Fast regridding of large, complex geospatial datasets

J.D. Blower
Reading e-Science Centre
University of Reading
Reading, United Kingdom
j.d.blower@reading.ac.uk

A. Clegg
School of Systems Engineering
University of Reading
Reading, United Kingdom

ABSTRACT

In the earth sciences, data are commonly cast on complex grids in order to model irregular domains such as coastlines, or to evenly distribute grid points over the globe. It is common for a scientist to wish to re-cast such data onto a grid that is more amenable to manipulation, visualization, or comparison with other data sources. The complexity of the grids presents a significant technical difficulty to the regridding process. In particular, the regridding of complex grids may suffer from severe performance issues, in the worst case scaling with the product of the sizes of the source and destination grids. We present a mechanism for the fast regridding of such datasets, based upon the construction of a spatial index that allows fast searching of the source grid. We discover that the most efficient spatial index under test (in terms of memory usage and query time) is a simple look-up table. A kd-tree implementation was found to be faster to build and to give similar query performance at the expense of a larger memory footprint. Using our approach, we demonstrate that regridding of complex data may proceed at speeds sufficient to permit regridding on-the-fly in an interactive visualization application, or in a Web Map Service implementation. For large datasets with complex grids the new mechanism is shown to significantly outperform algorithms used in many scientific visualization packages.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Information Storage and Retrieval; I.3.3 [**Picture/Image Generation**]: Computer Graphics; H.2.8 [**Database management**]: Database applications—*Spatial databases and GIS*

General Terms

Experimentation, Performance

Keywords

regridding, GIS, Web Map Service, visualization, curvilinear grids, spatial index

1. INTRODUCTION

Gridded data appear very frequently in environmental and geospatial science. Such data are produced by numerical models and many kinds of remote sensing instruments, and may also be derived from the interpolation of in situ measurements. It is common for a scientist to re-cast such data onto new grids for the purposes of visualization, comparing datasets with each other, for calculating derived fields that are functions of more than one variable, or simply for reasons of convenience in data handling. This is known as *regridding*.

Geographic Information Systems (GIS) commonly perform regridding operations on raster datasets in order to display a dataset in a user-specified coordinate reference system (CRS). In a service-oriented architecture, this regridding may be performed on a server, such as a Web Map Service (WMS, [6]). The transformation from one CRS to another is often defined as a mathematical function. However, in the Earth sciences, notably meteorology, oceanography and climate science, complex grids are commonly used [9], which do not have transformations from grid space to real space that are easily captured as mathematical functions. This leads to technical difficulties in regridding these datasets and hence hinders their use in visualization tools, including GIS.

Although earth science data are commonly multidimensional (covering two horizontal dimensions, a vertical dimension and time), we are most concerned here with the two-dimensional horizontal component of the grids, which is usually the largest and most complex in terms of its relation to real space. We use the following terminology: in a *rectangular* grid, each cell is addressable by two integer indices i and j. Rectangular grids may be further categorized by their relation to real space. In *rectilinear* grids, the two grid axes are aligned with axes in a real-world CRS, and so the grid cells are rectangular in the CRS. *Regular* grids are rectilinear grids in which the grid cells are square in the CRS. *Curvilinear* grids are rectangular grids that are referenceable to a real-world CRS, but the grid cells are not rectangles in the CRS.

Curvilinear grids are found in many Earth science communities, notably oceanography, in which they are used to ensure that the grid follows complex features such as coastlines and straits (see figure 1 for an example). They can also be defined in order to avoid singularities at the poles; the tripolar grid [9] is an example of this. Curvilinear grids are usually defined empirically by explicitly specifying the real-

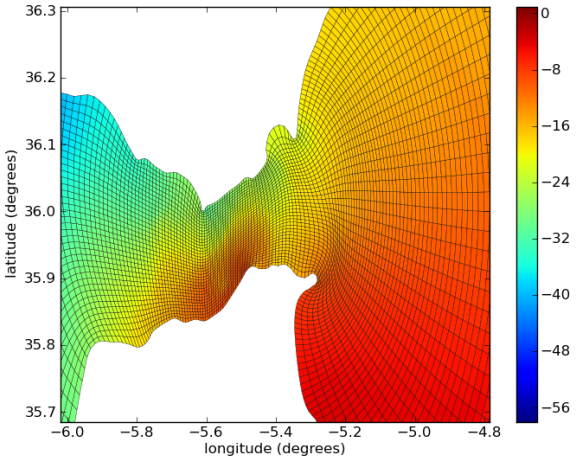

Figure 1: An portion of a sample curvilinear grid covering the Strait of Gibraltar, illustrating a snapshot of sea level in centimetres derived from one of the test datasets in this study (UCA, see table 1). In latitude-longitude space, the quadrilateral grid cells are distorted in order to resolve small-scale features in the strait.

world coordinates (usually spherical coordinates, i.e. longitude and latitude) of each grid cell[1].

The primary motivation behind this study was to devise an efficient algorithm for server-side generation of images of curvilinear datasets for use in a Web Map Service implementation [3]. This paper tests a number of different approaches against datasets of various sizes, from a few thousand to a few million grid cells. This is a short and practical study: a discussion of the theory and technical details behind the techniques under test is beyond the current scope.

2. REGRIDDING ALGORITHMS

Regridding algorithms vary greatly in the manner in which they transfer data from one grid to another. Some algorithms treat the grids as sets of points, largely ignoring the shapes of the grid cells and casting data from one set of points to the other using an interpolation scheme such as nearest-neighbour, bilinear or bicubic. Others calculate the precise overlap of cells in each grid, transferring data between grids in an area-weighted scheme. Some algorithms are designed to be *conservative*, preserving the values of integrated quantities such as mass, energy and fluxes on the new grid [8].

Algorithms can broadly be divided into two types. An algorithm may iterate through the source grid S, transferring data to the destination grid D; we call this a "source-push" algorithm. Such algorithms are common in scientific visualization tools including Matplotlib [7] and Panoply[2], which both iterate over S, drawing each grid cell onto a canvas representing D. The Modis Swath-To-Grid toolbox[3] also iterates over S, recording weights (based on euclidean distance) and data values in D, normalizing the values by the weights at the end of the iteration to produce an output image.

Alternatively, the algorithm may iterate through D, extracting corresponding data from S; this is a "destination-pull" algorithm. Such an algorithm is employed in ncWMS, a Web Map Service implementation for multidimensional Earth science data [3].

A key factor controlling the efficiency of these algorithms is the *invertibility* of the source and destination grids. For the purposes of this paper we say that a grid is invertible if there exists an $\mathcal{O}(1)$ algorithm that will find the grid cell that contains a given real-world position. By this definition, rectilinear grids are invertible, but curvilinear grids are usually not. In this paper we are particularly concerned with cases involving large, non-invertible source grids. The relative performance of the two classes of algorithm will depend on many factors including the relative sizes of the two grids. In general, however, for a source grid with N_S cells and a destination grid of N_D cells, it is clear that a "destination-pull" algorithm will be highly inefficient ($\mathcal{O}(N_S N_D)$) if S must be searched exhaustively to find correspondences with every point in D. We therefore seek strategies that allow for much more efficient searching of large, non-invertible source grids such as curvilinear grids.

3. SPATIAL DATA STRUCTURES

There is a rich literature in the use of data structures and algorithms that allow multidimensional space to be searched efficiently by using the data structure as an index into the source data [10]. In this study we consider that the primary purpose of the spatial data structure is to find the cell C_{SP} within S that contains a given point P. From this initial search, most regridding schemes are then implementable. Note that in the case of non-rectilinear grids, P may be closer to the centre of a neighbouring cell than it is to C_{SP} (see figure 2) so this query is not quite the same as a nearest-neighbour query on the set of the centres of cells in S.

4. TESTING

In this study, we investigate the suitability of different spatial indexes for finding C_{SP} in three curvilinear datasets of different sizes and characteristics (Table 1). The indexes in this study were chosen because they are well-known and readily available (or relatively easily implemented). Many other alternatives are possible, including algorithms specifically designed to index spheres [11]. A fuller investigation of the large number of alternative approaches is beyond the scope of this short study. Each of the spatial indexes was implemented in Java and tested against each dataset in the following manner:

1. The source grid S was loaded from the source data file.
2. The cells of S were inserted into the spatial index under test, measuring the time taken to construct the

[1]http://www.cfconventions.org

[2]http://www.giss.nasa.gov/tools/panoply/

[3]http://nsidc.org/data/modis/ms2gt/

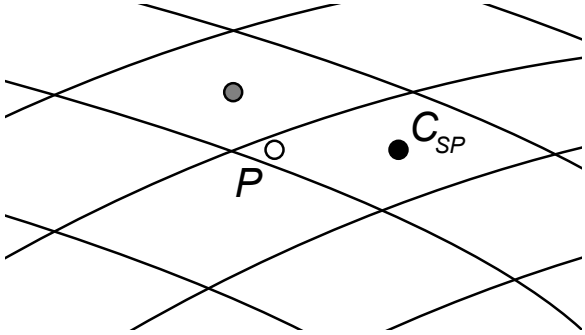

Figure 2: Sketch of a portion of a curvilinear grid in latitude-longitude space. This illustrates that the cell C_{SP} that contains a given point P is not always the cell that contains P's nearest-neighbour grid point (grey point).

index. The cells were defined in longitude-latitude coordinates.

3. The memory footprint of the index was estimated by using Java system calls to measure the size of data on the heap before and after construction of the index[4].

4. A destination grid D of size 256×256 was constructed in longitude-latitude coordinates, covering the spatial extent of the source grid.

5. The spatial index was used to find C_{SP}, taking P as the centre of each grid cell in D, measuring the average time per query.

The test machine was a laptop running Fedora 13, with an Intel Pentium dual core T3400 2.16 GHz processor and 3 GB of RAM. Four spatial indexes were tested:

R-Trees [10] are among the most widely-used spatial indexes, storing rectangles (in the 2D case) and allowing fast queries to find all the rectangles within the R-tree that intersect a given rectangle. We tested an R-tree implementation from the Java Spatial Index library[5], version 1.0b6. The **Priority R-Tree** (PR-tree [2]) is an R-tree variant that exhibits $\mathcal{O}(\sqrt{N})$ worst-case query performance. We used a Java implementation of the PR-tree[6], version 1.4. In each case, the index was used to store the minimum bounding rectangle (MBR) of each cell in S, then the intersection query was used to find all MBRs that contain P. The cells corresponding to these MBRs were then searched to find which cell contains P, using the `contains()` method of the `java.awt.geom.Path2D` class. The R-tree implementations allowed the setting of a parameter (the branching factor). Results were found to be insensitive to the choice of value for this parameter, and so the default value of 10 was used.

Adaptive Kd-trees [10] store points rather than rectangles and exhibit $\mathcal{O}(\sqrt{N})$ performance for searching for points that lie within a given target rectangle. We implemented an adaptive kd-tree in Java, populating it with the centres of all the grid cells in S. Initially the standard nearest neighbour search algorithm was implemented, but this was found to perform very poorly, due to the existence of cells in D which do not have a corresponding P in S, and therefore having a nearest neighbour lying a considerable distance away (e.g. cells in the centre of a masked region). An alternative approach was adopted, where for each P, the kd-tree was searched by constructing a bounding rectangle around P and finding the points in the kd-tree that contain the rectangle. If the rectangle did not contain any points, it was repeatedly expanded by an expansion factor until results were returned or a defined size limit was reached. Of these results, the nearest point to P was found by linear search through the result set. This cell, together with its neighbouring cells, were then searched to find C_{SP} using `Path2D.contains()` as above. The use of this technique improved performance, but suffers from the need to choose empirically the initial rectangle size and the expansion factor. In these tests, these parameters were chosen to be optimal for each individual dataset by exhaustively testing a large number of parameter combinations and selecting the combination that yielded the shortest query time.

The final approach we investigated was to construct a simple **look-up table** (LUT). The LUT is a regular grid of latitude vs longitude, in which each cell contains the i,j index of the cell in S whose centre is closest to the latitude-longitude point represented by the cell of the LUT. The LUT is implemented as two congruent arrays of short (2-byte) integers, one array for the i indices and one for the j indices. We found an $\mathcal{O}(N_S)$ means to construct the LUT using a "source-push" algorithm, avoiding brute-force searches of S for each point in the LUT. Queries on the LUT were performed as follows: For each point P, the nearest latitude-longitude grid point P' within the LUT was found (note that the LUT is an invertible grid). The LUT was then used to find the cell in S whose centre is closest to P'. This lookup is a very fast process ($\mathcal{O}(1)$) but is only an approximation to a nearest-neighbour search on P itself. The true nearest neighbour of P was then found by a short gradient-descent algorithm. The cell containing the true nearest-neighbour of P and its immediate neighbours were then searched to find C_{SP} using `Path2D.contains()`. The size (i.e. resolution) of the LUT was chosen to be approximately 3 times finer than the source grid; empirically we have found that this gives an acceptable balance of performance and memory usage. A coarser look-up table would occupy less memory, but would require a lengthier gradient-descent step to refine the initial lookup. A finer look-up table would provide a more accurate lookup at the expense of a larger memory footprint.

5. RESULTS

The results of the above tests are shown in Table 1. Three test datasets were used (UCA, ORCA025 and METEOSAT), described in the table. Of the spatial indexes under test, the look-up table occupies the smallest amount of memory in each case, whereas the kd-tree was consistently the fastest index to build. Query times were consistently of the order of a few microseconds for the kd-tree and LUT for all datasets, with the LUT query being slightly quicker. Results for the R-trees are more complex. Although both R-tree implementations are competitive with other indexes for the smallest dataset (UCA), query times for the other datasets are many times slower. Unexpectedly, query times for the R-tree and

[4]Unfortunately Java does not provide an easy way to calculate the size of an object. We were careful to maximize the possibility that all unused objects had been garbage-collected.

[5]http://jsi.sourceforge.net/

[6]http://www.khelekore.org/prtree/

Table 1: Results of constructing and querying four different types of spatial index based upon three different curvilinear datasets. Results are means of five trials in each case; a single trial for query time consisted of taking the mean of 256×256 queries of the source grid. Standard deviations were small (a few percent) and omitted for clarity.

Dataset	No. cells	Notes	Index	Build time (s)	Size (MB)	Query time (μs)
UCA	7,920	Covers Mediterranean outflow, including large distortions around Strait of Gibraltar (figure 1)	kd-tree	0.03	0.82	5.75
			R-tree	0.43	0.76	8.22
			PR-tree	0.06	0.96	5.48
			LUT	0.13	0.62	4.05
ORCA025	1,472,282	Based on Murray tripolar grid, global coverage at nominal resolution of 0.25 degrees	kd-tree	10.5	163	8.62
			R-tree	81.3	160	513
			PR-tree	332	193	36.7
			LUT	22.0	76.4	5.60
METEOSAT	4,576,701	Geostationary satellite image over Africa / Middle East from Meteosat-7	kd-tree	54.9	564	8.10
			R-tree	247	542	24.0
			PR-tree	3620	824	6.59
			LUT	95.4	295	5.36

PR-tree were faster for the largest dataset (METEOSAT) than for ORCA025. We interpret this as a consequence of the highly-distorted grid cells (in latitude-longitude space) in ORCA025, which we expect to cause greater overlap between bounding rectangles in the R-trees, causing more candidate results to be returned and requiring filtering. The two R-tree variants were by far the slowest indexes to build.

5.1 Application to a Web Map Service

Although the above approaches could be applied to improve the performance of a variety of regridding algorithms, we focus here on the problem of generating raster imagery from complex data in a Web Map Service implementation. It is important that WMS implementations can return images quickly in response to requests from many simultaneous users (the INSPIRE[7] Draft Implementing Rules for View Services v3.0 say that the response time for an 800×600 8-bit image shall be a maximum of 5 seconds, for 20 requests per second).

We compare here the performance of a "source-push" algorithm for generating images with a "destination-pull" algorithm. The source-push algorithm was simply to paint the source grid cells directly onto a canvas, in the manner of a typical visualization package. The destination-pull algorithm iterated through each pixel in the required image, using a look-up table as an index to help to find the source grid cell that contained the position representing the centre of the pixel. The pixel was then coloured according to the value of the source grid cell. The two algorithms produce images that appear identical to the naked eye.

Figure 3 shows the results of applying these algorithms to the three datasets for different requested image sizes. As expected, the performance of the source-push algorithm dependes strongly on the size of the source grid, but only very weakly (if at all) on the size of the destination grid. The performance of the destination-pull algorithm depends strongly on the size of the destination grid.

We see that, for the smallest dataset (UCA), the source-push algorithm significantly outperforms the destination-pull algorithm. For the middle-sized dataset (ORCA025) the destination-pull algorithm is faster for images up to and including 1024×1024 pixels. For the largest dataset (METEOSAT), the destination-pull algorithm is faster for all image sizes under test (up to 2048×2048 pixels).

[7] http://inspire.jrc.ec.europa.eu/

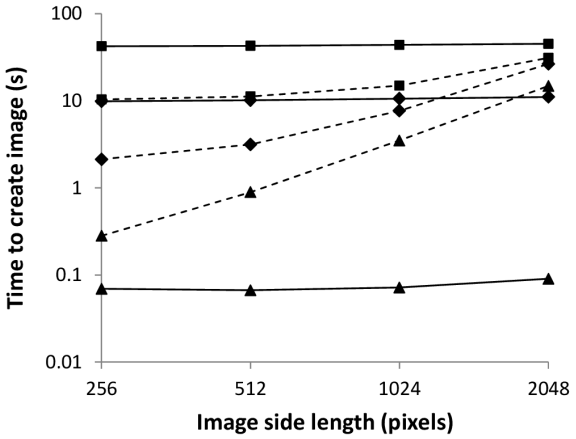

Figure 3: Times to create images of different sizes using source-push (solid lines) and destination-pull (dashed lines) algorithms. Source datasets were UCA (triangles), ORCA025 (diamonds) and METEOSAT (squares); see table 1 and section 5.1. Results are means of five trials in each case. Standard deviations were small (a few percent) and omitted for clarity.

5.2 Comparison with other tools

It is difficult to compare these results fairly and accurately with the performance of other visualization packages. Such packages may perform a variety of tasks in addition to regridding, including the addition of contextual information (axes, colour scale bars, titles) as well as data manipulations such as interpolation. However, in simple trials we found that the Panoply visualization package (version 2.9.4) took around 10 seconds to display an image of ORCA025. Panoply uses a source-push algorithm and this result matches very closely the equivalent result in figure 3, which is perhaps not surprising since Panoply is implemented using many of the same Java libraries. The Matplotlib li-

brary for Python (version 1.0.1), which also uses a source-push algorithm, took around 200 seconds to produce an image of the ORCA025 data at full resolution, or around 30 seconds if the data are subsampled prior to plotting by a factor of six[8]. These results were not sensitive to the size of the generated image, as we would expect from the results from our own source-push algorithm in figure 3.

6. DISCUSSION AND CONCLUSIONS

We have shown that it is possible to regrid large, complex data in a matter of seconds or less, allowing for regridding to be performed "on the fly" in an interactive graphical application. For example, a 256×256 image of a curvilinear source grid containing nearly 1.5 million points (the ORCA025 dataset) can be generated in 2.1 seconds, or 3.1 seconds for a 512×512 image.

This study suggests that different strategies for regridding may be optimal in different situations:

- Where the source grid is small or the destination image is large, a simple source-push algorithm may be sufficient, avoiding the need to create a spatial index. With large source grids, a destination-pull approach may be optimal, particularly if (as is usual) it is not required to view the data at full resolution.

- Where a spatial index is required, the look-up table approach appears optimal in a server situation (such as a Web Map Service), since it gives the fastest query times with the smallest memory footprint. With a persistent server process, the look-up table can be generated once and cached in memory, therefore the creation time is not very important.

- For "one-shot" regridding operations that are not associated with a persistent process, the kd-tree may be the best choice of spatial index since its creation time is much shorter than that of the look-up table, leading to a shorter overall time for the operation. However, its memory footprint is much higher, so the user would need to take care with very large source grids. It would also be necessary to tune the parameters of the kd-tree to the source grid in question.

In addition to its performance for large source datasets, the destination-pull approach has further advantages. If the destination grid is coarser than the source grid (i.e. not all of the data in the source grid is required), the destination-pull approach ensures that it is not necessary to read all source data from disk, assuming that the spatial index has already been constructed. Disk input/output can be a bottleneck to fast visualization and processing. Although in most use cases the destination grid is invertible, the destination-pull approach does not require this to be the case, allowing for easy regridding onto a complex destination grid.

Further performance enhancements are possible. The use of the `java.awt.geom.Path2D.contains()` method to test whether a candidate point falls within a grid cell appears to be expensive. Tuning the spatial indexes to return fewer candidate points would lead to fewer calls to this method, tending to increase overall performance. The R-tree and PR-tree implementations under test were not specifically designed to hold large numbers of items (nevertheless, the PR-tree performed competitively for the largest dataset in terms of query time). There are many ways of constructing [1, 5] and querying [4] R-trees and so it may be possible to choose more efficient algorithms for the situation at hand.

Future work will include the application of these techniques to *unstructured* data (in which space is filled by incongruent shapes, commonly triangles or a mixture of shapes) and *ungridded* data (where there is no topological relationship between data points). Heuristics could be developed to determine the optimal approach (source-push vs. destination-pull) and spatial index to employ in a given situation. In this study, spatial indexes were constructed using spherical coordinates (latitude-longitude), which are not ideal for data that approach the poles. Therefore future refinements will involve dealing with the poles more appropriately through the use of alternative coordinate systems or spatial indexes (e.g. [11]).

[8]This factor was chosen to match the resolution of the data to that of the image, so that the library was not attempting to plot data that would not be seen by the user.

7. ACKNOWLEDGMENTS

We thank the providers of the datasets used as test cases in this study: ASA Science, the NEMO group and EUMETSAT. This work was partially funded under the European FP7 project MyOcean (FP7-SPACE-2007-1). We are grateful to Dr James Anderson and an anonymous reviewer for helpful comments.

8. REFERENCES

[1] H. Alborzi and H. Samet. Execution time analysis of a top-down R-tree construction algorithm. *Information Processing Letters*, 101(1):6–12, 2007.

[2] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The priority R-tree: a practically efficient and worst-case optimal R-tree. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, page 347–358, New York, NY, USA, 2004. ACM. ACM ID: 1007608.

[3] J. Blower, K. Haines, A. Santokhee, and C. Liu. GODIVA2: interactive visualization of environmental data on the web. *Philosophical Transactions of the Royal Society A*, 367:1035–1039, 2009.

[4] A. Corral and J. M. Almendros-Jiménez. A performance comparison of distance-based query algorithms using R-trees in spatial databases. *Information Sciences*, 177(11):2207–2237, 2007.

[5] M. de Berg, J. Gudmundsson, M. Hammar, and M. Overmars. On R-trees with low query complexity. *Computational Geometry*, 24(3):179–195, 2003.

[6] J. de la Beaujardière (ed.). OpenGIS Web Map Service Implementation Specification, version 1.3.0, Mar. 2006.

[7] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, June 2007.

[8] P. W. Jones. First- and second-order conservative remapping schemes for grids in spherical coordinates. *Monthly Weather Review*, 127(9):2204–2210, Sept. 1999.

[9] R. J. Murray. Explicit generation of orthogonal grids for ocean models. *Journal of Computational Physics*,

126(2):251–273, July 1996.

[10] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison Wesley, Sept. 1989.

[11] A. Szalay, J. Gray, G. Fekete, P. Kunszt, P. Kokul, and A. Thakar. Indexing the sphere with the hierarchical triangular mesh. TechReport MSR-TR-2005-123, Microsoft Research, Aug. 2005.