Efficient Extraction of Regional Subsets from Massive Climate Datasets using Parallel IO

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 is serial and remains at the scale of workstations. Further, many climate analysis tools are designed to process regularly gridded data but lack sufficient features to handle 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 parallel analysis of climate data succeeds in practice, although IO remains the single greatest bottleneck.

I. Introduction

Parallel programming is needed to analyze the size of output data produced by today's climate models [1], [2]. A single snapshot of a Global Cloud Resolving Model (GCRM) at 4 kilometer resolution will produce terabytes of data [3]; 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, IO 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 [4]–[6].

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 [7] or cubed sphere [8] 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 to 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 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 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 climate data generated on a geodesic grid while preserving the explicit topology. The code is built using the Global Arrays (GA) toolkit [9] which provides an efficient and portable "shared-memory" programming interface for distributed-memory computers and features truly one-sided communication. GA traditionally represents dense arrays, however its sparse data operations over one-dimensional arrays 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 (PGAS) programming model and one-sided communication
- Novel algorithms for the maintenance of unstructured grid topology data
- A novel algorithm for the subset and even distribution of unstructured grid data
- An 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 Section 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 dataset sizes of model data up to a resolution of 4 kilometers. 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.

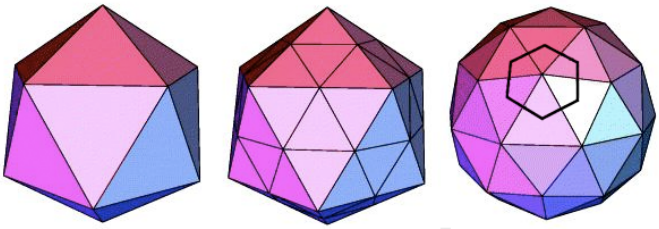

Fig. 1. The geodesic grid is created by recursively bisecting an icosahedron of 20 triangular faces and 12 vertices and projecting the resulting faces on to a unit sphere. The first recursion and a sample hexagonal cell can be seen here.

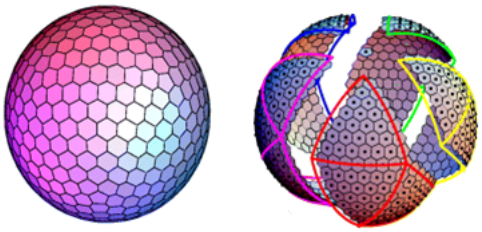

Fig. 2. Once the desired grid resolution is reached (left), if the original triangles of the icosahedron are paired such that each pair shares an edge, then the grid points generated from each pair form a logically structured block (right). These blocks can be stored as a set of regular square arrays of data points.

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 [10]. 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 the definition of spatial and temporal properties of the data have enabled an extensive set of tools for data manipulation and display such as [6], [11], and [12]. 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. These vertices represent the centers of hexagonal grid cells with the exception of twelve pentagons (the centers of the original twelve vertices.) See Fig. 1 for an example of the first stage of bisection and projection, followed by the definition of one of the hexagons. Once the desired grid resolution is reached (Fig. 2(left)), the original triangles of the icosahedron are paired such that each pair shares an edge. The grid points generated from each pair form a logically structured block (Fig. 2(right)) that can be stored as a set of regular square arrays of data points. Further details can be found in [7].

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 to provide more information about the topology of the grid. Other unstructured grids such as triangular, cubed sphere [8], 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 [13].

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 8 kilometers while $R = 11$ is approximately 4 kilometers. The number of cells, corners, and edges are represented as dimensions within a NetCDF file.

The data for the GCRM is ordered using a space-filling curve. As seen in Fig. 2, the data is divided into square panels. The data within each panel can be reorganized using a Morton-ordering scheme (also referred to as Z-ordering). This has the advantage that it is much easier to guarantee that each read or write represents a contiguous chunk in the file. The Morton-ordering scheme works very well for square arrays that are an integral power of 2 in dimension and is illustrated schematically in Fig. 3. Each array element is indexed by successive locations in the self-similar space-filling curve. Fig. 3 shows a panel that has been divided up into 16 blocks. Note, however, that if the panel had been divided into only 4 blocks, the points within each block would still lie along a continuous segment of the space-filling curve.

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)`

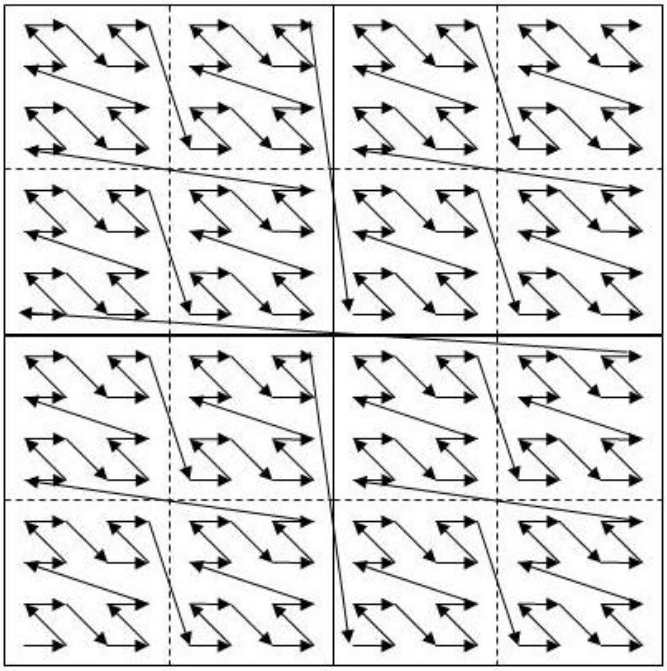

Fig. 3. Illustration of Morton-ordering curve for a 16x16 array of points. Each array element is indexed by successive locations in the self-similar space-filling curve.

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 and further modifications to the tools may be needed to conform to evolving standards.

B. Data Parallelism

Larson, Ong, and Tokarz note that the current popular climate data analysis packages remain single-processor applications. These lack the memory required to handle large data volumes as well as the processing power to analyze the data in a timely fashion [1]. Although they emphasize using OpenMP as a first step toward parallelism, we instead emphasize using a distributed data model first. Well designed communication libraries such as the Message Passing Interface (MPI) [14] 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 4 kilometers and 100 levels is nearly $10 \times 2^{2\times R} \times 3 \times 100 \times 4\,\mathrm{bytes} \approx 50\,\mathrm{gigabytes}$ in size, where $R = 11$. 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 [15] or HDF5/NetCDF4 [16] [17], both of which are in turn built on top of the MPI-IO libraries [18]. Currently, GCRM output is stored in netCDF [17] files, a format for storing array-oriented machine-independent data.

D. 64bit Libraries

64bit libraries are also required for the GCRM output at 4 kilometer resolution. At this resolution, the GCRM output already requires 64bit file offsets. At resolutions smaller than 4 kilometers, the GCRM output will begin to produce edge variables that exceed the 4 byte limit for array indexing.

E. 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 [19], [20]. 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.

F. Maintenance of Topology Variables

Regular grids such as the Cartesian, rectilinear, or curvilinear grids 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 Section 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 for other grids, these variables are not

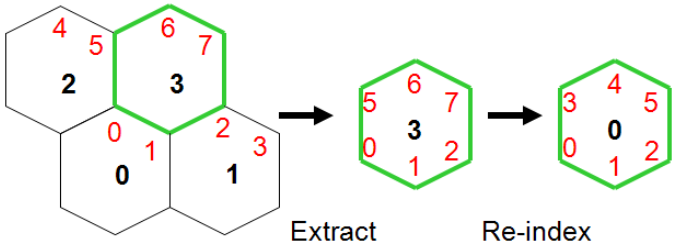

Fig. 4. Maintenance of Topology Variables and Integrity of Entire Grid Cells. The hexagonal cells remain well-formed during extraction while the indices of the corners are reindexed to reflect the fewer overall corners in the subset.

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 as seen in Fig. 4.

G. Maintaining Integrity of Entire Grid Cells

Howe and Maier detail the properties of well-formed grids in [13]. 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 as seen in Fig. 4. The first step in forming a subset results in a collection of cell and corner indices that do not form a contiguous sequence of integers. The cells and corners must be reindexed and the corresponding topology lists updated to give a well formed description of the topology.

III. Design

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

A. Subsetter User Interface

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 indicating specific variables (-v), dimension ranges (-d), or a latitude and longitude bounding box (-b) to extract. Parallel software requires the use of common startup programs (such as ”mpirun” or ”mpiexec” for MPI programs) capable of spawning parallel jobs on a cluster or some other parallel architecture. Users of serial climate analysis tools must become familiar with how to run parallel software. Looking past 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 out.nc

Similarly, the NetCDF Operator’s *ncks* application [6] is run like the following, noting that *ncks* does not aggregate input files:

- ncks -X 90,160,-20,20 -v vorticity january.nc MJO_vorticity_january.nc
- ncks -X -180,180,0,90 -d levels,1,5 geopotential.nc out.nc

B. Dataset Abstraction

A dataset abstraction is essential for the comprehension of these large datasets, hiding the details of which files contain which data. The subsetter currently 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 [19]. 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 so long as the data structures conform to the Common Data Model (CDM) [21]. This is similar to how the Java NetCDF library works, supporting many different data formats conforming to the CDM [22]. 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 and can simplify programming. The Global Arrays (GA) library supports both global address spaces and one-sided communication.

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 which owns the subset.

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

- `GA_Patch_enum`,

- `GA_Scan_add`, and
- `GA_Unpack`

Those operations have been demonstrated in the computation of sparse matrix multiplication [9] but are equally useful in the manipulation of unstructured grids. The remaining GA operations used 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 models 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 Parallel-NetCDF or GA. GA allocates and evenly distributes the arrays. The starting and ending indices owned by each process are used directly to fill the arrays using Parallel-NetCDF. GA operations are then used to prepare the data for packing, at which point a custom n-dimensional packing routine is used. After packing, evenly-distributed data is written back to disk using Parallel-NetCDF. (Note that instead of immediately writing the data back to disk, any number of other mathematical operations could take place.) 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, an integer array representing 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 zero (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 the bitmasks are currently created using a relatively simple latitude-longitude bounding box, arbitrary bitmasks could be created and fed into the subsetter without modification of the remaining algorithms. This would allow for data sets based on feature extraction (e.g. cyclones) or continental datasets. 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.

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 Section III-E4. The partial sum operation here is semantically similar to the one found in the C++ STL [23]. 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. For example, if x represents an element in the source array and y represents an element in the destination array, the ys can be calculated as:

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= x_0 + x_1 \\ y_2 &= x_0 + x_1 + x_2 \\ y_3 &= x_0 + x_1 + x_2 + x_3 \\ &\dots \end{aligned} \tag{1}$$

The partial sum is elegantly computed using a single call to the `GA_Scan_add` routine.

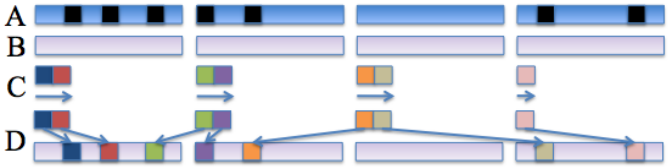

Fig. 5. Reindexing of a Dimension Index. The index array (B) is filled with a value of -1. The masked bits of (A) are tallied such that a new array (C) is created based on the tallied size. The new array C is enumerated and then unpacked into the index array (D).

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 Fig. 5A with the masked bits indicated in black. `GA_Fill` fills the index array, the array in Fig. 5B, with a value of -1. Each process counts how many masked bits they own and then collectively sums their count. A third array is created (Fig. 5C) based on this count. `GA_Patch_enum` enumerates the values in the newly created array starting from zero with an increment of 1 (indicated by the arrows in Fig. 5C). `GA_Unpack` expands the enumerated array values into the filled array based on the associated mask array as seen in Fig. 5D.

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 (Fig. 6A). The mapping values represent indices into the recalculated index array. The original mapping arrays are iterated over to prepare the required indices for the subsequent GA routine `NGA_Gather` to query. The `NGA_Gather` routine gathers array elements from a global array into a local array by specifying the desired indices. In this way each process gathers the new values for the mapping from the index array and then

Fig. 6. Reindexing of Topology Variables. The recalculated index array from the Fig. 5 reindexing operation, seen here as (A), will replace values found within the topology array (B). Each process gathers values from the index array after using the original values from (B) in an NGA_Gather call. All values from the original index array are replaced (only replacement of non-negative indices is shown for clarity).

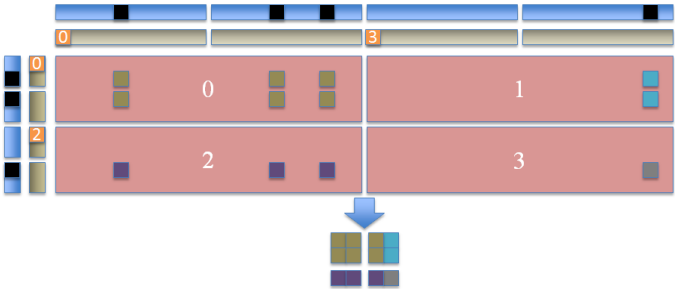

Fig. 7. Pack. The blue arrays represent the masks with masked bits indicated in black. The olive arrays are the partial sums over the masks. The orange values represent the values of the partial sum array at those indices. Each process queries the partial sum arrays for where to put their subset data, packs their local data based on the mask bits, and calls NGA_Put to deliver the subset data. The destination array is by default evenly distributed.

appropriately replaces the old mapping values. For clarity, Fig. 6B shows four processes gathering and replacing only the non-negative indices, but it should be noted that all indices are replaced by this operation.

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 blue arrays in Fig. 7 with masked bits indicated in black). The size of the subset is determined by counting the number of masked bits per dimension. Each process determines where to `NGA_Put()` its portion of the subset by examining the partial sum array (the olive arrays in Fig. 7). The values indicated in orange in Fig. 7 represent the values of the partial sum array at those particular indices and are used by each process to know where to put their portion of the subset data into the subset array. For example, process 1 in Fig. 7 calls `NGA_Put()` to place its data starting at index (3,0).

The data location abstraction offered by GA is even more important as differing numbers of processors and/or different data distributions are used. As can be seen in the case diagrammed in Fig. 7, the partial sum arrays are as evenly distributed as the data arrays. Although GA evenly distributes arrays by default, different distributions can be specified. For example, it might be beneficial to keep all vertical levels for a given region local to a process. The desired values from the partial sum arrays will change process owners depending on how many processes are used or how the data is distributed. `GA_Get` and `GA_Put` abstract away that need to track which process owns the data and simplifies the programming model.

IV. Evaluation

All tests were performed on the Franklin Cray-XT4 supercomputer [24] located at NERSC [25]. The Franklin supercomputer features 9,572 Opteron 2.3 gigahertz quad core processors with 2 gigabytes of memory per core. Its Lustre parallel file system supports a theoretical peak of 16 gigabytes/second to each of two /scratch file systems.

In order to evaluate the scalability of our subsetter we performed strong scaling tests. Data from the GCRM is not yet available, so we used data from a GCRM precursor that runs Jablonowski's test case [26]. This test case uses only 25 vertical levels, instead of the 100 or so expected for the GCRM, so data set sizes are about a quarter the size of comparable GCRM data. These data sets are still on the order gigabytes to 10s of gigabytes per variable.

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 gigabytes/second. 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 as a general measure of performance. The IO and profile data were collected on separate runs so that the collection of the former (with additional barriers) would not affect the latter. This test was run over 24 timesteps of an edge variable at a 4 kilometer resolution ($R = 11$) specifying a subset region corresponding to the Madden-Julien Oscillation [27] (20N,-20S,160E,90E). The MJO region is roughly 6.5% of the global data. One timestep of the edge variable is 12.1875 GB. The number of processors was doubled for each run, starting from 64 and going up through 2048 processors.

Two versions of the subsetter were originally tested, one to test the subsetter as a whole and the the other to test the algorithms by turning off nearly all IO. (Reading of the grid topology was still required resulting in an insubstantial amount of IO.) Turning off the IO is important in order to evaluate the scalability of our packing and reindexing algorithms. Since our current software does not involve any algebraic operations, this case is also an accurate representation of communication.

After the initial tests were performed, it was noted that the majority of data being read was discarded. An optimized version of the subsetter was developed which determines ahead of time which processes will not participate in the subset operation. This allowed for the nonparticipating processes to specify empty regions in the collective read operation, reducing the total amount of data read prior to the subset. It was important to show test results for both the original and optimized versions of the subsetter in order to illustrate the

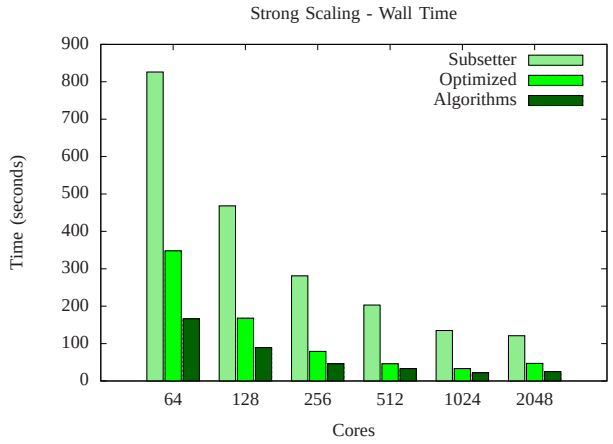

Fig. 8. Strong Scaling Test. First, the subsetter was run as a whole program (”Subsetter”). Second, the subsetter was optimized to read less data in the case of a subset (”Optimized”). Lastly, nearly all IO was stripped from the program to capture the performance of the packing routines (”Algorithms”). At smaller numbers of cores, there is a greater difference between time spent in IO and time spent everywhere else.

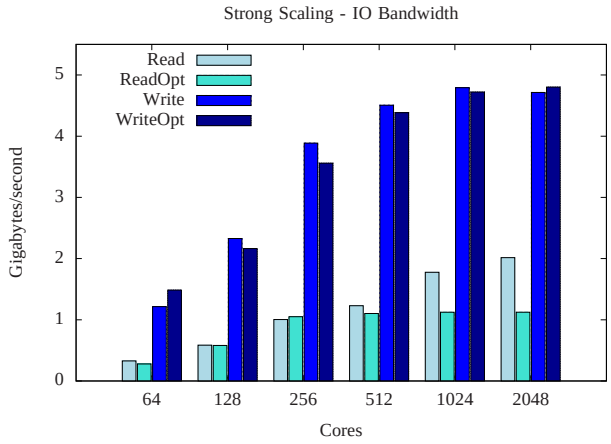

Fig. 9. Strong Scaling Test for IO Bandwidth. The Read and Write bandwidths are compared as the number of cores are increased. The subsetter and its optimized versions are both shown. In general, all bandwidths increased with respect to the number of cores. The optimized bandwidths matched those of the original subsetter given a small amount of variability. The notable exception is that of the optimized read bandwidth which leveled off near 512 cores.

need for efficient IO algorithms as well as to accurately capture the IO requirements when reading the entire horizontal region of data.

Fig. 8 shows the timing results of the tests. Three versions of the code were run. ”Subsetter” represents the original program, ”Optimized” represents our optimized read, and ”Algorithms” represents executing with nearly all IO turned off. Comparing the original subsetter to the algorithms case, it is clear that IO represents a significant portion of the execution time. As the number of cores increases, the optimized case more closely resembles the algorithms alone. The benefit of additional processors appears to have reached a plateau near 1024 processors for all cases, however this may be due to the size of the problem since run times at this scale are only a few minutes in length.

Fig. 9 shows the IO bandwidth results of the test. This test captures the scalability of the IO system which correlates with the scalability portrayed in Fig. 8. Although in the unoptimized case the read bandwidth may have benefited from additional cores beyond 2048, the write bandwidth appears to have reached a plateau near 1024 processors. The hardware is likely reaching its saturation point if not already there. The optimized bandwidths matched those of the original subsetter given a small amount of variability. The notable exception is that of the optimized read bandwidth which leveled off near 512 cores. The fact that the optimized read tracks the unoptimized read bandwidth up to 512 processors, even though the number of cores that are nominally reading data is smaller, reflects optimizations in both the Parallel-NetCDF and the MPI-IO libraries that are redistributing the reads to more processors.

Table I represents a profile for process zero for the 2048 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 2048 run was selected arbitrarily, however the profiles for the other strong scaling runs (not shown) demonstrated a similar trend. The functions which did not contribute a significant amount of time to the program were removed and the table is sorted by total time spent in the function. The number of calls to the read and write routines were small. (Not shown in Table I is the number of calls to the function comparing each latitude/longitude pair to the desired subset region which accounts for over 60% of the function calls.) However, the amount of time spent in the read and write routines was significant. In all cases, the amount of time spent in IO was at least 75% of the program’s execution. IO is clearly the dominant factor in running our code.

Table II represents a profile for process zero for the 2048 process run using the optimized subsetter. The notable differences between Table II and Table I are the total time and percentage of time spent in each function. Although the optimized subsetter spends substantially less time reading, the majority of time is still spent in IO. Although still dominant, IO is a less significant contributor to the total execution time.

A. Discussion

Running our subsetter at a scale of over 2000 cores is initially promising. Fig. 8 demonstrates that the algorithms we developed scale to at least 1024 processors. The performance of the original subsetter can be considered as the case where operations on the entire globe are performed. For these operations, the subsetter would be reading all of the available data. These global subset will stress the IO system the greatest. The optimized subsetter will not help in cases of operations on the entire globe. The significant improvement afforded by

TABLE I
PARTIAL PROFILE FOR PROCESS 0 AT 2048 CORES - MJO REGION

Name	Calls	Percent Calls	Time (seconds)	Percent Time	Time/call (seconds)
NetcdfVariable::read()	39	0.1	156.1	93.0	4.00
NetcdfFileWriter::write(int,int,int)	34	0.1	5.9	3.5	0.17
VariableDecorator::read()	5	0.0	1.7	1.0	0.34
NetcdfDataset::NetcdfDataset(string)	5	0.0	1.2	0.7	0.24
Dataset::adjust_masks(LatLonBox)	1	0.0	1.0	0.6	1.05
ConnectivityVariable::reindex()	3	0.0	0.7	0.4	0.25
NetcdfFileWriter::NetcdfFileWriter(string)	1	0.0	0.2	0.2	0.29

TABLE II
PARTIAL PROFILE FOR PROCESS 0 AT 2048 CORES - MJO REGION - OPTIMIZED

Name	Calls	Percent Calls	Time (seconds)	Percent Time	Time/call (seconds)
NetcdfVariable::read()	39	0.1	22.8	65.4	0.58
NetcdfFileWriter::write(int,int,int)	34	0.1	6.5	18.7	0.19
NetcdfDataset::NetcdfDataset(string)	5	0.0	1.2	3.4	0.24
VariableDecorator::read()	5	0.0	1.1	3.0	0.21
Dataset::adjust_masks(LatLonBox)	1	0.0	0.8	2.4	0.82
ConnectivityVariable::reindex()	3	0.0	0.8	2.2	0.26
NetcdfFileWriter::NetcdfFileWriter(string)	1	0.0	0.4	1.1	0.37

the optimized subsetter is likely due to both the data layout using a space-filling curve as explained in Section II-A and optimization of collective read operations in both the Parallel-NetCDF and MPI-IO libraries. The space-filling curve places logically adjacent cells near each other in memory allowing for the maximum number of nonparticipating processes during the read operation.

The profile of the global reads in Table I suggests that IO accounts for nearly 95% of program execution at 2048 cores. This is not a stretch of the imagination since our software reads, subsets, and then writes. Even if our code performed a significant calculation after each read of the edge variable, the profile would likely remain IO bound. At fewer numbers of cores the percentage of time spent in IO ranged from 75% to the 95% shown in Table I. Even in the optimized case, the profile remains IO bound.

The reason for the disproportionately faster write bandwidth versus read bandwith seen in Fig. 9 is probably due to caching by the Lustre parallel filesystem. The relatively small amount of data being written is most likely being copied to an internal buffer allowing the functions to return quickly.

V. FUTURE WORK

The success of the NetCDF Operators [6] and similar tools demonstrate the need for user-ready applications for the analysis of their users' data. The success of tools such as CDAT [4] 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.

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 communication. 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 Section IV revealed that IO for our application remains the greatest bottleneck. This fact is exacerbated when large regions representing the entire grid are subset. Our current optimized strategy of reading chunks of variables based on whether a process participates in the later subset still reads more data than is eventually redistributed. If a masked read or a read based on individual array index tuples similar to `NGA_Gather` were available within the Parallel-NetCDF library we might see further performance gains. Work along these lines has been suggested as part of the future development of Parallel-NetCDF [28].

VI. CONCLUSION

We developed a novel data parallel subsetter of climate data based on unstructured grids, the PGAS programming model, and one-sided communication. The experimental evaluation showed scalability to thousands of processors and acceptable IO bandwidth. IO bandwidth is the greatest bottleneck in scaling these types of applications.

ACKNOWLEDGMENT

The authors are indebted to Wei-keng Liao at Northwestern University for invaluable help in getting the subsetter running and optimized on the Franklin platform.

This research was funded by the U.S. Department of Energy's (DOE) Office of Advanced Scientific Computing Research through its Scientific Discovery through Advanced Computing program and was performed at DOE's National Energy Research Scientific Computing Center.

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] W. Washington, "Challenges in climate change science and the role of computing at the extreme scale," Presentation at DOE ASCR-BER Scientific Grand Challenges Workshop, November 2008.

[3] 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).

[4] D. N. Williams, C. M. Doutriaux, R. S. Drach, and R. B. McCoy, "The flexible climate data analysis tools (cdat) for multi-model climate simulation data." in *ICDM Workshops*, Y. Saygin, J. X. Yu, H. Kargupta, W. Wang, S. Ranka, P. S. Yu, and X. Wu, Eds. IEEE Computer Society, 2009, pp. 254–261.

[5] U. Schulzweida, L. Kornblueh, and R. Quast, "Climate data operators user's guide," December 2009, version 1.4.1.

[6] C. Zender, "The netcdf operators (nco)," http://nco.sourceforge.net/.

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

[8] 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.

[9] 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

[10] 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.

[11] P. Cornillon, J. Gallagher, and T. Sgouros, "Opendap: Accessing data in a distributed, heterogeneous environment." *Data Science Journal*, vol. 2, pp. 164–174, 2003.

[12] S. Hankin, D. E. Harrison, J. Osborne, J. Davidson, and K. Obrien, "A strategy and a tool, ferret, for closely integrated visualization and analysis." *Journal of Visualization and Computer Animation*, vol. 7, no. 3, pp. 149–157, 1996.

[13] B. Howe and D. Maier, "Algebraic manipulation of scientific datasets," in *In VLDB*, 2004, pp. 924–935.

[14] 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.

[15] 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.

[16] "Hdf5 user's guide," http://www.hdfgroup.org/HDF5/doc/UG/index.html.

[17] 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.

[18] 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.

[19] 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.

[20] B. Domenico, J. Caron, E. Davis, R. Kambic, and S. Nativi, "Thematic real-time environmental distributed data services (thredds): Incorporating interactive analysis tools into nsdl." *J. Digit. Inf.*, vol. 2, no. 4, 2002.

[21] S. Nativi, J. Caron, B. Domenico, and L. Bigagli, "Unidata's common data model mapping to the iso 19123 data model." *Earth Science Informatics*, vol. 1, no. 2, pp. 59–78, 2008.

[22] J. Caron, "Netcdf java library," http://www.unidata.ucar.edu/software/netcdf-java/.

[23] 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.

[24] "Franklin (cray xt4)," http://www.nersc.gov/nusers/systems/franklin/.

[25] "National energy research scientific computing center," http://www.nersc.gov/.

[26] C. Jablonowski and D. L. Williamson, "A baroclinic instability test case for atmospheric model dynamical cores," *Quarterly Journal of the Royal Meteorological Society*, vol. 132, no. 621C, pp. 2943–2975, May 2006.

[27] C. Vera, W. Higgins, J. Amador, T. Ambrizzi, R. Garreaud, D. Gochis, D. Gutzler, D. Lettenmaier, J. Marengo, C. R. Mechoso, J. Nogues-Paegle, P. L. Silva Dias, and C. Zhang, "Toward a unified view of the american monsoon systems," *Journal Of Climate*, vol. 19, no. 20, pp. 4977–5000, OCT 15 2006, 1st Climate Variability and Predictability Science Conference (CLIVAR), Baltimore, MD, JUN 21-25, 2004.

[28] K. Gao, W. keng Liao, A. Choudhary, R. Ross, and R. Latham, "Combining i/o operations for multiple array variables in parallel netcdf," 31 2009-sept. 4 2009, pp. 1 –10.