Efficient Extraction of Regional Subsets from Massive Climate Datasets

Jeff Daily*, Karen Schuchardt* and Bruce Palmer*
* Pacific Northwest National Laboratory
jeff.daily,karen.schuchardt,bruce.palmer@pnl.gov

***Abstract*—**
The size of datasets produced by current climate models is increasing rapidly to the scale of petabytes. To handle data at this scale parallel analysis tools are required, however the majority of climate analysis software remain at the scale of workstations. Further, many climate analysis tools adequately process regularly gridded data but lack sufficient features when handling unstructured grids. This paper presents a data-parallel subsetter capable of correctly handling unstructured grids while scaling to over 2000 cores. The approach is based on the partitioned global address space (PGAS) parallel programming model and one-sided communication. The paper demonstrates that IO remains the single greatest bottleneck for this domain of applications and that parallel analysis of climate data succeeds in practice.

I. Introduction

Parallel programming is needed to analyze the size of output data produced by today's climate models [1]. A single snapshot of a Global Cloud Resolving Model at 4Km resolution will produce terabytes of data [2]; the analysis of even a modest time series of this data will quickly overwhelm today's software and traditional climate analysis systems. For these data sizes, I/O bandwidth represents the single greatest bottleneck for analysis tools. Parallel software leveraging parallel file systems must be used to process this data, however current climate analysis tools are at most task parallel and rely on a single data reader [3]–[5].

Many climate analysis tools robustly handle the manipulation and display of regularly gridded data. However, these same applications lack sufficient features when handling unstructured or irregular grids such as the geodesic [6] or cubed sphere [7] grids. Unstructured grids are gaining popularity, further widening the gap between current software and these types of models. For unstructured grids it is necessary to provide more information about the topology of the grid and maintain the integrity of this topology information in the face of data culling.

Regular grids allow for the topology to be implicitly defined by how the data is stored; coordinate variables are generally monotonic and cell neighbors are adjacent both logically and in memory. These assumptions allow for operations over regular grids which are otherwise more difficult to perform over unstructured grids. In the case of partitioning these grids for data parallel processing, unstructured grids will often have more of the logically adjacent cells scattered across memory partitions than in the regular case.

Subsetting is a fundamental capability for any analysis tool and allows users to operate over the regions of the data in which they are interested. The subsetting operation is useful as part of a larger operation over the data, such as for obtaining regional averages, but is also useful to post-process data into a new dataset such that the cost of subsetting can be amortized across future operations over the same region. Further, as the size of datasets grow subsetting is important to reduce the data to a size that traditional analysis tools are capable of handling.

In this paper, we present a parallel tool for subsetting very large geodesic climate data in parallel while preserving the explicit topology. The code is built using the Global Arrays (GA) toolkit [8] which provides an efficient and portable "shared-memory" programming interface for distributed-memory computers and features truly one-sided communications. GA traditionally deals with dense arrays, however its sparse data operations as well as its one-sided operations allow for efficient subsetting over unstructured grids.

The primary contributions of the paper are:

- A parallel subsetter of geodesic data based on the partitioned global address space programming model and one-sided communications
- A novel algorithm for the maintenance of unstructured grid data
- A novel algorithm for the subset and even distribution of unstructured grid data
- Evaluation showing IO to be the greatest bottleneck in scaling these types of applications

The paper is organized as follows. Section II describes the requirements while III describes the design of the subsetter, its algorithms, and how GA's unique features were leveraged. Section IV presents our experimental design and the performance characteristics of the subsetter on nearly full-scale data set sizes of model data up to a resolution of 4Km. We present the capabilities under development as well as the capabilities we would like to see in section V. Finally, section VI presents our conclusions.

II. Requirements

The requirements for our software stem from the growing need for parallel analysis in the domain of climate science [1] but also from the use of the geodesic grid.

Wei-keng Liao

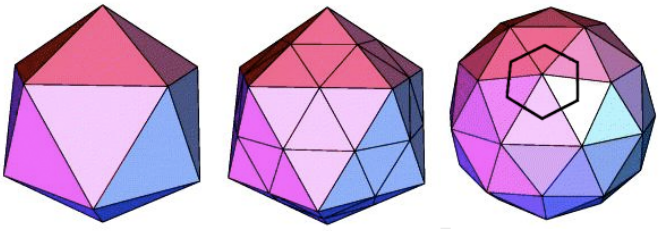

Fig. 1. The geodesic grid is created by recursively bisecting an icosahedron of 20 triangular faces and 12 vertices and projecting the resulting faces on to a unit sphere. The first recursion and a sample hexagonal cell can be seen here.

A. Geodesic Grid

Until recently, climate and weather models have primarily been simulated on structured grids that divide the latitude and longitude axes in even increments, resulting in logically structured simulation grids. Standard conventions for describing this data in the NetCDF data model have been formalized by the Climate and Forecast (CF) conventions [9]. CF defines conventions and metadata standards that enable both human and computer interpretation of the data. Human interpretation is supported through the use of standard names while definition of spatial and temporal properties of the data have enabled an extensive set of tools for data manipulation and display such as [5], [10], and [11]. The CF Conventions have been evolving to support many variations of structured grids including Orthographic, Polar stereographic, Transverse Mercator, and many others .

The GCRM uses a geodesic grid. The geodesic grid is created by recursively bisecting an icosahedron of 20 triangular faces and twelve vertices and projecting the resulting faces onto a unit sphere. The resulting vertices represent the centers of hexagonal grid cells with the exception of twelve pentagons (the centers of the original twelve vertices.) See Figure 1 for an example of the first stage of bisection and projection, followed by the definition of one of the hexagons. Further details can be found in [6].

From the previous description, it can be seen that the geodesic grid used by the GCRM is fairly regular. However, the horizontal dimension has some important properties in common with unstructured grids: the grid coordinates are not monotonic and simple conventions are not available for identifying the neighbors of all cells. As a consequence, it is necessary provide more information about the topology of the grid. Other unstructured grids such as triangular, cubed sphere [7], and arbitrary unstructured polygons are also being applied to various models. There is a recognized need to extend the CF conventions to unstructured grids so that general data analysis, regridding, and display tools can be developed. As yet, no such standard exists. Lacking a standard, development of general purpose tools has been slowed. However some preliminary tools and approaches have progressed by focusing on the in-memory operations and abstracting the data loading from the processing [12].

Although the geodesic grid is fairly structured, we choose to represent it in an unstructured way. Each of the grid's cells, corners, and edges are uniquely indexed from zero. For a given positive integer R, there are $N = 10 \times 2^{2R} + 2$ cells, $C = (N-2)\times 2$ corners, and $E = (N-2)\times 3$ edges. Increasing the value of R increases the resolution of the model. For example, a value of $R = 10$ is approximately 8Km while $R = 11$ is approximately 4Km. The number of cells, corners, and edges are represented as dimensions within a NetCDF file.

The horizontal topology describes the connectivity relationships between cells, nodes, and edges, all of which may have associated 3D data. The topology consists of three primary arrays: a mapping between cells and cell corners, a mapping between cells and cell edges, and a mapping between edges and corners. Because neighbor lists are important for visualization programs but difficult to generate in the general case, they are included as part of the topology as the `cell_neighbors(cells,neighbors=6)` variable. The full list of topology variables include:

- `cell_neighbors(cells,neighbors=6)`,
- `cell_corners(cells,cellcorners=6)`,
- `cell_edges(cells,celledges=6)`, and
- `edge_corners(edges,edgecorners=2)`

The vast majority of cells are hexagons which is why most of the last dimensions are of size six except for the obvious case of `edgecorners`. For the twelve pentagons, the sixth value in these arrays are repeated but could just have easily been set to a negative index and interpreted appropriately.

The horizontal geometry describes the longitude and latitude location of each object. That is, there is one latitude and one longitude array for each of the topology objects (cell centers, corners, and edges).

The vertical grid consists of layers sandwiched between interfaces with the number of layers equal to the number of interfaces minus 1. Because grid variables are associated with both locations, layers and interfaces are defined as dimensions. They can be thought of as two distinct vertical grids from a representation standpoint.

The data model is designed to support efficient model output, fully describe the grid topology, and provide sufficient information for tessellation to triangles for 3D visualization. The approach taken with the model is to adopt the CF conventions to the extent possible and adopt early ideas circulating within the community. However, as many of the details have not been decided, custom data analysis tools are currently required.

B. Data Parallelism

Larson, Ong, and Tokarz note that the current popular climate data analysis packages remain single-processor applications which lack the memory required to handle large data volumes as well as the processing power to analyse the data in a timely fashion [1]. Although they emphasize using OpenMP as a first step toward parallism, we instead emphasize using a distributed data model first. Well designed communication libraries such as the Message Passing Interface (MPI) [13] may

already take advantage of shared-memory parallelism within a compute node or multi-core desktop computer.

The data parallelism offered by libraries such as MPI or Global Arrays is absolutely necessary to handle the size of data of modern climate models. An edge data variable of the geodesic grid at an approximate resolution of 4Km and 100 levels is nearly $10 \times 2^{2R} \times 3 \times 100 \times 4\,\text{bytes} \approx 50\,\text{gigabytes}$ in size. Even a modest number of these variables will surpass the memory available in most desktop systems and even some small clusters.

C. Fast IO

For data of this size, efficiently reading from and writing to disk requires the use of parallel IO libraries such as Parallel NetCDF [14] or the HDF5/NetCDF4 [15] [16], both of which are in turn built on top of the MPI-IO libraries [17]. Currently, GCRM output is stored in netCDF [16] files, a format for storing array-oriented machine-independent data.

D. Dataset Abstraction

Model output is often distributed across many files for a given model run. There are any number of schemes for organizing so many files e.g. one variable per file with multiple timesteps per file, separating out the grid into a separate file, one timestep per file with multiple variables. The reconstitution of these files into a logical set of variables and metadata is an established practice [18], [19]. We emphasize that the aggregation of files into an abstract dataset is required in order to operate on the data itself. Operations on a dataset are more intuitive than needing to know the addling details of which files hold which variables.

E. Maintenance of Topology Variables

Regular grids such as the cartesian, rectilinear, or curvilinear lend themselves to representations as multidimensional arrays such that logically adjacent cells are either adjacent in memory or can be located via a shape-based index calculation. Although some attempt is made to keep logically adjacent cells nearby in memory, geodesic grids do not have the luxury of using relatively simple shape-based index arithmetic to locate neighbors.

The topology variables mentioned in II-A are not unique to our grid; any grid could be described using a similar set of variables. However, since topology is often implicitly defined by other grids, these variables are not correctly handled by current software. When a subset occurs, the indices of these variables must be updated to reflect the remaining corners, edges, and cells.

F. Maintain Integrity of Entire Grid Cells

Howe and Maier detail the properties of well-formed grids in [12]. Proper subsets should maintain the same well-formed properties of the original in order to remain useful to further analysis. Therefore, the cell and its surrounding corners and edges must remain intact during a subset.

III. Design

In this section we describe the design of our classes and algorithms based on the requirements established in II.

A. How to Run the Subsetter

The success of the NetCDF Operators [5] and similar tools demonstrate the need for user-ready applications for the analysis of their data. The success of tools such as CDAT [3] validate the need for a scriptable interface and customization of basic and advanced operators. We plan to provide both the scriptable interface as well as a set of predefined command-line tools.

The subsetter is the first in a series of planned parallel command-line tools based on unstructured grids, the PGAS model, and one-sided communication. It takes arguments specifying specific variables (-v) or dimension ranges (-d) to extract, or at a higher level a latitude and longitude bounding box (-b). In this way it is most akin to the NetCDF Operators' "kitchen sink" application. Besides the particular use of "mpiexec -np" below to invoke our MPI program with a given number of processors, example usage of the subsetter looks like:

- mpiexec -np 128 subsetter -b 20,-20,160,90 -v vorticity january.nc february.nc MJO_vorticity_janfeb.nc
- mpiexec -np 64 subsetter -b 90,0,180,-180 -d levels,1,5 geopotential.nc

B. Dataset Abstraction

The subsetter minimally supports two forms of input file aggregation, either across a specified dimension e.g. time or by taking the union of all input files such that duplicate dimensions and grid and topology variables within later files are ignored. These forms of aggregation are modeled after what is available when using NetCDF Markup Language [18]. NcML input is not directly supported at this time but is planned for a future release.

C. Parallel IO Abstraction

IO operations are hidden behind abstract base classes. Any IO library can be supported. This is similar to how the Java NetCDF library works [20]. Further, differing IO strategies using the same IO library can also be developed behind the same API. The use of Parallel NetCDF was selected because of the ubiquity of the NetCDF libraries and data format in climate applications.

D. The Global Arrays Library

The PGAS programming model assumes a global address space which is partitioned such that each process is associated with a local portion of the space. One-sided communication allows a process to access another process's address space without any explicit participation by the latter process. Such communication can reduce synchronization, reduce data movement, and can simplify programming. The Global Arrays (GA) library supports both models.

The subsetter was built using the GA library for the wealth of features it provides which are tailored to our problem domain. GA provides a distributed dense multidimensional array programming abstraction and the data we will be operating over is stored as dense arrays within NetCDF files. It should be noted that dense distributed arrays would also work well for regularly gridded data. However, due to the use of unstructured grid data, the algorithm for subsetting the data will look quite different than for the structured case. Recall that for unstructured grids, logically adjacent cells are not necessarily adjacent in memory. In order to evenly distribute a subset, a single process will need to send a varying amount of data to any number of other processes. Certainly a collective operation could be considered, but GA provides the necessary functionality without needing any explicit cooperation from any other process. Any given process will simply put the section of the subset into the remote process's memory.

There are certain GA one-sided operations which are tailored for use on one-dimensional arrays which are prevalent within our program. These operations include:

- `GA_Patch_enum`,
- `GA_Scan_add`,
- `GA_Scan_copy`,
- `GA_Pack`, and
- `GA_Unpack`

Those operations have been demonstrated in the computation of sparse matrix multiplication [8] but are equally useful in the manipulation of unstructured grids. The remaining GA operations are N-dimensional and include:

- `NGA_Scatter`,
- `NGA_Gather`,
- `NGA_Put`, and
- `NGA_Get`

Those operations are useful for redistributing the subset data and for querying the values of the various distributed arrays without regard to which process owns the data being queried.

E. The Algorithms

The one-sided communication and PGAS model supported by GA allowed us to develop some novel algorithms for the manipulation of unstructured grids. In this section we diagram and describe the algorithms we developed. The vast majority of functionality within the subsetter is provided by either PnetCDF or GA. GA allocates and evenly distributes the arrays which are then filled with data by PnetCDF. GA operations are then used to prepare the data for packing, at which point a custom n-dimensional packing routine is used. The packed, evenly-distributed data is then written back to disk using PnetCDF. Of these algorithms, the novel ones include reindexing the masks, reindexing the topology variables, and the n-dimensional pack routine.

Each dimension of the data has two arrays associated with it, a bitmask and an integer array representing the new indices of the dimension in case of a subset. For instance, if any of the bits are turned off, the corresponding indices of the index array will have negative values. The remaining values of the index array will increase monotonically, skipping the negative or masked indices. The bitmasks are generated based on a rectangular latitude and longitude region specified on the command-line, or by specifying one or more indices of a dimension to select. Although a rectangular region is currently used for simplicity, once translated the bitmasks allow for arbitrary subsets to be defined. These bitmasks are then used to evenly distribute the resultant subset across all processes. Note that these bitmask and associated index arrays are one-dimensional and distributed.

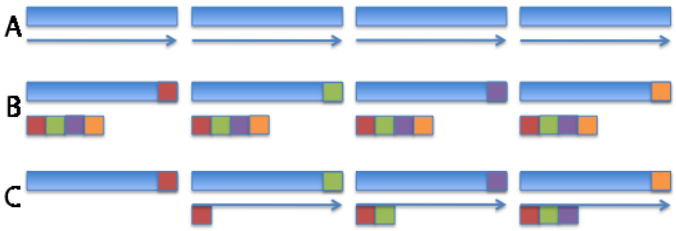

Fig. 2. A distributed partial sum over a 1-D array (in blue). A partial sum is first computed over the local portions (A), the last values of each portion are collected on each process (B), and lastly each local portion adds the last values taken from each previous process (C).

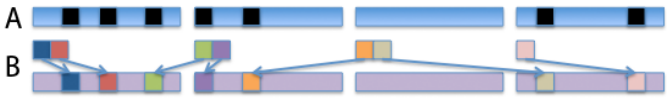

Fig. 3. Reindexing of a Dimension Index

1) Partial Sum: A partial sum of the index array associated with each dimension is useful for later determining where subset data is to be placed. That feature will be explained in more detail in III-E4. The partial sum operation here is semantically similar to the one found in the C++ STL [21]. It computes a series of sums over an array from the first element through the *i*th element and stores the result of each such sum in the *i*th element of a destination array.

The partial sum is computed by first performing partial sums of each local portion of the source array. This operation is represented by Figure 2A. The last value of each local sum is then collectively distributed to each process as seen in Figure 2B. Figure 2C is the last step where each local portion adds the last values of each process's sum which come before.

2) Reindexing of Dimension Index: Creating the index array associated with a mask requires three specific GA operations, `GA_Fill`, `GA_Patch_enum` and `GA_Unpack`. The mask array is represented in Figure 3A. `GA_Fill` fills the index array, the bottom array in Figure 3B, with a value of -1. A second array is created after each process counts how many masked bits are present and collectively sums to get the size of the array to create. `GA_Patch_enum` enumerates the values in the second array starting from zero with an increment of 1. The enumerated array is shown in Figure 3B. `GA_Unpack` expands the enumerated array values into the filled array based on the associated mask array seen as Figure 3B.

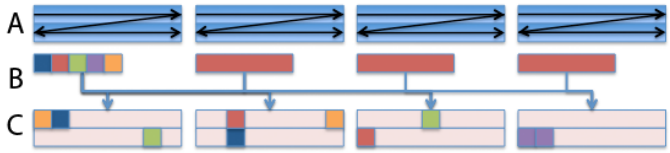

Fig. 4. Reindexing of Topology Variables

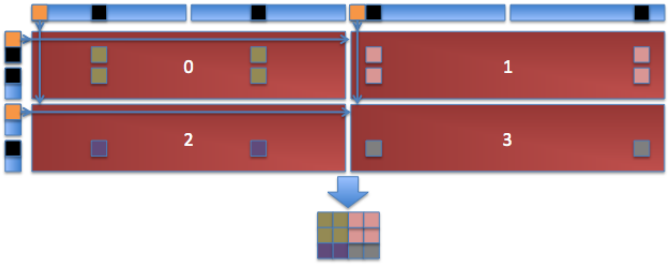

Fig. 5. Pack

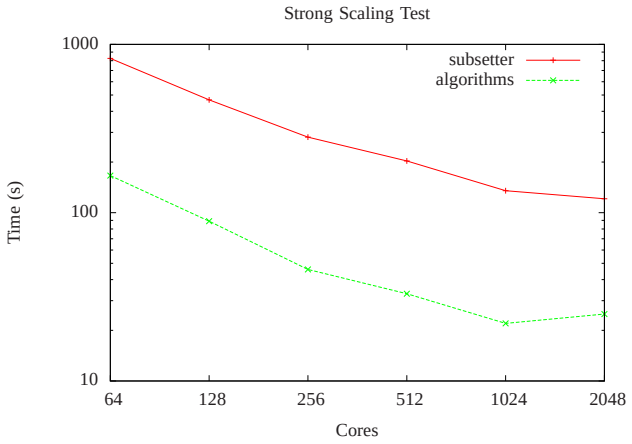

Fig. 6. Strong Scaling Test

3) Reindexing of Topology Variables: Recall that the topology variables are those which map from one index to one or more other indices such as from a cell index to each of its corner indices. A typical subset operation reduces the number of cells, corners, and edges within the grid, so it is important to maintain the integrity of these mapping arrays such that they map to real indices.

The reindexing of the topology variables relies on the recalculated index array of the associated domain. For example, when reindexing the mapping from edges to corners, the recalculated corners index array is required. The mapping values represent indices into the recalculated index array. The mapping arrays are iterated over to gather the required indices for the subsequent GA routine `NGA_Gather` to query, represented in Figure 4A. The `NGA_Gather` routine gathers array elements from a global array into a local array. In this each process gathers the new values for the mapping from the index array (Figure 4B) and then appropriately replaces the old mapping values in Figure 4C.

4) N-Dimensional Pack: The goal of the pack routine is to start with an evenly distributed source array, subset it, and evenly distribute the subset. The subset is specified using mask arrays, one for each dimension of the source array. The problem lies with where to put the data once each process computes its portion of the subset; each process would need to know a priori the size of the subsets that logically come before their own computed portion and `NGA_Put()` their data after then.

The partial sum of the mask array is a convenient solution to this problem. Each process owns a portion of the source array in terms of the global address space of the array. For each dimension, the value in the corresponding partial sum array at the index represented by the lowest index owned by each local portion describes where to put the subset data.

Figure 5 should clarify this. In the figure, the blue arrays represent both the mask arrays as well as the partial sum arrays. The masked bits are indicated in black. Each of the four processes query the partial sum arrays using `NGA_Get()` at the corresponding locations indicated in orange and the small blue arrows. Now knowing where to put their subset data, each process can `NGA_Put()` their portions resulting in the subset array. Because the destination array was created using GA, it is by definition evenly distributed. The `NGA_Get()` and `NGA_Put()` operations do not require explicit knowledge of where the data lives, reducing the burder of the programmer.

IV. Evaluation

In order to evaluate the scalability of our subsetter we performed strong scaling tests. All tests were performed on the franklin Cray-XT4 supercomputer [22] located at NERSC [23]. The performance data collected was generated by custom wrappers to the functions we developed. The IO wrappers perform barriers before and after each collective IO operation is called with the start and end timestamps collected immediately after each barrier using `clock_gettime()` or `gettimeofday()` depending on platform availability. Before program termination, the number of bytes written and read and the total time spent performing IO is collected on the zeroth process and displayed in terms of GB/s. The profiling information collects the number of times each function is called and how much time was spent in each function. It is only reported for the zeroth process and is meant only as a general measure of performance. The IO and function profile data were collected on separate runs so that the collection of one would not affect the other.

A. Strong Scaling

This test was run over 24 timesteps of an edge variable at a 4Km resolution (R=11) specifying a subset region of the Madden-Julien Oscillation [24] (20N,-20S,160E,90E). One timestep of the edge variable is 12.1875 GB. Figure 6 shows the timing results of the test while Figure 7 shows the IO bandwidth. The number of processors was doubled each run starting from 64.

TABLE I
STRONG SCALING PROFILE FOR PROCESS 0

Name	Calls	Prcnt	Times	Prcnt	T/call	Prcnt
NetcdfVariable::read()	39	0.1	47710000	0.79	1223333.3	79.5
NetcdfFileWriter::write(int,int,int)	34	0.1	8460000	0.14	248823.5	14.1
ConnectivityVariable::reindex()	3	0	1480000	0.02	493333.3	2.5
VariableDecorator::read()	5	0	780000	0.01	156000	1.3
NetcdfDataset::NetcdfDataset(string)	5	0	670000	0.01	134000	1.1
Dataset::adjust_masks(LatLonBox)	1	0	600000	0.01	600000	1

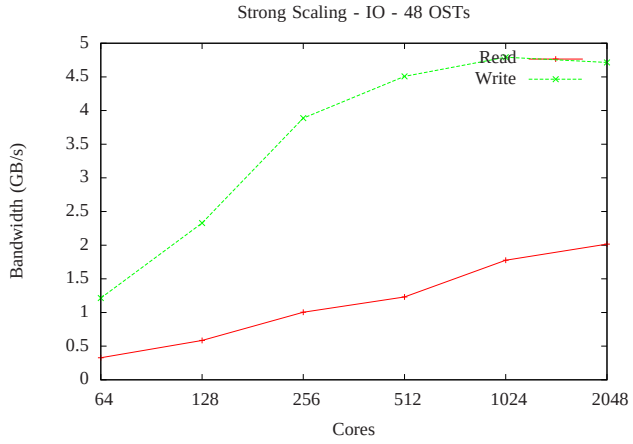

Fig. 7. Strong Scaling Test - IO

Table I is a table of a profile for process zero for the 1024 process run. The function names are sufficiently self-documenting for those familiar with C++ syntax. The zeroth process represents a worst-case scenario since the default distribution of data using GA will always put data on at least the zeroth process implying process zero should have at least as much work to perform as any other processor. The profile for the 1024 run was selected arbitrarily, however the profiles for the other strong scaling runs demonstrated a similar trend.

Although initially promsing at scales of over 2000 cores, it is not entirely clear that the algorithms we have developed are sufficient due to the large mismatch between IO and everything else. The profile suggests that IO accounts for nearly 85% of program execution. This is not a stretch of the imagination since our software merely reads, subsets, and writes. Even if our code performed a significant calculation after each read of the edge variable, the profile would likely remain IO bound but less so.

The reason for the disproportionately faster write bandwidth versus read bandwith is likely due to caching by the Lustre filesystem. The relatively small amount of data being written is likely being copied to an internal buffer allowing the functions to return rather quickly. Furthermore, there is ongoing work to allow the subsetter program to read in a less substantial amount of data based on the desired subset. As it is now, the subsetter reads the entire horizontal domain and immediately culls what falls outside of the specified region. For example, the MJO region is roughly 6.5% of the global data. An idea worth further research would be to implement a collective read routine similar to the one-sided `NGA_Gather` provided by GA.

V. FUTURE WORK

We are currently developing a general C++ API for climate data analysis in a data parallel fashion based on the PGAS model and one-sided communications. The API will be leveraged to produce additional command-line tools, however it is intended primarily to be used by climate scientists to produce the kinds of tailored analyses which they require. Time permitting, the API will be exposed to the Python language in order to facilitate ease of use in a scripted environment. Larson, Ong, and Tokarz also point out certain capabilities missing from popular climate data analysis packages such as probability density function (PDF) estimation as well as the sorting and ordering of data.

The evaluation performed in IV revealed that IO for our application remains the greatest bottleneck. This fact is exacerbated when small regions representing a mere fraction of the entire grid are subset. Our current strategy is terribly inefficient because it reads in entire variables only to immediately cull the majority of them. If a masked read were available within the Parallel NetCDF library we would likely see performance gains. The algorithms presented in this paper which evenly distribute the subset data will hopefully encourage work in this area.

VI. CONCLUSION

We developed a novel data parallel subsetter of climate data based on unstructured grids and the PGAS model. The experimental evaluation showed scalability to thousands of processors and acceptable IO bandwidth.

ACKNOWLEDGMENT

TODO

REFERENCES

[1] J. W. Larson, E. T. Ong, and C. Tokarz, “The spheroidal analysis library and toolkit: Tools for climate model output analysis,” in *MODSIM 2007 International Congress on Modelling and Simulation*. Modelling and Simulation Society of Australia and New Zealand Inc., December 2007, pp. 2974–2980.

[2] D. A. Randall, “Design and testing of a global could-resolving model,” February 2009, midterm Progress Report to the U.S. Department of Energy’s Program for Scientific Discovery through Advanced Computing (SciDAC).

[3] TODO, “Climate data analysis tools,” TODO TODO, tODO.

[4] ——, “Climate data operators,” TODO TODO, tODO.

[5] C. Zender, “The netcdf operators (nco),” http://nco.sourceforge.net/.

[6] D. A. Randall and R. P. Heikes, “Spherical geodesic grids: A new approach to modeling the climate,” http://kiwi.atmos.colostate.edu/BUGS/geodesic/.

[7] A. Adcroft, J.-M. Campin, C. Hill, and J. Marshall, "Implementation of an atmosphere-ocean general circulation model on the expanded spherical cube," *Monthly Weather Review*, vol. 132, no. 12, pp. 2845–2863, December 2004.
[8] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra, "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, 2006. [Online]. Available: http://hpc.sagepub.com/cgi/content/abstract/20/2/203
[9] B. Eaton, J. Gregory, B. Drach, K. Taylor, S. Hankin, J. Caron, R. Signell, P. Bentley, and G. Rappa, "Netcdf climate and forecast (cf) metadata conventions: Version 1.4," http://cf-pcmdi.llnl.gov/documents/cf-conventions/1.4/cf-conventions.html, February 2009.
[10] TODO, "Ferret," TODO.
[11] ——, "Ferret," TODO.
[12] B. Howe and D. Maier, "Algebraic manipulation of scientific datasets," in *In VLDB*, 2004, pp. 924–935.
[13] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, September 1996.
[14] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netcdf: A high-performance scientific i/o interface," in *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 39.
[15] TODO, "Todo," TODO.
[16] R. K. Rew and G. P. Davic, "Netcdf: An interface for scientific data access," *IEEE Computer Graphics and Applications*, vol. 10, no. 4, pp. 76–82, July 1990.
[17] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in romio," in *Proceedings of the 7th Symposium on Frontiers of Massively Parallel Computation*. Washington, DC, USA: IEEE Computer Society, February 1999, pp. 82–189.
[18] S. Nativi, J. Caron, E. Davis, and B. Domenico, "Design and implementation of netcdf markup language (ncml) and its gml-based extension (ncml-gml)," *Computers and Geosciences*, vol. 31, no. 9, pp. 1104–1118, November 2005.
[19] TODO, "Todo," TODO.
[20] ——, "Java netcdf," http://www.unidata.ucar.edu/software/netcdf-java/.
[21] D. R. Musser, G. J. Derge, and A. Saini, *STL tutorial and reference guide, second edition: C++ programming with the standard template library*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.
[22] TODO, "franklin," http://www.nersc.gov/nusers/systems/franklin/.
[23] ——, "Nersc," http://www.nersc.gov/.
[24] ——, "Mjo," TODO.