b) *data-parallel* implementations. With a task-parallel implementation, each time-series variable is processed by a single rank and only that single rank writes to the time-series file. With a data-parallel implementation, the data of each time-series variable is distributed across multiple ranks, and these multiple ranks simultaneously write to the time-series file.

A. Technical Approach

Our incremental approach to solving this problem was to identify the individual NCO-based "slice-to-series" transformation script used in the existing CESM workflow. After identifying the NCO script and timing it on our dataset, we reproduced the same serial functionality in a similar Python implementation. Then, our approach to speeding up the "slice-to-series" transformation was to identify obvious ways of rapidly parallelizing the prototype Python code. We identified two (2) general approaches to parallelization: *task parallelization* and *data parallelization*. We describe the two approaches below.

With *task parallelization*, we parallelize over the output variables, ideally giving the responsibility of writing a single time-series file to a single parallel rank. Figure 2a depicts this approach. This involves every time-slice file being opened and read by every rank, but only the data needed for the parallel rank's assigned time-series file (*i.e.*, the time-series variable and necessary metadata) is written. This is the simplest parallelization approach, as it effectively implements the serial approach independently on every parallel rank. There is no messaging required. However, task parallelism has limited scalability, as it can only be parallelized up to the number time-series variables. This is the approach that was used in the PyReshaper, using the external Python modules mpi4py [20]–[23] and PyNIO [24].

The second approach, *data parallelization*, involves dividing the responsibility of writing each time-series file amongst the parallel ranks. Figure 2b depicts this approach. Data parallelism solves the limited scalability problem of the task parallel approach, and it allows for much greater flexibility and potential performance. However, data parallelism is more complex and harder to efficiently implement. A prototype Fortran code, using the Parallel I/O Library (PIO) [28], [29] and MPI, was created with an implementation of this approach, called the NcReshaper.

B. Results

Parallel tests were performed with both the PyReshaper (using serial NetCDF3, NetCDF4 classic and NetCDF4 libraries) and the NcReshaper (using the pNetCDF library [30], [31]). The NetCDF4 format uses the HDF5 [32] data format, and can provide lossless compression via the zlib library. The parallel tests were run using 16 cores on Yellowstone, with only 4 cores per node to limit saturating the I/O bandwidth and to provide enough memory to each job for some of the larger (high-resolution) datasets. There is nothing special to the choice of 16 cores, other than that it was less then that fewest number of time-series variables found in any of the test datasets. The duration and throughput results are shown in Figures 3a, 3b, 3c, and 3d. We have shown the serial NCO results with the parallel results for comparison purposes only.

These results show significant speedups with both the PyReshaper and the NcReshaper tools compared with the original serial NCO scripts, as would be expected. For low-resolution data, the PyReshaper with NetCDF4 classic format is between 7× and 22× faster than the existing serial NCO solution. For high-resolution data, the PyReshaper with NetCDF4 classic format is between 11× and 17× faster than the existing serial NCO solution. While the speedups in the low-resolution data can be less significant than the high-resolution data, the slowest low-resolution job was less than 15 minutes, and most were less than 5 minutes.

The throughput results show the PyReshaper outperforming the NcReshaper, suggesting that the task-parallel implementation is better for almost all datasets. Using NetCDF4

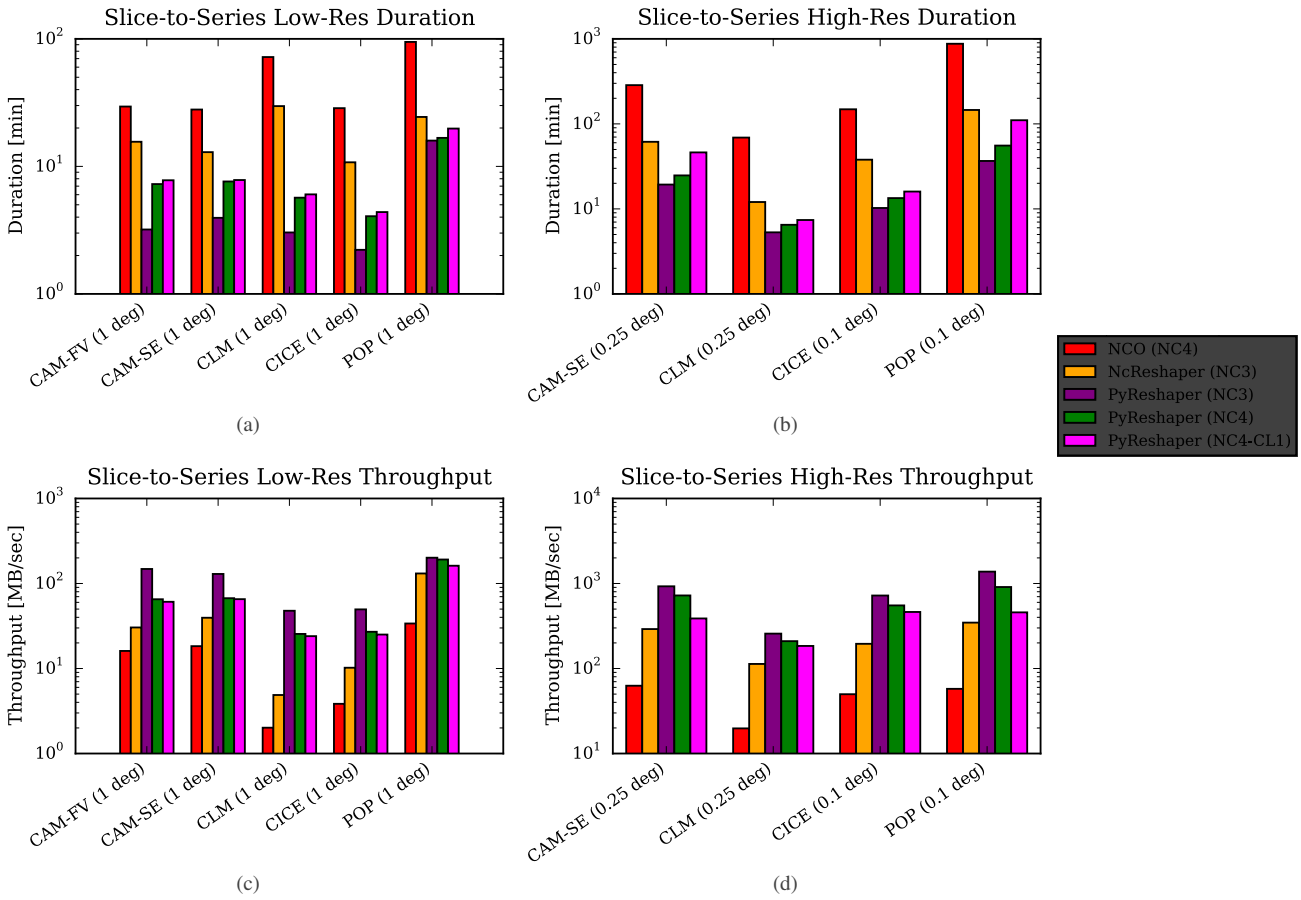


Fig. 3. The duration (in minutes) and throughput (in MB/sec) of the slice-to-series transformations for the low- and high-resolution datasets. The "NC3", "NC4", and "NC4-CL1" labels indicate that the PyReshaper was run with NetCDF3, NetCDF4 classic, and NetCDF4 compressed (compression level 1) output. Note the use of a logarithmic scale on the y-axes.

classic output reduces the performance, and using NetCDF4 compressed output (compression level 1) shows a further reduction in throughput. In some cases, the NcReshaper outperformed the PyReshaper when the PyReshaper was set to use NetCDF4 compressed output. It is also of note that none of the throughput results reach the theoretical maximum I/O bandwidth, but the largest datasets approach between 60% and 90% of the theoretical bandwidth.

In our last test, we performed a scaling study of the PyReshaper tool with some of the larger datasets, the high-resolution CAM-SE (0.25 deg), CLM (0.25 deg), and CICE (0.1 deg) datasets. Due to the task-parallel nature of the PyReshaper, one can expect that the amount of parallelism available from the PyReshaper is limited by the number of time-series variables in the time-slice files. Figure 4 shows the results of this study, clearly showing saturation as the number of cores (16 per node) used approaches the number of time-series variables in the datasets. Recall from Table I that the CAM-SE (0.25 deg) dataset has 198 time-series variables, the CLM (0.25 deg) dataset has 150 time-series variables, and the CICE (0.1 deg) dataset has 112 time-series variables. The run times to which each dataset saturates depend greatly on the size of each time-series variable in the dataset and the amount of metadata.

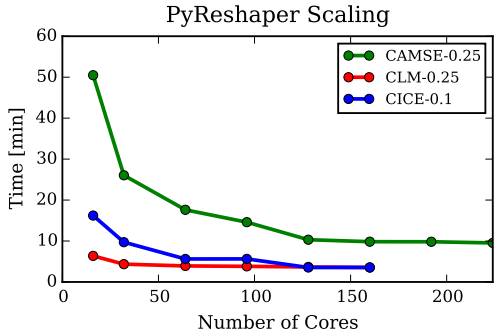


Fig. 4. This figure shows the parallel scaling of the PyReshaper on the high-resolution CAM-SE (0.25 deg), CLM (0.25 deg), and CICE (0.1 deg) datasets. The total time to complete the slice-to-series transformation is shown versus the number of cores (16 cores per node) used for the operation on Yellowstone. One can see the performance saturate as the number of cores used for the operation approaches the number of time-series variables in the dataset.

VI. The PyAverager

The analysis of climate simulation data typically involves two phases: the calculation of climatological average files, and

the execution of component specific diagnostic packages. The diagnostic packages primarily generate large numbers of plots, which we did not identify as a significant bottleneck in the workflow. Instead, our effort concentrated on the I/O intensive calculation of the climatology files themselves. This is for what the PyAverager was designed.

The climatology files are typically produced by calling NCO commands in serial from a shell script. By itself, this serial approach is problematic. However, as mentioned in the introduction, the transition from time-slice to time-series format has stressed this existing approach even further. Currently, the diagnostic packages expect as input a small number of climatology files containing all fields. Using time-slice input files, a single NCO command can generate the necessary inputs. However, with time-series data, a three step process is required:

1) extraction of the variables from the time-series data,
2) computation of an average, and
3) appending of the averaged results to the final file.

In each of these three steps, one must open a file, read the values, compute, and write. This approach, with time-series files, slowed the production of the climatology files down by a factor of around 20×.

The current method is also a poor match for high-resolution datasets. Not only do the high-resolution datasets take considerably more time to generate the climatology files, but the serial NCO jobs can often fail due to memory constraints. While previous work added task parallelism into the diagnostic packages using Swift [33], [34], the Swift approach still retains some of the underlying memory limitation issues. Furthermore, because Swift is designed to orchestrate the execution of computationally expensive tasks, it is not an ideal solution for more data-intensive calculations where data movement should be minimized.

A. Technical Approach

Again, our approach to speeding up the averaging operations within the CESM diagnostic packages was similar to the approach described for the PyReshaper. We identified the climatology computation scripts that were part of the CESM diagnostic packages and reimplemented them in serial Python. Guided by our experiences with the PyReshaper, we chose a new task parallelization approach, using the external Python modules mpi4py [20]–[23] and PyNIO [24], to improve the performance of the averaging operations. The resulting light-weight Python tool we named the PyAverager.

While the PyAverager currently utilizes task parallelization, the execution differs slightly from the task parallelization within the PyReshaper (and the one achievable with Swift). Our approach partitions the available MPI tasks into sub-communicators. Each sub-communicator receives a local list of averages to compute and each average is computed in parallel within the sub-communicator. Each sub-communicator reserves a dedicated MPI task whose only job is to create the output climatology file. All other MPI tasks within the sub-communicator read one time-series variable at a time and accumulate into an averaging buffer. The averaging buffer is then sent to the MPI rank that subsequently generates the climatology file. Typically, climate analysis involves multiple types of averages. For example, in addition to a yearly average, multiple seasonal averages, as well as monthly averages, are typically required. Figure 5 illustrates this operation where a single type of average is necessary.

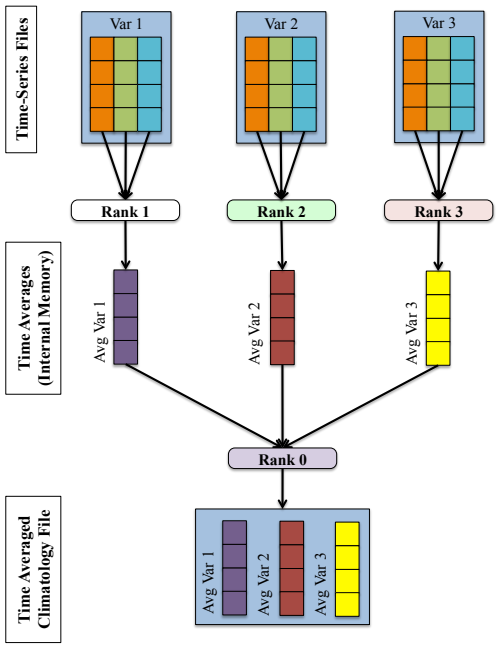


Fig. 5. A task parallel approach to averaging time series files. A list of fields for temporal averaging are decomposed across MPI ranks. Once the temporal average has been calculated, the results are gathered to single rank where it is then written to disk. This process exists for each sub-communicator.

B. Results

To test the performance of the PyAverager, we calculated the averages of the various CESM component datasets described in Table I, using both the time-slice format and time-series formats as input. For each year of the CAM and CLM models, twelve monthly mean files, four seasonal averages, and one yearly average were computed. For each year of the POP and CICE time-slice data, we calculated the yearly average. The CICE time-series format convention dictates that the data be split into separate northern and southern hemisphere files with data loss at the tropics. While both the time-series and time-slice datasets contain the same number of variables, time-series files at both CICE resolutions are smaller than the time-slice datasets because of the grid size differences. This split results in the generation of twenty yearly average files, one for each year and hemisphere. This must be considered when comparing the time-slice and time-series results for the CICE dataset. All methods used produced output in the NetCDF4 format without compression.

We measured the time to creating the averages from time-slice and time-series data using the original NCO scripts,

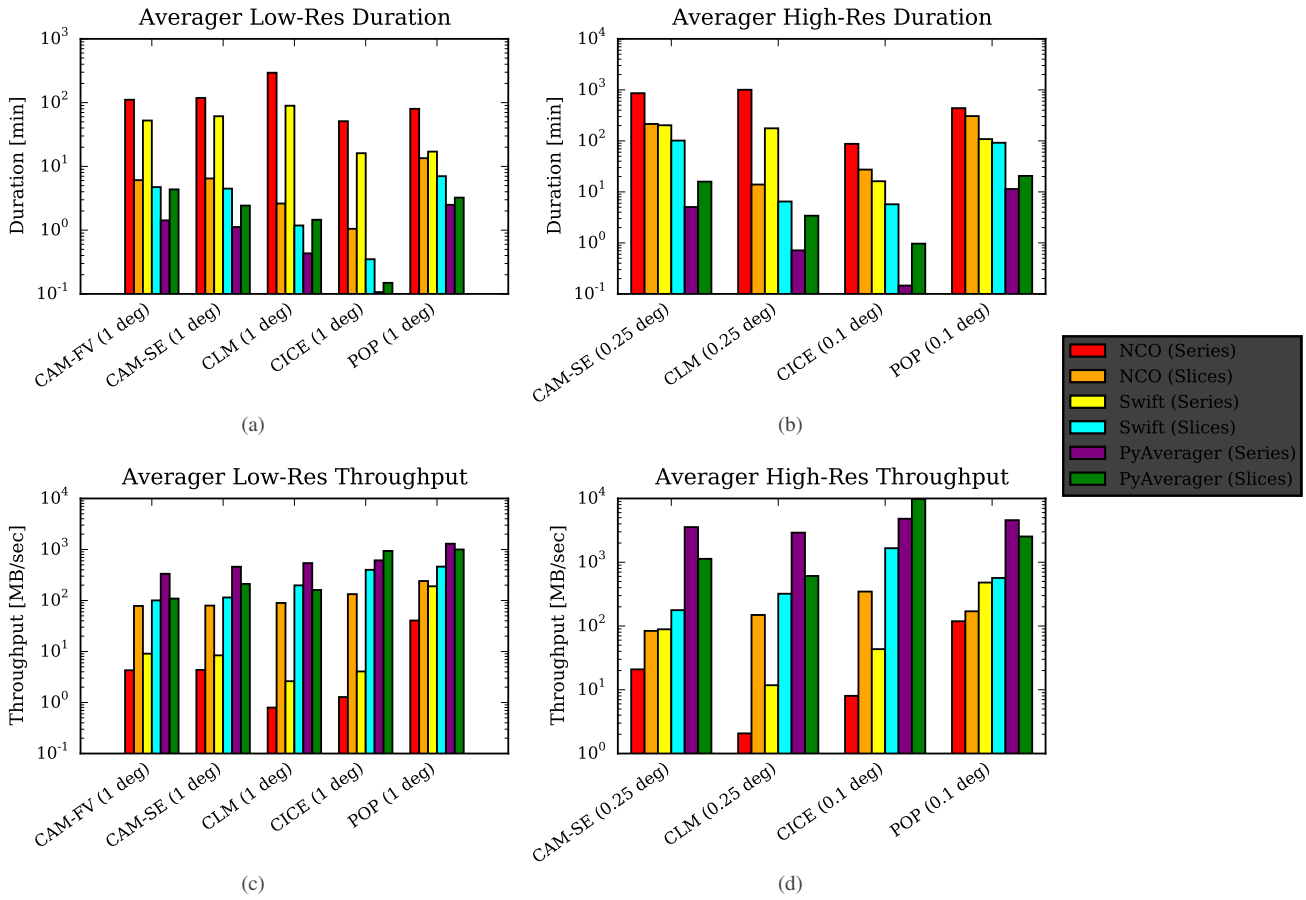


Fig. 6. The I/O throughput (in MB/sec) and duration (in minutes) of the climatological averaging for the PyAverager, compared with the original serial NCO method and the limited parallel Swift method. All methods were outputted in the NetCDF4 format.

Swift scripts, and the PyAverager. All of the averages were computed on the Yellowstone cluster, but because of memory and scheduling requirements, different averages were sent to different queues. The low-resolution results for the serial NCO method were run on one dedicated node on Yellowstone. Because of the scheduling requirements of Swift, the low-resolution Swift scripts were run in the Geyser queue on Yellowstone using 4 nodes with 4 cores per node (16 cores total). The low-resolution PyAverager jobs were executed on Yellowstone using 20 nodes with 8 cores per node (160 cores total). The high-resolution results for the serial NCO method required more memory and required us to run within the GPGPU and BigMem queues on Yellowstone. Again because of Swift's scheduling requirements, the high-resolution Swift scripts were run in the Geyser queue on Yellowstone using 4 nodes with 4 cores per node (16 cores total), except for the POP 0.1 degree results from time-slices. These were generated in the BigMem queue using 4 nodes with 1 core per node (4 cores total). The high-resolution PyAverager were all run on Yellowstone using 20 nodes with 8 cores per node (160 cores total), except for the POP 0.1 degree results from time-slices. These were generated in the Geyser queue.

The timing results for the NCO scripts, Swift scripts, and the PyAverager are illustrated in Figure 6. The figures show that the worst performing option is NCO acting on time-series input, with these calculations taking the longest amount of time. While the durations for Swift and NCO acting on time-series input show that some improvement was gained by adding task parallelism, the concerning problem is that in most cases the averages computed from time-series input still take more time to compute than the original serial NCO method acting on time-slice input. The PyAverager results show a considerable amount of improvement over the results for NCO acting on time-series input, and are comparable or faster than the results for the original NCO method acting on time-slice input. Within the four categories of CESM data that we compared (low-resolution time-slice input, low-resolution time-series input, high-resolution time-slice input, high-resolution time-series input), the PyAverager was able to reduce the time it takes to compute the climatological averages for CESM data by a factor at least 2X, 26X, 3.5X, 400X, respectfully.

We see a similar pattern in the throughput results. The throughput increased using Swift over the equivalent serial NCO tool method. We also see a severe performance hit while computing the averages from time-series input using both the serial NCO and Swift methods. This further demonstrates

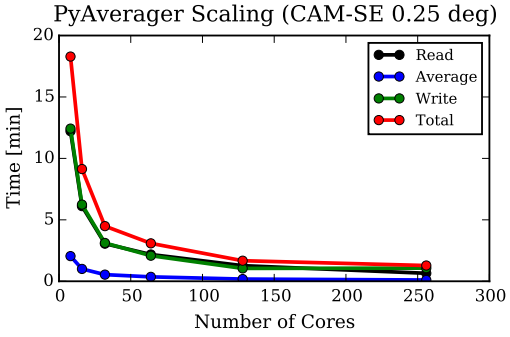


Fig. 7. The execution time for the PyAverager computing twelve monthly mean files, four seasonal averages, and yearly average for the CAM-SE 0.25 data set. The Read, Write, and Average measurements represent the max time it took a task read the data slices from disk, compute the average of these slices, and then write the results to disk. The Total measurements represent the total time the PyAverager took to run.

the need for a new tool to calculate climatological averages from time-series input. We can also see that the PyAverager throughput is substantially better than all of the other methods acting on time-series input.

In Figure 7, we examine the scalability of the PyAverager. We see that the reading of variables takes roughly the same amount of time as the writing operation. The total time, read time, and write time all scale to about 128 cores (16 compute nodes with 8 cores per node). Based on these results, the application is I/O bound. In future versions of the PyAverager, we will examine ways to improve this performance.

VII. Related Work

The need to rapidly and efficiently post-process climate simulation data has been recognized for a number of years. We next describe related work and explain our decision to develop new Python-based tools instead of utilizing existing tools.

The Parallel Climate Analysis Toolkit (ParCAT) [35] provides similar functionality to the PyAverager, but it utilizes C (instead of Python) and the Parallel NetCDF (PNetCDF) library to enable parallel access to NetCDF files. ParCAT utilizes both task and data parallelism, and it is a command-line tool that fits well into a script-driven workflow, such as CESM. However, it does not provide the exact functionality needed, and it's implementation in C makes it less easily extensible than a similar object-oriented Python tool.

The Ultrascale Visualization Climate Data Analysis Tools (UV-CDAT) project [36]–[38], which uses ParCAT, aims to provide a comprehensive set of tools to enable the analysis of large scale climate data sets, including parallel visualizing and post-processing computation. Its large size and difficulty to install impose significant barriers to adoption within the CESM workflow. Additionally, if UV-CDAT were to be adopted, its whole-sale transformation of the CESM user interface would require user re-education and would consequently take a toll on user productivity.

Another existing tool is the Parallel Toolkit for Extreme Climate Analysis (TECA) [39], which is a parallel toolkit whose focus is on performing a particular set of operations to identify extreme events in large climate datasets. TECA is still currently under development, and it was not been publicly released yet.

The NetCDF Operators (NCO) [18] are a set of executables which perform single atomic operations on NetCDF input files. The NCO toolset provides the functionality of both the PyReshaper and the PyAverager and more. While NCO has supported parallelism through both MPI and OpenMP, the parallelism is typically targeted at computationally expensive routines and not I/O intensive operations. The atomic design of its operators typically require multiple different NCO operators to be used in series, often resulting in the use of the disk system to store intermediate results.

Similar in nature to the NetCDF Operators, the Climate Data Operators (CDO) [40] are also capable of much the same operations. However, CDO supports a pipelined-based parallelism which allows for the chaining of multiple atomic operators while maintaining intermediate products in memory. Unfortunately, while the operations of the PyReshaper and PyAverager can be represented by a chain of operations, it is not possible to minimize the amount of data movement either from disk or through memory using pipeline parallelism.

Lastly, Pagoda [41] is designed to be a set tools that mimic the NCO tools, but relies on Global Arrays, PNetCDF, and MPI for parallelization. However, these tools have some of the same problems as the NCO tools and would still require three separate steps when averaging time-series files.

VIII. Conclusion

Both of the tools described in this paper, the PyReshaper and the PyAverager, took only days to prototype and a few months to implement and deploy. Since they replace existing scripts in the CESM workflow, and can operate in the same batch-job MPI framework, they seamlessly fit into the existing CESM workflow with minimal disruption to CESM users. Their codebase is small, and their complexity is low, making them approachable and easily understandable from the perspective of an existing CESM user. Additionally, the way these tools were developed, as light-weight parallel Python tools, allows them to be easily extended to address other problems that might arise in the future.

For each targeted bottleneck in the CESM workflow that each tool addresses, we achieved substantial performance improvements. We achieved an average of 13× speedup over the existing CESM workflow tools with the PyReshaper on high-resolution data. The PyAverager achieved an average speedup of 400× over the existing CESM workflow tools on high-resolution time-series data. These numbers alone show a substantial success, though we believe that both tools can be improved further. There are a number of improvements that we are planning for future releases of these tools, including data-parallelism in the PyReshaper and PyAverager, which should substantially improve write-speeds and scalability. Still,

even in their simple initial releases, they satisfy the need of the CESM community.

Other challenges resulting from the adoption of time-series formats for long-term storage certainly exist. Certainly, as well, new challenges will be uncovered as research continues. We believe that by judicial application of the same incremental development approach, building off of the tools that we have already created, we will address these issues with equal success as they arise.

ACKNOWLEDGMENTS

The authors would like to thank all of the CESM developer staff at NCAR for their assistance in developing and testing the PyReshaper and the PyAverager.

REFERENCES

[1] "CMIP5: Coupled Model Intercomparison Project Phase 5," http://cmip-pcmdi.llnl.gov/cmip5/.

[2] "IPCC – intergovernmental panel on climate change," http://www.ipcc.ch/organization/organization.shtml.

[3] T. F. Stocker, D. Qin, G.-K. Plattner, M. M. Tigner, S. K. Allen, J. Boschung, A. Nauels, Y. Xia, V. Bex, and P. M. Midgley, Eds., *Climate Change 2013: The Physical Science Basis*. Cambridge University Press, 2013, Working Group I of the Intergovernmental Panel on Climate Change (IPCC).

[4] "CESM: Community Earth System Model," http://www2.cesm.ucar.edu/.

[5] "National Center for Atmospheric Research: Home," http://ncar.ucar.edu/.

[6] "CAM: Community Atmosphere Model," http://www.cesm.ucar.edu/models/cesm1.2/cam/.

[7] "CLM: Community Land Model," http://www.cesm.ucar.edu/models/cesm1.2/clm/.

[8] "POP: Parallel Ocean Program," http://www.cesm.ucar.edu/models/cesm1.2/pop2/.

[9] "CICE: Community Ice CodE," http://www.cesm.ucar.edu/models/cesm1.2/cice/.

[10] "Earth system grid," https://www.earthsystemgrid.org/home.htm.

[11] G. A. Meehl, R. Moss, K. E. Taylor, V. Eyring, R. J. Stouffer, S. Bony, and B. Stevens, "Climate model intercomparisons: Preparing for the next phase," *Eos, Transactions American Geophysical Union*, vol. 95, no. 9, pp. 77–78, 2014. [Online]. Available: http://dx.doi.org/10.1002/2014EO090001

[12] "Unidata: NetCDF," http://www.unidata.ucar.edu/software/netcdf/.

[13] "CISL Research Data Archive," http://rda.ucar.edu/.

[14] P. Seebach, "The cranky user: The Principle of Least Astonishment," http://www.ibm.com/developerworks/library/us-cranky10/, August 2001.

[15] A. Ronacher, "Python and the Principle of Least Astonishment," http://lucumr.pocoo.org/2011/7/9/python-and-pola/, July 2011.

[16] "The PyReshaper (GitHub)," https://github.com/NCAR-CISL-ASAP/PyReshaper.

[17] "The PyAverager (GitHub)," https://github.com/NCAR-CISL-ASAP/pyAverager.

[18] The NCO Project, "NCO Homepage," http://nco.sourceforge.net/, 2014.

[19] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[20] L. Dalcin, P. Kler, R. Paz, and A. Cosimo, "Parallel Distributed Computing using Python," *Advances in Water Resources*, vol. 34, no. 9, pp. 1124–1139, 2011.

[21] L. Dalcin *et al.*, "Bitbucket: mpi4py/mpi4py," https://bitbucket.org/mpi4py/mpi4py.

[22] L. Dalcin, R. Paz, and M. Storti, "MPI for Python," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 1108–1115, 2005.

[23] L. Dalcin, R. Paz, M. Storti, and J. D'Elia, "MPI for Python: performance improvements and MPI-2 extensions," *Journal of Parallel and Distributed Computing*, vol. 68, no. 5, pp. 655–662, 2008.

[24] "PyNIO," https://www.pyngl.ucar.edu/Nio.shtml.

[25] "Yellowstone: High-performance computing resource," https://www2.cisl.ucar.edu/resources/yellowstone.

[26] R. Loft, A. Andersen, F. Bryan, J. M. Dennis, T. Engel, P. Gillman, D. Hart, I. Elahi, S. Ghosh, R. Kelly, A. Kamrath, G. Pfister, M. Rempel, J. Small, W. Skamarock, M. Wiltberger, B. Shader, P. Chen, and B. Cash, "Yellowstone: A dedicated resource for earth system science," in *Contemporary High Performance Computing: From Petascale Toward Exascale, Volume Two*, 1st ed., ser. CRC Computational Science Series, J. S. Vetter, Ed. Boca Raton: Chapman and Hall/CRC, 2015, vol. 2, p. 262.

[27] B. Eaton, J. Gregory, H. Centre, B. Drach, K. Taylor, and S. Hankin, "NetCDF Climate and Forecast (CF) Metadata Conventions: Version 1.6, 5 December, 2011," http://cfconventions.org/Data/cf-conventions/cf-conventions-1.6/build/cf-conventions.pdf.

[28] J. M. Dennis, J. Edwards, R. Loy, R. Jacob, A. A. Mirin, A. P. Craig, and M. Vertenstein, "An application-level parallel i/o library for earth system models," *Int. J. High Perf. Comput. Appl.*, vol. 26, pp. 43–53, February 2012, doi: 10.1177/1094342011428143.

[29] J. M. Dennis, J. Edwards, R. Jacob, R. Loy, A. Mirin, and M. Vertenstein, "PIO: Parallel I/O Library," https://code.google.com/p/parallelio/, 2014.

[30] J. Li, W. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallager, and M. Zingale, "Parallel netCDF: A high-performance scientific I/O interface," in *In Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, November 2003.

[31] "Parallel netCDF: A Parallel I/O library for NetCDF file access," http://trac.mcs.anl.gov/projects/parallel-netcdf.

[32] The HDF Group, "Hierarchical Data Format, version 5," http://www.hdfgroup.org/HDF5/, 1997.

[33] M. Woitaszek, J. M. Dennis, and T. Sines, "Parallel high-resoution climate data analysis using swift," in *4th Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS)*, Seattle, WA, November 2011.

[34] S. Mickelson, R. Jacob, M. Wilde, and D. Brown, "How to use the task-parallel omwg/amwg diagnostic packages," http://www.cesm.ucar.edu/events/ws.2012/Presentations/Plenary/parallel.pdf.

[35] B. Smith, D. M. Ricciuto, P. E. Thornton, G. Shipman, C. A. Steed, D. Williams, and M. Wehner, "Parcat: Parallel climate anlysis toolkit," *Procedia Computer Science: Proceedings of the International Conference on Computational Science, ICCS 2013*, vol. 18, pp. 2367–2375, 2013, doi:10.1016/j.procs.2013.05.408.

[36] D. N. Williams and et. al., "Ultrascale visualization of climate data," *Computer*, vol. 46, no. 9, pp. 68–76, September 2013.

[37] D. Williams *et al.*, "Ultrascale visualization climate data analysis tools: Three-year comprehensive report," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-643624, September 2013.

[38] "Ultrascale visualizaiton climate data analysis tools," http://uvcdat.llnl.gov/.

[39] Prabhat, O. Ruebel, S. Byna, K. Wu, F. Li, M. Wehner, and W. Bethel, "Teca: A parallel toolkit for extreme climate analysis," *Procedia Computer Science: Proceedings of the International Conference on Computational Science, ICCS 2012*, vol. 9, pp. 866–876, 2012, doi: 10.1016/j.procs.2012.04.093.

[40] Max Planck Institute for Meteorology, "CDO: Climate Data Operators," https://code.zmaw.de/projects/cdo.

[41] Pacific Northwest National Laboratory, "Pagoda: Parallel Analysis of Geoscience Data," https://svn.pnl.gov/gcrm/wiki/Pagoda.