Analyzing and visualizing next generation climate data

Tech-X Corporation
5621 Arapahoe Ave.
Boulder, CO 80303
Alexander Pletzer, PI

SBIR II DE-FG02-08ER85153
Grant supported by DOE office of
Advanced Scientific Computing Research
Topic: 40b
Final report covering the period of August 2009 to August 2012

Contents

1 Introduction

This report summarizes work carried out since the start of Phase II of this project in August 2009 to its completion in August 2012. It includes discussion of highlights, overall progress and period goals, collaborations, and outreach.

1.1 Highlights

1.1.1 Development of a standard for representing Mosaic gridded data

Our starting point for this project was the Gridspec proposal by Balaji and Zhiang (GFDL) [1]. Prior to implementing their data layout, Gridspec had to become an accepted standard of the Climate and Forecast (CF) metadata conventions [2], which governs how climate data should be described. Tech-X Corp. played a leading role in articulating critical details of the Gridspec standard by bringing together scientists from multiple research laboratories across the country and overseas. The Gridspec proposal [3] is the result of a close collaboration between Tech-X Corp., Lawrence Livermore National Laboratory (LLNL), Pacific Marine Environmental Laboratory (PMEL), Geophysical Fluid Dynamics Laboratory (GFDL), University of Reading/UK Met Office, and University Corporation for Atmospheric Research (UCAR). The proposal was submitted to the CF conventions body in early 2011 and approved in May 2011. The Gridspec extensions enhance the present CF conventions in two major ways: (1) data can be scattered over multiple logically rectangular grids (M-SPEC) and (2) different time slices can be stored in different files (F-SPEC). Together, M-SPEC and F-SPEC allow data produced for the Coupled Model Intercomparison Project, Phase 5 (CMIP-5), to be generated on all currently known mosaic grids with time dependent and static data residing in separate files from coordinate data. Data consumers can access all the files by opening a "host" file, which acts as a single entry point to the aggregation. Examples of supported file aggregations are shown in Fig. 2.

1.1.2 Development of an open-source library (LibCF) to read, write, and analyze Mosaic data

As a natural extension to the above Gridspec standard, we wrote a library, LibCF, which implements the Gridspec standard. LibCF existed prior to this project but the size of the library has more than doubled as a result of the Gridspec implementation. The LibCF library is not restricted to Gridspec data, it supports general curvilinear data. The library is designed to allow users to access functionality at different levels; for instance it can be used for linear interpolation in arbitrary number of dimensions.

1.1.3 Participation in the UV-CDAT team effort

We have extensively worked with LLNL and collaborators to bring in state-of-the art LibCF [4] and ESMF [5] interpolation to the UV-CDAT [6] software stack, which until then supported interpolation on rectilinear meshes but not on the curvilinear meshes that are prevalent in Gridspec and other grids. This required extending the CMake [7] build system to compile the LibCF and ESMF libraries. Interoperability between ESMF and Python was

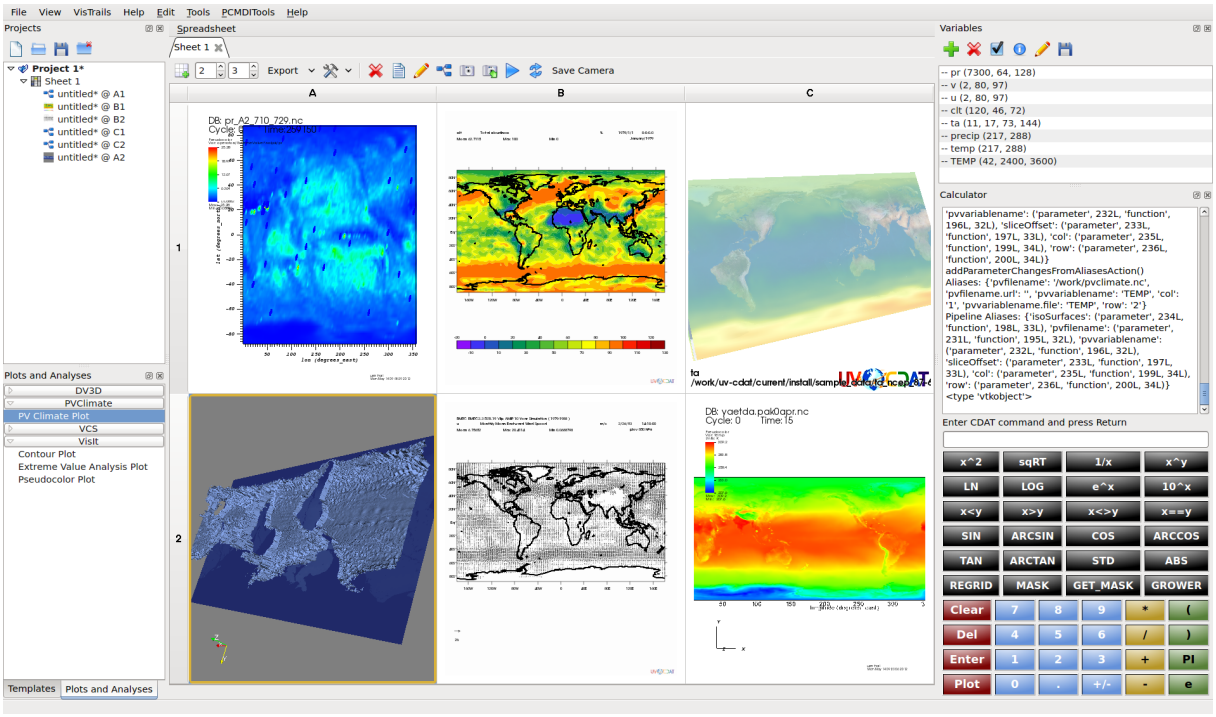

Figure 1: Using UV-CDAT to analyze and visualize climate data. Visualization is as simple as clicking on fields and dragging these into the scene.

achieved through a thin layer provided by collaborator Ryan O'Kuinghttons at NOAA. ESMF offers more interpolation choices than LibCF and also handles the problem of grid singularity by interpolating in Cartesian space. Of particular interest to researchers is the ability to apply conservative interpolation methods with ESMF. The close collaboration between NOAA and Tech-X has been extremely fruitful, yielding a more complete and robust interface to ESMF. ESMF has been completely integrated into UV-CDAT and is part of the upcoming UV-CDAT 1.2 release.

1.1.4 Parallelism and distributed arrays

Our last focus was to add MPI [8] parallelism to ESMF in UV-CDAT, which required further alteration to the build system. Interpolation, and in particular conservative interpolation, can be computationally expensive. ESMF interpolation was written from the ground up using domain decomposition and MPI as the communication engine. ESMF interpolation in UV-CDAT will automatically decompose the data across the domain in optimal fashion. The UV-CDAT interface to ESMF will perform the final MPI "gather" data call to bring the the data back to the root process. The `MPI_Init` and `MPI_Finalize` calls are invoked automatically for the user, under the hood. As only externally visible concession to the MPI library, users should launch their script using `mpiexec` command when running on multiple processors.

All elements are now in place in UV-CDAT to effectively leverage parallelism, with the possible exception of a method to write data in parallel. Postprocessing applications involving

zonal or time averages, which used to take hours, can now be concluded at a fraction of the time. The speedup naturally depends on the number of time steps, load balancing between processes, and whether some postprocessing require sequential execution. For perfectly well balanced postprocessing tasks with no sequential part, speedups will only be limited by time and/or spatial resolution.

For postprocessing that requires communication between processes, we developed a highly flexible, multi-dimensional distributed array class in Python, which relies on remote memory access calls provided by MPI-2 library. Each local array can expose an arbitrary number of "windows" to other processes. A collective "get" method allows any process to access any data window stored on any other process. The methods to access remote data are also accessible through UV-CDAT `cdms2` variables.

1.2 Implement a connectivity algorithm for multi-resolution mosaics (Task 1)

The mosaic LibCF library is compatible with multi-resolution mosaic grids. The library makes no assumption regarding the resolution of the different tiles comprising the mosaic. In some applications, it may be desirable to increase the resolution on a specific tile. For instance, the Weather Research and Forecasting model (WRF) [9] applies Structured Adaptive Mesh Refinement (SAMR) to improve local grid resolution. There is a need to extend SAMR to mosaics, including the case where the local mesh refinement borders a mesh tile.

Recall that inter-tile contacts are described using strings such as

```
"99:99 0:99 | 99:0 0:0"
```

which explicitly refer to the range of indices `jbeg:jend ibeg:iend` on either side of the contact (" | "). Here, j, i are the two indices in two dimensions with j pointing in the logical north direction and i pointing in the logical east direction. C/Python index ordering is assumed with the data being contiguous in memory along the i direction. The ranges are inclusive, i.e. `jbeg:jend` represents the list `jbeg, jbeg+1, ... jend-1, jend`. In this particular case, the leading index j for the first tile can be seen to be 99 with the second index (i) running from 0 to 99 along the contact. For a tile of resolution 100×100, this would be the north side of the contact. For the second tile, j runs from 99 to 0 along $i = 0$ and this corresponds to the west side. The corresponding connectivity is shown in Fig. 7.

The contact syntax extends naturally to tiles with different resolution. If the second tile has half the resolution of the first tile, we would express this as

```
"99:99 0:99 | 49:0 0:0".
```

The ratio of resolution on one side and the other need not be an integer. Clearly, any mismatch in resolution will cause some nodes to be dangling, i.e. have no corresponding node on the other side of the interface. Applications in need to cross contact boundaries will likely require some form of interpolation to determine field values across interfaces. We anticipate that the interpolation method used will be application dependent. The adaptive mesh refinement library Chombo [10] is known to apply quadratic interpolation at fine-coarse interfaces for elliptic operators [11].

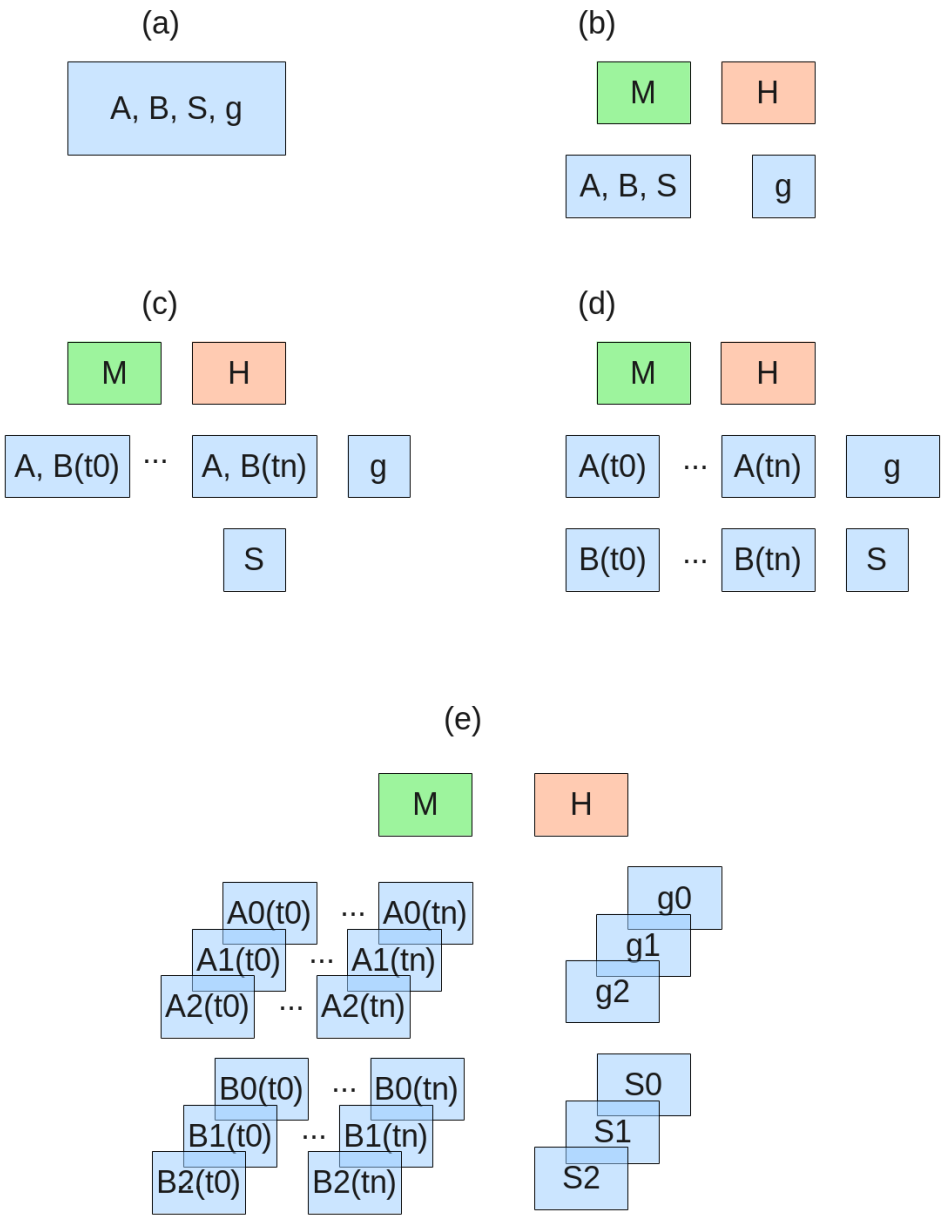

Figure 2: Examples of file aggregation scenarios supported by Gridspec for two time dependent fields (A and B) and one static field (S): (a) fields and grid coordinates (g) are stored in a single file; (b) data and grid are stored in separate files; (c) files are organized by time slices $t0 \cdots tn$; (d) files are organized by time slices and variable names; and (e) files are organized by time slice, variable name, and mosaic grid partition. A host file (H) must be supplied whenever data are scattered across multiple files. A mosaic (M) file must be supplied whenever the data and coordinates are not stored in the same file.

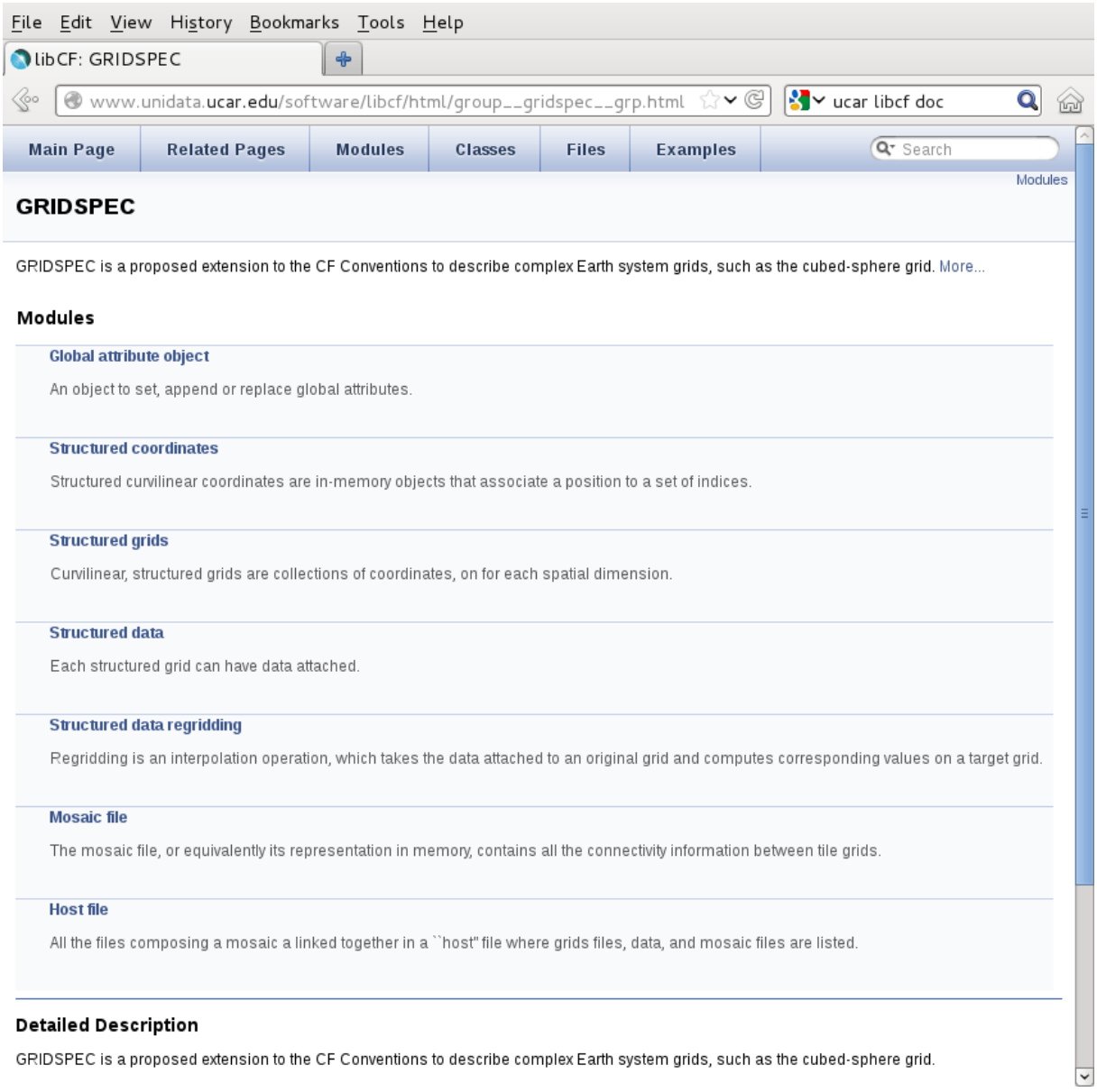

Figure 3: Gridspec/LibCF object layering. Objects such as `axes`, `coordinates`, `grids`, `data`, `regrid` objects, etc., only depend on objects listed above.

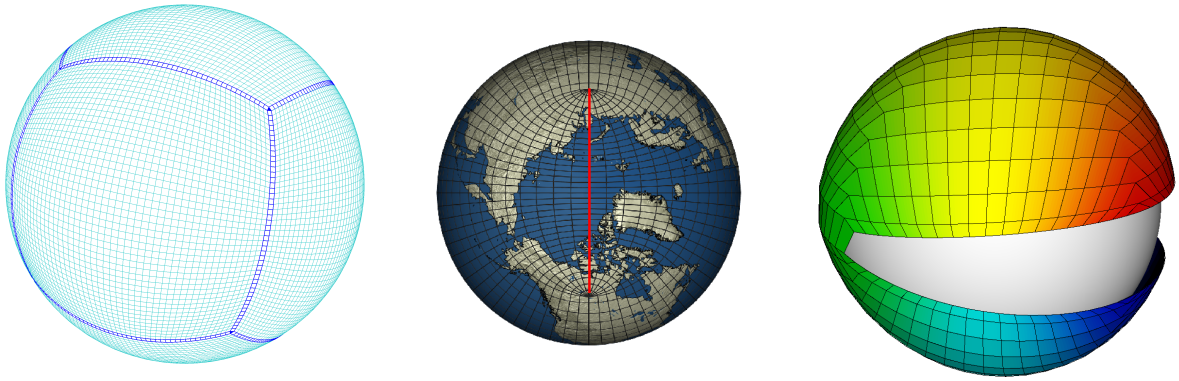

Figure 4: Examples of mosaics conforming to the M-SPEC described in [3]. Left: cubed-sphere mesh. Center: tripolar mesh with a cut along the red line. Right: the logically rectangular grid proposed by Calhoun, Helzel, and LeVeque.

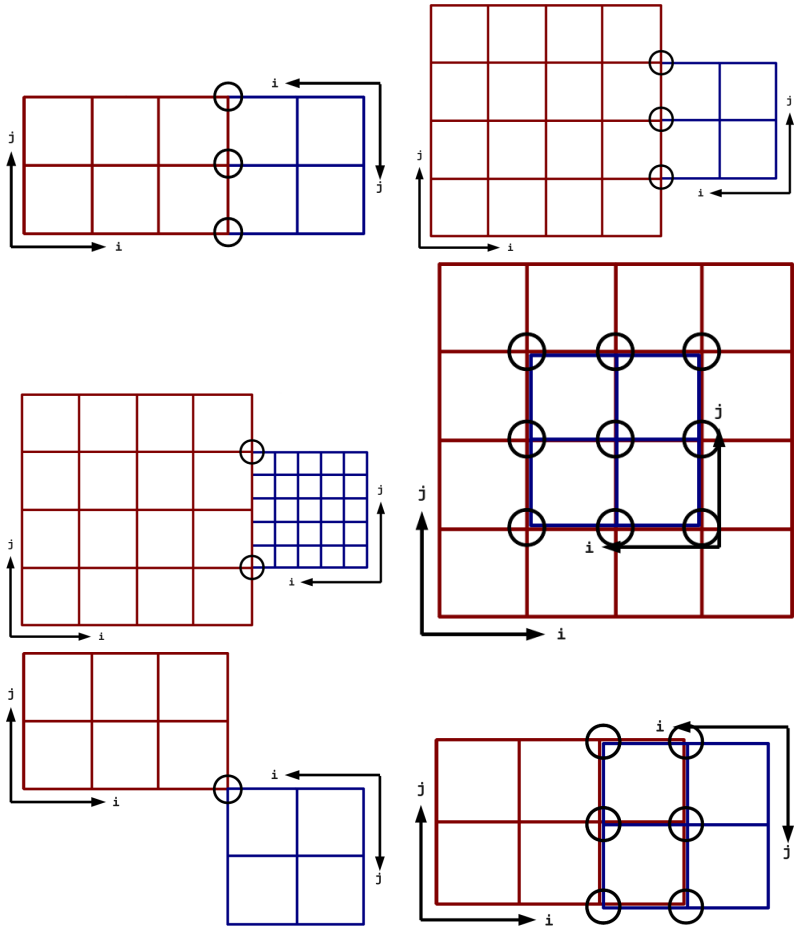

Figure 5: Examples of mosaic tile connectivities conforming to the M-SPEC described in [3]. This covers cases where contacts are a surface in three dimensions (surfacial contacts) and volumes in three dimensions (volumetric contacts). As can be noted, the resolution can differ from one tile to its adjacent.

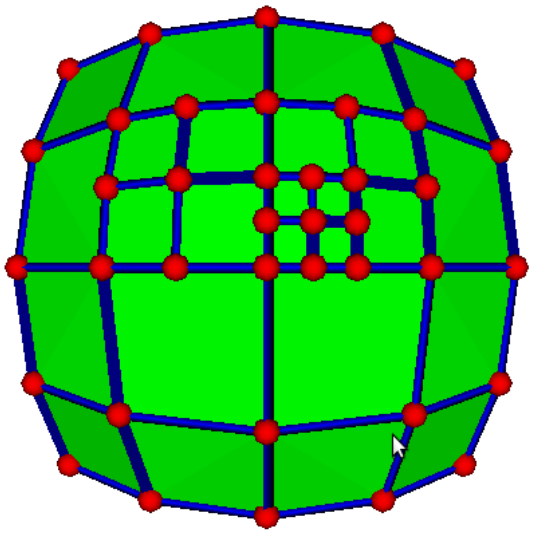

Figure 6: A multi-resolution grid can be obtained by refining any cells of a rectangular grid. Shown here is a cubed-sphere tile built of polytopes and containing two levels of refinements.

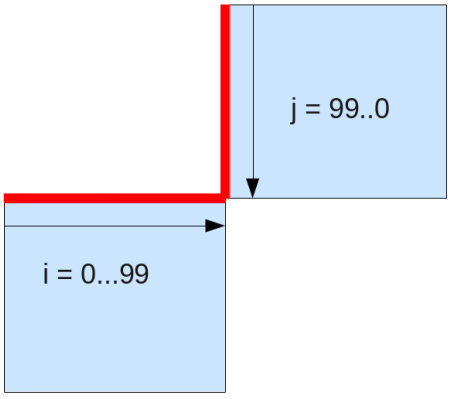

Figure 7: Folding of two tiles of the cubed-sphere in index space.

Of interest is the mapping between one set of indices to the indices on a neighboring tile that shares a contact. Let the tuple $n = (n_0, n_1, \cdots n_{D-1})$ denote the corner, edge, or face of a tile, with the elements of n taking one of *three* values: $n_d \in \{-1,0,1\},\ d = 0, \cdots D-1$ (D is the number of dimensions). A value $n_d \neq 0$ indicates that the tile has a boundary face with a normal vector n_d along the dimension d. There are two directions along such a dimension, the values -1 and $+1$ serve to distinguish between the north/east and south/west faces of a tile. A value of n_d =0 indicates that the tile has no boundary contact along dimension d. Specifically, $(1, 0)$ refers to the east side, $(0, 1)$ to the north side, $(-1, 0)$ to the west side, etc. Corner contacts can be specified using two non-zero elements; e.g. $(1, -1)$ to denote south-east. In three dimensions, more possibilities for combining 1, 0, and -1 exist and can be used to tag faces, edges, and corners. Figure 8 shows the association between contacts and the n tuple.

Once contacts can be uniquely identified using the n-tuple, we need to describe how indices on one tile map to indices on a neighboring tile. Figure 9 shows an example of three tiles that have a complex folding pattern. Each tile has a regular indexing scheme, which can start at any of the four corners of the tile and this is represented as a set of i and j axes. The dimensions of tiles 0, 1, and 2 are $n \times k$, $n \times m$, and $k \times m$, respectively, with the number of cells n, m, and k not necessarily equal.

Starting at index position S on tile 1, an observer moving along the i axis would encounter tile 2 when $i = n$. As our observer crosses the $(1, 0)$ contact and enters tile 2, its indices get rotated and translated according to

$$\begin{pmatrix} i_2 \\ j_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i_1 \\ j_1 \end{pmatrix} + \begin{pmatrix} -n \\ m \end{pmatrix}$$

where i_1, j_1 (i_2, j_2) refer to indices on tile 1 (tile 2). Similarly, when moving along the $-j$ direction on tile 2 and through point A, our observer crosses the south contact of tile 2, denoted by tuple $(0, -1)$, to enter tile 0, at which point the transformation

$$\begin{pmatrix} i_0 \\ j_0 \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} i_2 \\ j_2 \end{pmatrix} + \begin{pmatrix} 0 \\ k \end{pmatrix}$$

applies. We see that the set of indices on one tile can be expressed in terms of indices on a neighboring tile as a *rotation matrix* and a *translation vector*. The rotation matrix involves multiples of 90 degree rotations and therefore contains only 0, 1, and -1 elements. This matrix is invertible; given the transformation from tile 1 to tile 2, for instance, the transformation from 2 to 1 can always be computed. Also note that the *absolute* resolution (number of cells) does not enter in the rotation matrix. Moreover, the rotation matrix can take an additional factor r or $1/r$ in cases where the contact involves tiles with different refinement ratios. The factor r represents the ratio of the two tile resolutions along the contact and this factor does not change if the resolution is, for instance, doubled (for all tiles).

Thus, we have shown that navigating a mosaic involves applying a sequence of rotation and translation operations, and that these operations apply to multi-resolution tiles with the slight complication that rotation matrices must be multiplied by the refinement ratio between two tiles. The translation vector can be determined from the distance in index

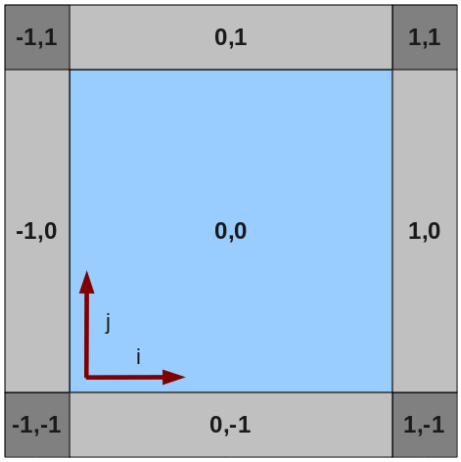

Figure 8: Example of contact labeling according to the n tuple. Values of ± 1 in n indicate a contact; several ± 1 values indicate that the contact is either an edge or a corner in 3D, or a corner in 2D. The n tuples are with respect to an index coordinate system, i and j in this case.

space between the two reference index coordinate systems connecting two tiles. Only for translations does the tile resolution matter.

1.3 Implement hierarchical mosaics (Task 2)

The incidence relation [12] defines the connectivity of a polytope in the hierarchy with respect to other (sub-)polytopes. Thus, an edge polytope is made of two vertices, a face is bounded by edges, and a cell is built from several faces. In this process, the incidence relation binds an n-polytope to $(n-1)$-polytopes. Similarly, a mosaic can be thought of as a polytope with an incidence relation pointing to its tiles. For instance, the cubed-sphere mosaic can be regarded as a 3-polytope containing 2-polytope structured grids. Hence, the incidence relation can be employed to connect grid tiles within a mosaic.

All n-polytopes, $n > 0$, are bounded by lower dimensional polytopes. For example, an edge is bounded by two vertices; a face is bounded by edges; a cell is bounded by faces. This process can be pursued to higher dimensions. The relation between a polytope and one of the lower-dimensional bounding polytope is known as the incidence relation [12]. Structured, unstructured, locally refined, block structured, or mosaics can be thought of as assemblies of polytopes. Moreover, these assemblies can be constructed recursively starting with vertices or nodes, and ending with cells, tiles, and/or mosaics. By using data structures that are aware of the underlying hierarchy between polytopes, it becomes then possible to navigate the hierarchy, query for the neighbors of a polytope, or determine its parent (e.g. the index of the cell owning an edge).

To illustrate how polytopes are constructed, consider the example shown in Fig. 10, which consists of three vertices indexed 0, 1, and 2. Nodes 0 and 1 are connected to form edge 3, nodes 1 and 2 to form edge 4, and nodes 0 and 2 to form edge 5. The three edges are then connected to produce a face. The corresponding polytope hierarchy is shown in Fig. 11 as a directed graph with the arrows representing an incidence relationship.

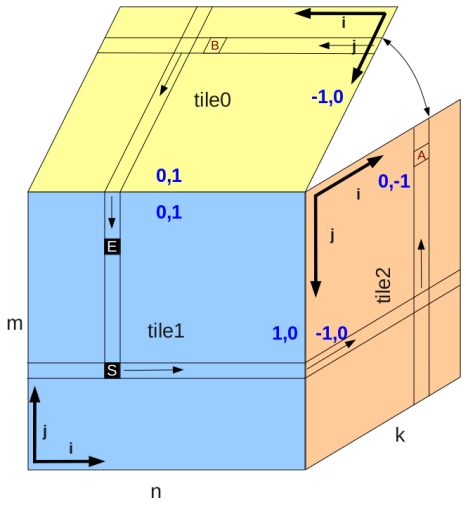

Figure 9: Example of folding in a three tile mosaic. The reference index coordinates for each tile are shown in black. The thin arrows show the path in index space mentioned in the text.

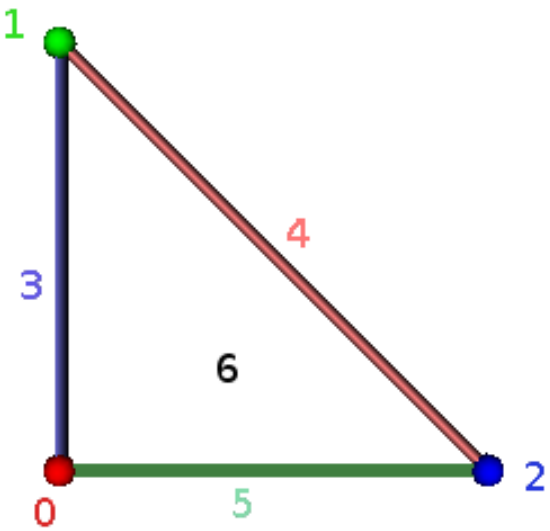

Figure 10: A two-dimensional polytope built from lower dimensional vertex, and edge polytopes. Each polytope has a single integer identifier.

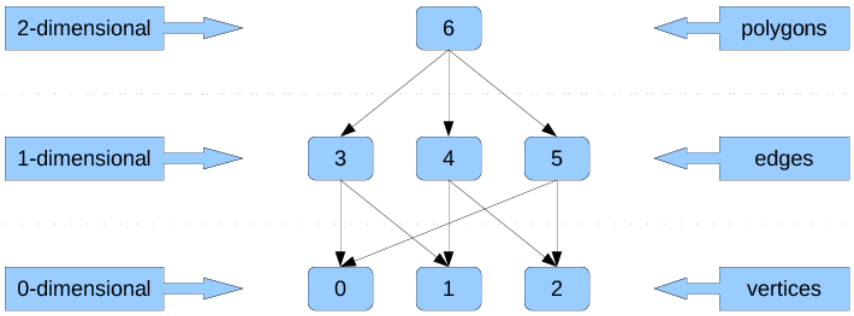

Figure 11: The polytope hierarchy corresponding of a triangular cell. An arrow indicates an incidence relation; for instance, edge 3 is bounded by vertices 0 and 1.

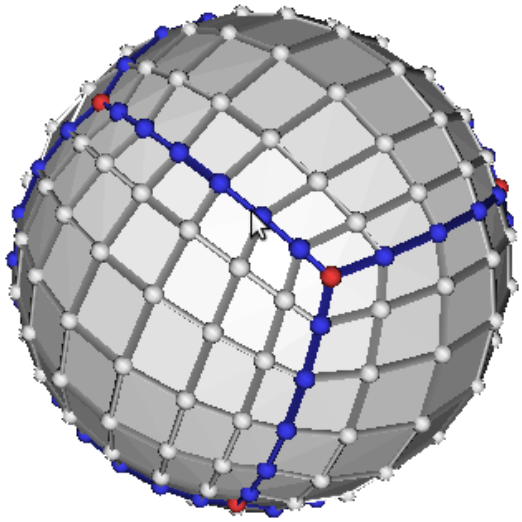

Figure 12: The cubed-sphere as a polytope hierarchy of nodes, edges, and cells. Notice that the number of cells need not be identical for each tile and can vary along each logical direction, as in this case.

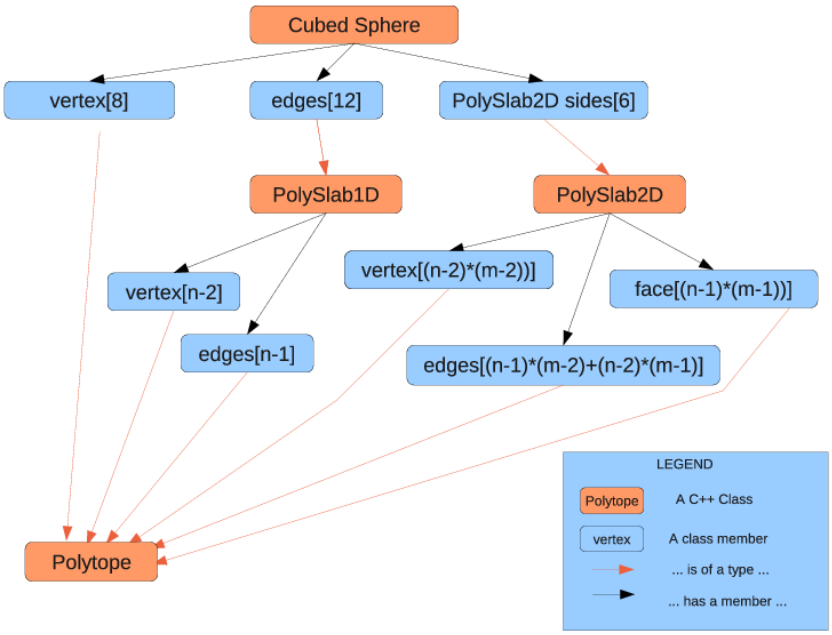

Figure 13: Hierarchy of polytopes composing the cubed-sphere grid (an example of mosaic grid). Each sub-grid is made of vertices, edges, and possibly faces. Notice that the numbers n and m of cells along each of the two logical directions are allowed to differ.

1.4 Implement building blocks for scalable post-processing (Task 3)

As the development of the project matures, the need for distributed arrays to represent data in a mosaic assembly arises. This is supported with distributed arrays, which exposes a sub-set of their data to be accessed by other processing elements. Domain decomposition for optimal load balancing in irregular polytope hierarchies will be challenging and we anticipate that we will need to use METIS [13] or some other software for that purpose. Here, we are addressing the smaller challenge of building a distributed array functionality, which in the spirit of this project, works in any number of dimensions.

We have redesigned our initial distributed array Python module by improving integration with `numpy` [14]. Previously, our prototype implementation defined boxes, i.e. slices of data arrays, which had starting and ending index sets that were not required to be positive. Negative indices have the special meaning of indicating distance in index space from the last element + 1 in Python, which is the language used in our target application CDAT [15]. For instance, if `a` is a Python array the `a[-1]` refers to the last element of `a` and `a[-2]` to the previous before last. This shortcoming was reported to us by Charles Doutriaux from Lawrence Livermore National Laboratory. Another major shortcoming was that other processing units had to access all the data held by a given processor as a contiguous block in memory. In many applications only a thin slab of data needs to be accessed. Clearly, our Phase I implementation of distributed arrays triggered too much communication.

The above shortcomings have been overcome in the present incarnation of `DistArray`

while keeping all the advantages of multi-dimensionality we had built in our original implementation. `DistArray` still depends on `mpi4py` [16] for remote memory access, which has proven to be both solid and complete by supporting MPI-2 windows [8].

The `mvDistArray` class inherits from `numpy.ndarray`. The constructor,

```
def __init__(self, shape, dtype):
    """
    Constructor
    @param shape tuple of dimensions
    @param dtype numpy type
    """
```

takes a shape and an array type (e.g. `numpy.float64`). Each distributed array object is then free to expose any subpart of its data to other processes using

```
def expose(self, slce, winID):
    """
    Collective operation to expose a sub-set of data
    @param slce a slice object
    @param winID the data window ID
    """
```

where `slce` is a tuple of python slice objects, e.g. `(slice(2, 3, 1), slice(4, 5, 1), ...)`, and `winID` a unique identifier of the window. There is no restriction as to the number of exposed windows. In many applications, we expect the size of each window to be significantly smaller than the amount of data held by each processor. This condition is not enforced, however. It is perfectly valid to expose the entire data or to expose overlapping slices of the data. Note that the user does not specify the destination rank of the data. By relying on the `expose` method, sizes and type of the data can be stored; the size of the messages are known ahead of any communication and this can provide a slight performance advantage over the more commonly use send/receive paradigm.

Messaging is initiated with a `get` method invoked by the processor in need to access data held by another process:

```
def get(self, pe, winID):
    """
    Access remote data (collective operation)
    @param pe remote processing element
    @param winID remote window
    @return array
    """
```

which returns the block of data identified by `winID` on process `pe`. This is a collective operation which requires synchronization on the data provider side (to allow the data to be copied to a buffer) and and on the data consumer side (the receive communication must complete). Thus, we have a pull paradigm where workers own the data and periodically

trigger requests to access data on other processes. At no point does the worker push the data. (This feature could however easily be added upon user request if desired.)

A typical use case for distributed arrays is when a given process needs only access a subset of the data held by neighboring processes. Such a case arises when one needs to apply finite differencing for instance. We have written a class `GhostedDistArray` which will automatically create the ghost regions for a regular domain decomposition in N dimensions given a halo width. For a two-dimensional array, the exposed windows are in the north, east, south, and west sides of the local data array. Figure 14 shows the result of applying the Laplacian operator on a distributed Gaussian field in x and y. Communication to access data in the regions delimitated by dashed lines is required. The fact that the contour lines are smooth is indicative that there are no race conditions or locking taking place. Figure 15 show the weak scaling for this problem on a laptop and a cluster. We note that good scaling can be obtained on commodity hardware. Today, most laptops come with four or more cores. When combined with hyper-threading, speedups approaching eight can be observed using this very simple application programming interface (API).

Several improvements can be made at the expense of a more complicated API. Parallel scaling can be further improved by overlapping computation and communication. We are still relying on the data to be copied to buffers and this step could be performed as soon as the data are ready, as opposed to when the data are ready. Similarly, the `get` operation could be performed well before there is a need for accessing neighboring data. If the windows are contiguous in memory then data copy can be avoided. The windows are presently managed with dictionaries which have $\log(n)$ access time (n being the number of windows in our case).

The latest UV-CDAT version incorporates support for distributed arrays.

1.5 Implement interpolation capability (Task 4)

CDAT lacked interpolation capability from arbitrary curvilinear source grids. We have alleviated this shortcoming by developing a regrid type in LibCF and exposing this functionality to Python. In contrast to other interpolation efforts (SCRIP, ESMF, ...), LibCF operates in arbitrary dimensional space. Only linear interpolation is supported in LibCF, however. (Conservative interpolation will be discussed at the end of this section.)

A Python interface to LibCF regrid has been written (`gsRegrid`). This interface is both simple to use and flexible. An example of `gsRegrid` invocation looks like:

```
from cdms2 import gsRegrid
#...
# .... src_y, src_x are the source curvilinear coordinate values
# or axes, ditto for dst_y, dst_x, ....
# takes numpy or cdat cdms2 type variables
src_grd = [..., src_y, src_x]
dst_grd = [..., dst_y, dst_x]
# constructor
rg = gsRegrid.Regrid(src_grd, dst_grid,
mkCyclic = False,
handleCut = False,
```

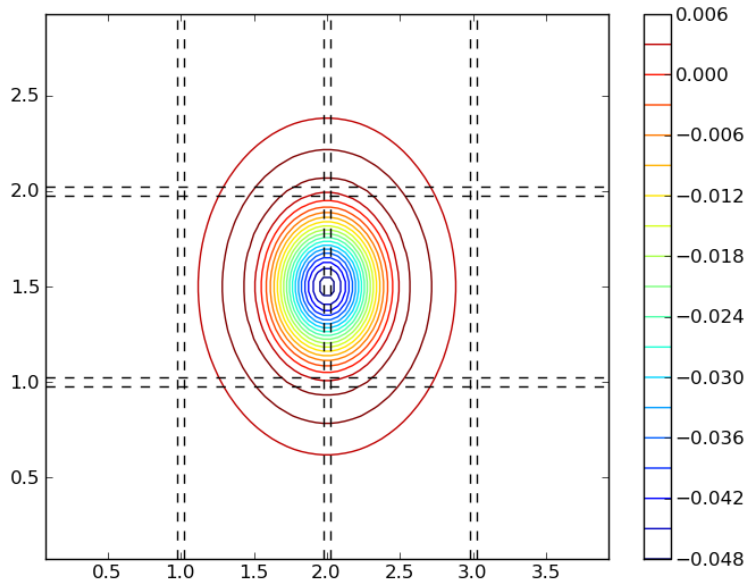

Figure 14: Two-dimensional Laplace operator applied to a distributed Gaussian field in x and y, which is centered around x and $y = 1$. A 3×4 domain decomposition was used with the vertical and horizontal dashed lines representing the delimitation between sub-domains. Twenty cells in x and y were used for each sub-domain. Each sub-domain must access ghost cell data held by other processing elements when applying the Laplacian 5 star stencil.

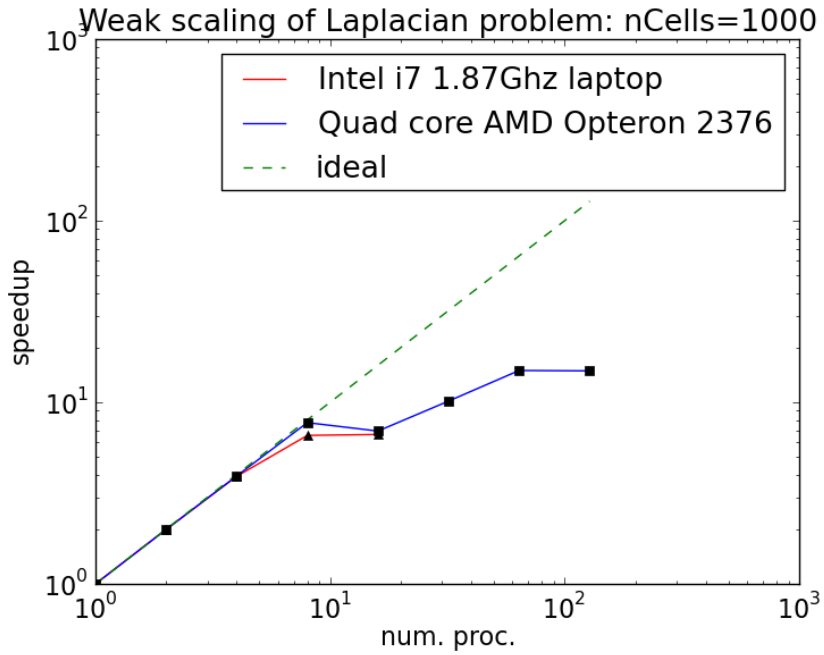

Figure 15: Weak scaling for the 2D Laplacian test problem. The number of cells (1000) along x and y was kept constant as the number of processes was increased.

```
src_bounds = None)
# compute interpolation weights
rg.computeWeights(nitermax=20, tolpos=0.01)
# interpolate src_field, result is dst_field
rg(src_var, dst_var)
```

In essence, the user provides a source `[..., src_y, src_x]` and a destination `[..., dst_y, dst_x]` grid to the constructor `Regrid` in the form of sets of coordinates. The `...` indicate that the number of coordinates is arbitrary. However, the number of coordinates and the number axes for each coordinates must match the number of space dimensions. (The ESMF/ESMP interpolation discussed does not suffer from this restriction, the number of topological and space dimensions can be different.) When the source grid is cell centered, it may be necessary to supply `mkCyclic = True` to fully cover the earth. An additional row must sometimes be added to the source grid in order to handle tripolar grids see Fig. 17 for such an example. The interpolation weights are computed by `computeWeights`, by far the most computationally intensive method. This method takes the maximum number of Newton iterations `nitermax` and the tolerance in (x, y) space as input. Figure 16 shows that the number of of iterations required to find a target position in index space depends in subtle ways on the initial guess — there are cases where two neighboring initial guesses require vastly different number of iterations due to the fractal nature of the nonlinear map from index to x, y coordinates. When the map is linear, i.e. the source is a cross product of axes and the cell spacing is uniform, a single iteration is required. Note that the computation of the interpolation weights can be amortized over the number of variables that share the same source and destination, the `gsRegrid` object will hold weights until it is destroyed.

The final call, `rg(src_var, dst_var)`, fills in the interpolated data `dst_var`. Note that the source and destination grids are not required to overlap. Only the values of `dst_var` that fall within the source grid will be filled. (More precisely, only the grid point values for which the Newton scheme converged to given accuracy will be filled). This makes it easy to iterate over the tiles of the cubed sphere when interpolating to a longitude-latitude grid, for instance.

In addition, it may be necessary to provide a mask before computing the weights. Care is taken that interpolation weights are computed only if the target point falls within a triangle of valid data (see Figs. 18 and 19).

LibCF regrid now has improved handling of the periodicity of longitude coordinates. Previously, the user was required to supply a "forbidden" box, a set of indices delimiting a region where the Newton search should not attempt to search target cells. This is no longer required, instead we now extend our search by allowing the target position to be known modulo the periodicity of the coordinate. We have successfully tested `gsRegrid` on 23 models of ocean data (salinity) provided by Paul Durack (LLNL). A selected sample of data sets is shown in Figs. 20 and 21.

By interpolating back onto the original source grid, we can determine where the interpolation error is largest. Figure 22 indicates that the cumulated error of the two interpolations can amount to 10 % and is mainly localized to the vicinity of coastlines, but is significantly smaller in the bulk part of the domain.

Linear interpolation allows for embarrassingly parallelization. Figure 23 shows the re-

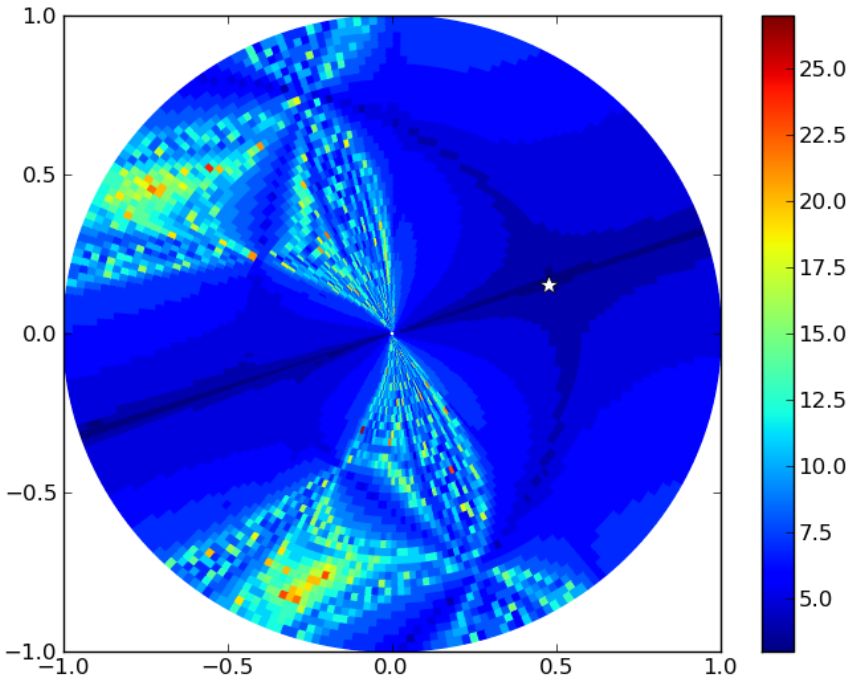

Figure 16: The fractal nature of the pseudo-Newton iteration scheme used to locate a target Cartesian position (white star) in curvilinear polar coordinates.

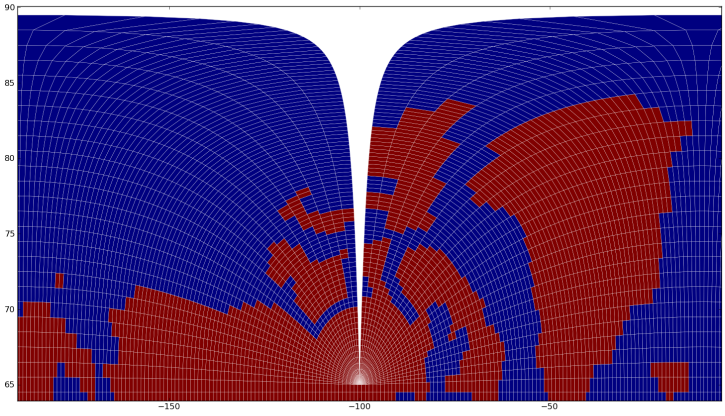

Figure 17: The grid of the tripolar often needs to be extended to prevent the cut to appear as a gap.

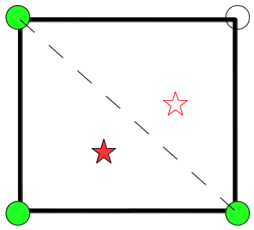

Figure 18: Interpolation of data with missing values (open circle) can be successfully applied if the target point falls within a triangle whose vertices have all valid data (solid star). Interpolation cannot be applied for the open star location. In three dimensions, one must be able to construct a tetrahedron from the valid points.

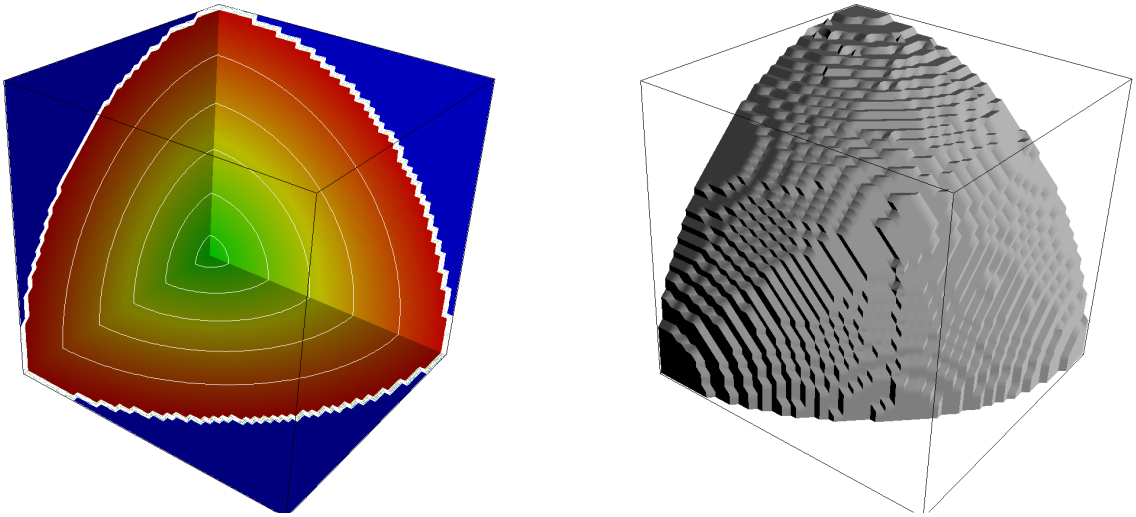

Figure 19: Left: interpolation of data with missing values outside a spherical radius (blue region) in three dimensions. Right: the last valid contour surface. Visualization with VisIt.

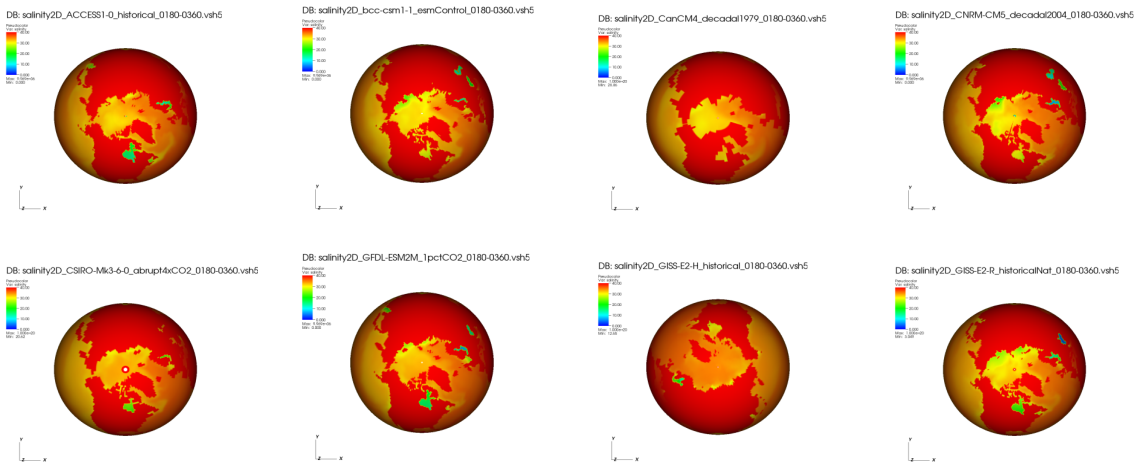

Figure 20: A small selection of ocean data sets (salinity) from different climate models which have been interpolated onto a longitude-latitude grid using `gsRegrid`.

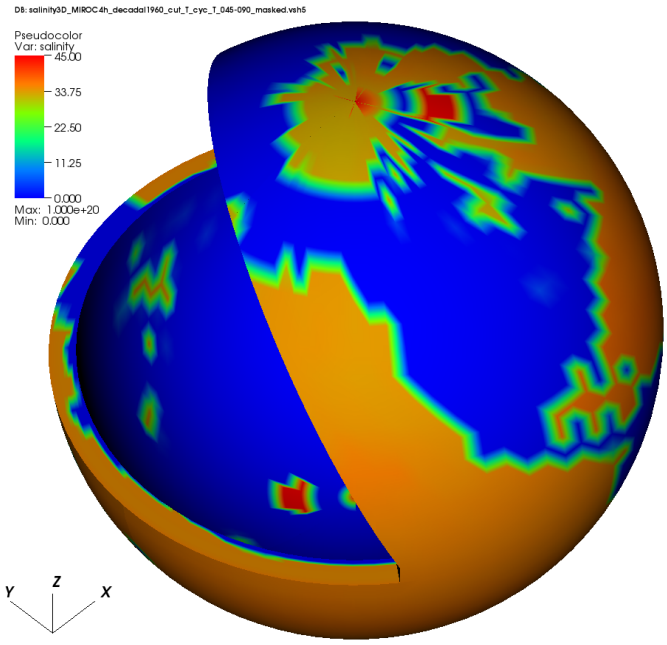

Figure 21: Example of three-dimensional interpolation of salinity.

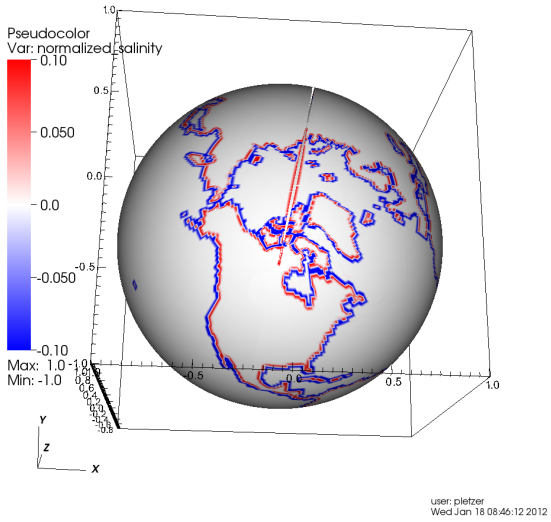

Figure 22: Cumulative linear interpolation error (back to the original grid) relative to the maximum field value.

duction of wall clock time on a workstation with 8 cores. Parallel scaling is limited by load balancing. The time required to compute the interpolation weights is highly dependent on the initial guess and the target location.

We have compared LibCF against ESMF linear interpolation. Figure 24 shows the difference for the same regridding operation using linear interpolation. Note the primary differences occur near the pole. These differences can be explained by the fact that ESMP uses both spatial and topological dimensions and can resolve the polar singularity better than LibCF.

A typical use case is of regridding operation in CDAT is:

```
taOnUGrid = ta.regrid(u.getGrid())
```

where the variable `ta` is regridded onto the mesh provided by `u`. This operation is performed under the hood by the `regrid2` module, specifically by the `regrid2.Regridder` class. We have extended `Regridder`, which takes a source and destination grid, to also take a regrid method and a regrid tool. The default regrid method for UV-CDAT is bilinear and the default tool is gsRegrid (LibCF).

The pseudo-code shows an example of the regridder method.

```
import regrid2
f = cdms2.open('test_data.nc')
g = cdms2.open('more_data.nc')

# read the data
v = f('some_data')
w = g('more_data')
regridMethod = 'Conservative' # Or Bilinear
```

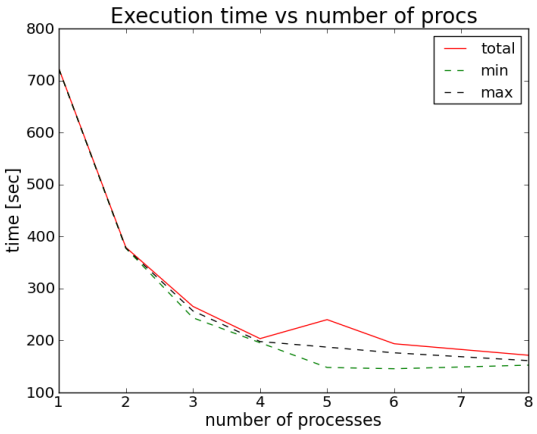

Figure 23: Strong parallel scaling of linear interpolation for a three-dimensional test problem.

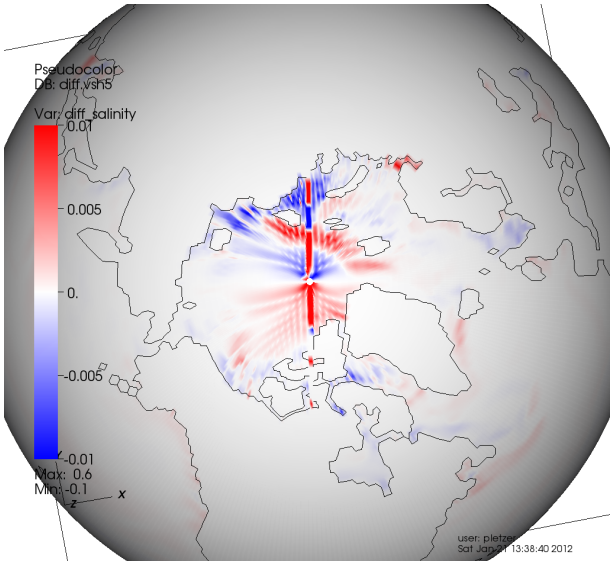

Figure 24: Comparison of LibCF versus ESMP linear interpolation for a three-dimensional test problem.

```
regridTool = 'gsRegrid' # 'libcf', 'ESMP', 'SCRIP', 'regrid'
regridObj = regrid2.Regridder(v.getGrid(), w.getGrid(),
                  regridMethod = regridMethod,
                  regridTool = regridTool)
newV = regridObj(v)
```

where the `regridTool` can be one of `"gsRegrid"`, `"ESMP"`, `"SCRIP"` or the original cdms regridding tool `"horizontal"`. The default is gsRegrid. The `regridMethod` at the moment can be `"Bilinear"` or `"Conservative"` (not all tools support conservative interpolation). Though SCRIP supports more options, these have not been activated.

Alternatively, one can access any of the methods directly by import the appropriate module. e.g.

```
from regrid2 import horizontal
from cdms2 import gsRegrid
import ESMP
import SCRIP
```

(the API will naturally change according to the tool).

Extending the `cdms2` method `regrid` to support three dimensional grids with elevation is on the to do list.

1.6 Implement methods for common statistical and filter operations (Task 5)

With interpolation, all the ingredients are in place to perform a wide range of statistical operations, including zonal averages of two- and three-dimensional fields.

1.7 Implement easy-to-use visualization capability (Task 6)

In the process of integrating LibCF into CDAT (see Sect. 1.8), we have written Python classes allowing `cdms2` structured grid objects to save data in file formats for which VisIt plugins exist. Two files formats have been selected: VTK structured grid [17] and VizSchema [18]. The former has the advantage of being supported by many visualization tools in addition to VisIt (e.g. VTK, Paraview, AVS/Express) while the latter relies on the HDF5 format [19] format to store data in binary and portable form. The VizSchema format requires the installation of the PyTables [20] package in Python.

Writing structured data to VTK format is as simple as:

```
import cdms2
f = cdms2.open('test_data.nc')
# read the data
v = f('some_data')
fname = v.id
v.toVisit(fname, format='VTK')
```

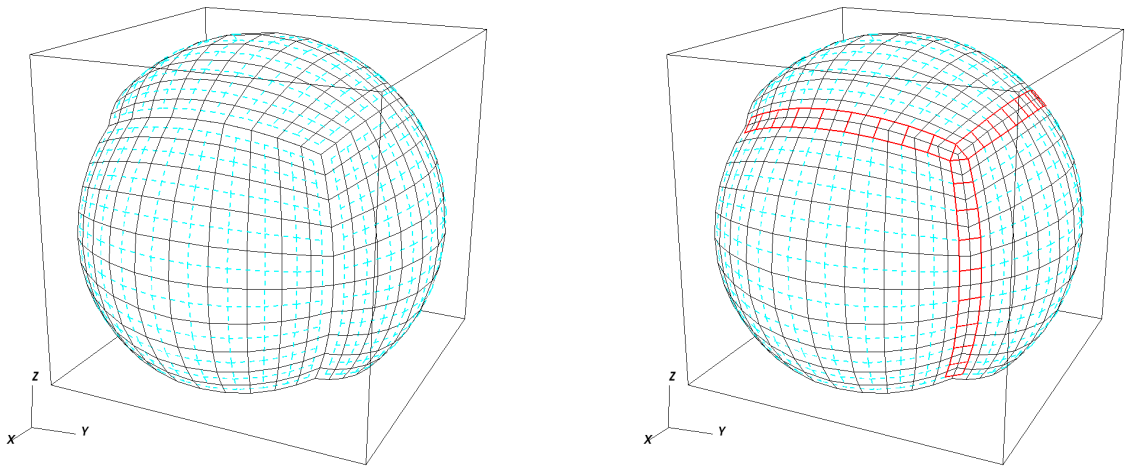

Figure 25: Left: Location of node and cell centered field values for cubed sphere data. Right: Seam grids required to bridge cell centered data across tile interfaces.

This will also project the longitude and latitude coordinates onto a sphere. To write data using the VizSchema format, set `format='Vs'`.

The above `toVisit` method allows any structured, scalar field to be saved for visualization by VisIt. A mosaic cell-centered field, such as one defined at the nodes of the dashed grid in Fig. 25 (left), cannot be simply visualized as a collection of structured fields as this would expose gaps between tiles. In other words, a correct cell centered visualization requires the connectivity information between mosaic tiles in order to generate the seam data at grids shown in red in Fig. 25 (right). Figure 26 shows how gaps can arise in visualizations when the connectivity between tiles is not properly accounted for (left). In the middle, the same data with seam data added; careful examination shows that the corner cell at the junction of three tiles is missing a datum. On the right, the final visualization with seam and corner data added.

Both two- and three-dimensional data are supported. Data can be static or time dependent (each time slice will be stored in a separate file). Figure 27 shows the cell centered air temperature produced by the coupled model CM2.1 run at the Geophysical Fluids Dynamic Laboratory.

1.8 Integration into the CDAT application (Task 7)

Starting with an initial implementation by Jeff Painter and Charles Doutriaux (LLNL), we exposed the full functionality of LibCF to CDAT. All API calls in the LibCF shared library are accessible via the built-in `ctypes` Python module without the need for Python-C wrapping code.

Upon opening a netcdf file [21] with the `cdms2.open` CDAT method, the file object will be identified as being a "host" object if the file contains the global attribute `gridspec_file_type` and that attribute is set to `'host_file'`. Then, the entire data set will be opened as a log-

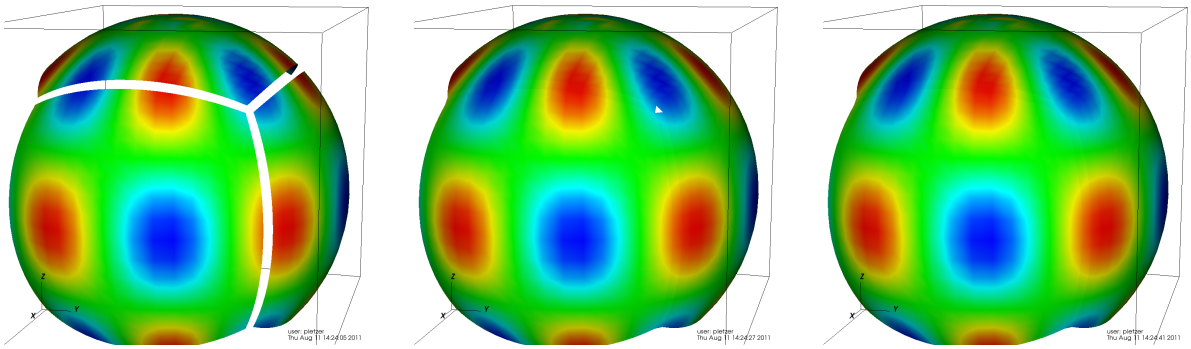

Figure 26: Filling in cell centered data gaps between three cubed sphere tiles.

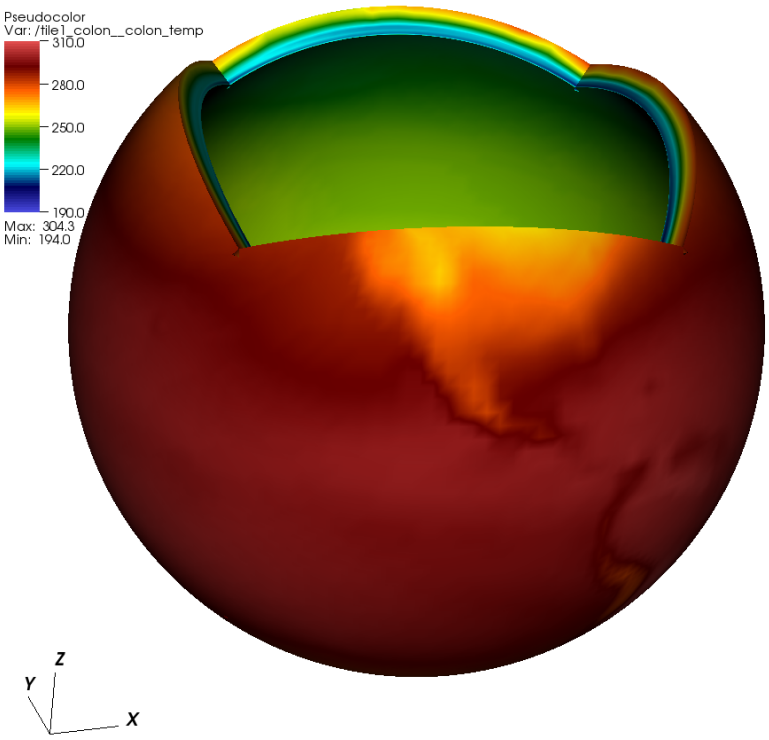

Figure 27: Example of mosaic visualization: air temperature from the CM2.1 GFDL model after removing one tile. The vertical axis direction was reversed in order to show the surface temperature on the outer surface.

ically single entity with the host acting as the entry point to the file collection (see Fig. 2). The host behaves as if all data were contained within a single file as per the newly extended CF file convention [3]. The host can be queried for the attributes, dimensions, grids, fields and the mosaic object, as illustrated in Table 1.

Query Type	**host object method**
global	`host.listglobal(s)`
attributes	`host.listattributes`
dimensions	`host.listdimensions`
variables	`host.listvariable(s)`
coordinates	`host.getCoordinates`
mosaic	`host.getMosaic`

Table 1: Query methods for a host object. Parentheses around the 's' in the method indicates the singular or the plural method is valid.

An aggregated variable can be requested simply by using the call `host('varname')`, a syntax that resembles a single file variable read operation in CDAT. This will currently load the data associated with each file in the aggregation and store the result in a list (one element for each tile). Each element in the list currently behaves as a standard CDAT transient (in-memory) variable of type `cdms2.transientVariable`. This means that the grid, attributes, dimensions, shape etc. for the data on each grid are available. The variables can be time varying or static and all of the CDAT time slice functions and methods as well as the spatial slicing methods are available for that individual element.

Time varying and static variables can be stored in separate files. The number of time steps within a time varying file need not be one, so building a time dependent variable requires an extra loop for the number of time files available. The files for time files should be listed in monotonically increasing order within the host file (the LibCF method `nccf_add_host_file` will take care of inserting the element in the correct position).

There are some issues left to address. First, the `cdms2.transientVariable` is very memory intensive. In addition to transient variables, CDAT also supports file (`cdms2.fileVariable`) variables, which only store the metadata in memory while leaving the actual data stored on disk. We have yet to implement a corresponding behavior when accessing a variable through host file, based on the square bracket syntax `host['varname']`. Second, only read access of Gridspec variables is presently supported; the need to generate a Gridspec aggregation conveniently within CDAT has to be addressed. (It is still possible to access all LibCF methods from Python, including those allowing the construction of a Gridspec aggregation; however, these methods are not presently accessible through a CDAT interface.)

2 Products and publications

The proposed Gridspec extension to the NetCDF Climate and Forecast Metadata Conventions [3], a collaboration between Tech-X, LLNL, PMEL, UCAR, and GFDL, has been officially accepted.

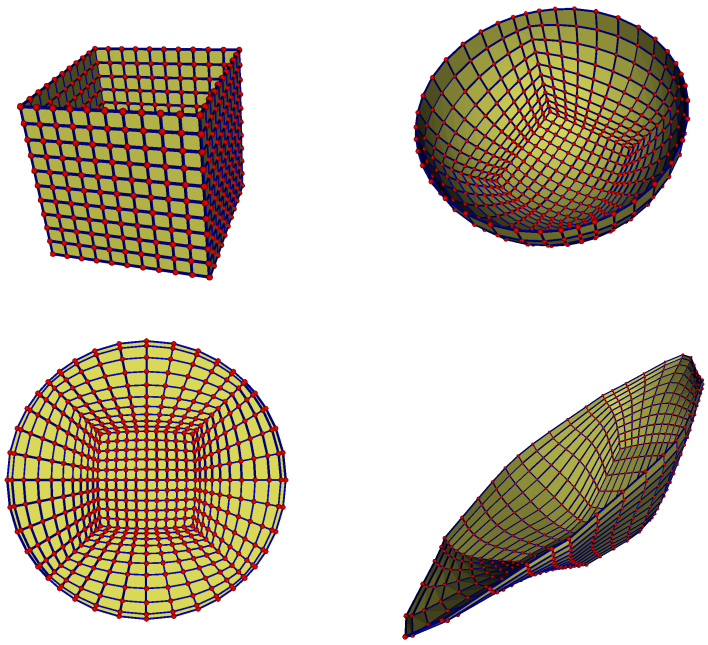

Figure 28: The five tile topology of the cube (top left) can be used as a starting point for the generation of many novel grids.

An article entitled "The Ultra-scale Visualization Climate Data Analysis Tools (UV-CDAT) Data Analysis and Visualization for Geoscience Data" with Dean Williams (LLNL) as lead author was submitted to the Computer magazine. This article features LibCF/ESMF interpolation and the use of distributed arrays to accelerate data analysis.

Spin-off technologies that are currently explored in other projects include the use of mosaics in solving partial differential equations. Starting a with simple box topology, families of grids can be generated by applying a sequence of coordinate transformations (see Fig. 28). The cubed-sphere grid is obtained by projecting the cube into a sphere. The "cubed-disk" grid can be obtained by projecting the cubed-sphere grid onto a plane. Novel grids can be generated by applying other types of transformations.

2.1 Technologies/techniques

The techniques developed during this project find applications outside climate data analysis — mosaic grids can be applied to any area where there is a need to avoid pole-like singularities.

3 Bibliography & References Cited

[1] "Gridspec: A standard for the description of grids used in Earth System models." http://http://www.gfdl.noaa.gov/~vb/gridstd/gridstd.html.

[2] "Climate and Forecast NetCDF Metadata conventions." http://cf-pcmdi.llnl.gov/.

[3] "The CF-GRIDSPEC Extensions." https://ice.txcorp.com/trac/modave/wiki/CFProposalGridspec.

[4] "LibCF - The NetCDF CF Library." http://http://www.unidata.ucar.edu/software/libcf/.

[5] "The earth system modeling framwork." http://www.earthsystemmodeling.org/.

[6] "Ultrascale Visuzalization – Climat Data Analysis Tools." http://uv-cdat.llnl.gov/.

[7] "Cross platform build system." http://www.cmake.org/.

[8] "Message Passing Interface Forum." http://www.mpi-forum.org/.

[9] "WRF: The Weather Research and Forecasting model." http://wrf-model.org/index.php.

[10] "Chombo AMR infrastructure." http://seesar.lbl.gov/ANAG/chombo/.

[11] D. Martin and K. Cartwright, "Solving Poisson's equation using adaptive mesh refinement." http://seesar.lbl.gov/anag/publications/martin/MartinCartwright.pdf.

[12] B. Howe and D. Maier, "Algebraic Manipulation of Scientific Datasets," *Proceeding of the 30th VLDB conference*, 2004. Toronto, Canada.

[13] "Metis - family of multilevel partitioning algorithms." http://glaros.dtc.umn.edu/gkhome/views/metis.

[14] "NumPy – Scientific computing with Python." http://numpy.scipy.org/.

[15] "Climate Data Analysis Tools." http://www-pcmdi.llnl.gov/software-portal/cdat.

[16] "MPI for Python." http://mpi4py.scipy.org/.

[17] "ASCII VTK Files." http://visitusers.org/index.php.

[18] "Vizschema-3.0." https://ice.txcorp.com/trac/vizschema/wiki/WikiStart/.

[19] "The HDF Group: HDF5." http://www.hdfgroup.org/HDF5/.

[20] "PyTables: Getting the most out of your data." http://www.pytables.org/moin.

[21] "NetCDF Web Site." http://hdf.ncsa.uiuc.edu/.