Scalable and Memory-Efficient Approaches for Spatial Data Downscaling Leveraging Machine Learning

Carlos Ribeiro

Abstract

In the context of spatial analysis, spatial disaggregation or spatial downscaling are processes by which information at a coarse spatial scale is translated to finer scales, while maintaining consistency with the original dataset. Fine-grained descriptions of geographical information is a key resource in fields such as social-economic studies, urban and regional planning, transport planning, or environmental impact analysis. One such method for spacial disaggregation is the dissever algorithm, a hybrid approach that combines pycnophylactic interpolation and regression-based dasymetric mapping, leveraging auxiliary data to increase desegregation precision. Implemented in R, the current dissever version assumes a sequential execution model and the usage of smaller-than-ram computational sequences, limiting the resolution and scale of usable datasets. This paper presents a variant of the dissever algorithm, called S-Dissever (Scalable Dissever) suitable for efficiently executing in parallel environments, taking advantage of modern multi-core architectures, while using secondary memory to scale out computations that exceed RAM capacity. Experimental evaluations on the S-Dissever demonstrate the feasibility of the desegregation procedure while using high-resolution auxiliary data at significant geographical extents, as well as notable speedups obtained by the utilization of multi-core environments.

1 Introduction

Spatial information data is widely available, ranging multiple sources from different fields like socio-economic activities or topographic related information. Despite this availability, is not unusual to find data often collected or released only at a relatively aggregated level. This is usually the case in information regarding human population distributions, coincidentally one of the most important material in formulating informed hypothesis in the context of population-related social, economic, and environmental issues. Population data is commonly released in association to government instigated national census, where the geographical space is sub-dived into discrete sub administrative areas, upon where population counts are aggregated. There are several reasons why population data aggregated to fixed administrative areas is not an ideal form for representing information about population density. First, this type of representation suffers from the modifiable areal unit problem, which states that analysis performed on data aggregates by administrative units may vary depending on the shape and size of the units rather than capturing the theoretically continuous variation in the underlying population (Lloyd, 2014). Second, the spatial resolution of aggregated data is variable and usually low. Although, highly-aggregated data might be useful for broad-scale assessments, has the danger of masking important local hotspots by smoothing out spatial variations throughout the spatial plane. Given the aforementioned limitations, the use of high-resolution population grids is a preferable solution to deliver population data, allowing different types of analysis in regard to the same source data, as all cells in a population grid have the same size and are stable throughout time. Population grids are built from national census data through the application of spatial disaggregation methods which range in complexity from simple mass-preserving areal weighting (Goodchild et al., 1993), to intelligent dasymetric weighting schemes.

One such method of intelligent dasymetric disaggregation is the Dissever algorithm (João Monteiro, 2017), leveraging regression analysis to combine multiple sources of ancillary data. Dissever is implemented in R, an open source programming language and software environment for statistical computing and graphics (Team, 2000). The the current version of the dissever algorithm, also referenced throughout this article as the baseline version, assumes (1) a sequential execution model and (2) the usage of smaller-than-ram computational sequences. As such, it can currently only be applied to small datasets.

In this dissertation I designed and implement a variant of the dissever algorithm, called S-Dissever (Scalable Dissever) suitable for efficiently executing in (1) a parallel environment, to take advantage of modern multi-core architectures, while (2) taking advantage of secondary memory to scale out computations that exceed RAM capacity. More in detail, this dissertation makes the following contributions: (1) Profiling of existing implementation to identify key bottlenecks; (2) Design of parallelized versions of the various dissever's sub-components, leveraged by the usage of R packages specifically design for high-performance computing; (3) Introduction of additional data structures in external memory, complemented by data partition schemes aiming at a sustainable usage of RAM, reduction of disk access frequency, as well to utilize the full capability of the multi-core parallelization; (4) Evaluation based on large and realistic data sets regarding the disaggregation of historical population census data, from Great Britain up to a resolution grid of 0.0008 decimal degrees (DD) or, 90 meters at the equator obtained by topological auxiliary variables like elevation and land cover. Every performed test with the high-resolution data was able to finish successfully all the necessary computations, and when a multi-core system was available, some components were able to reach speedups up to 12 times without exhausting the available memory.

The rest of this document is organized as follows: Section 2 presents fundamental concepts and important related work. Section 3 describes the baseline version of the dissever algorithm. Section 4 details the new implementation of the S-Dissever algorithm. Section 5 presents analysis and results from the work developed. Finally, Section 6 presents the main conclusions, and highlights possible directions for future work.

2 Concepts and Related Work

The simplest spatial disaggregation method is perhaps mass-preserving areal weighting, whereby the known population of source administrative regions is divided uniformly across their area, in order to estimate population at target regions of lower spatial resolution (Goodchild et al., 1993). The process usually relies on a discretized grid over the administrative regions, where each cell in the grid is assigned a population value equal to the total population over the total number of cells that cover the corresponding administrative region. Pycnophylactic interpolation is a refinement of mass-preserving areal weighting that assumes a degree of spatial auto-correlation in the population distribution (Tobler, 1979a). This method starts by applying the mass-preserving areal weighting procedure, afterwards smoothing the values for the resulting grid cells by replacing them with the average of their neighbors. The aggregation of the predicted values for all cells within a source region is then compared with the actual value, and adjusted to keep the consistency within the source regions. The method continues iteratively until there is either no significant difference between predicted values and actual values within the source regions, or until there have been no significant changes from the previous iteration.

Dasymetric weighting schemes are instead based on creating a weighted surface to distribute the known population, instead of considering a uniform distribution as in mass-preserving areal weighting. The dasymetric weighting schemes are determined by combining different spatial layers (e.g., slope, land versus water masks, etc.) according to rules that relate the ancillary variables to population counts.

While some weighting schemes are based on expert knowledge and manually-defined rules, recent methods leverage machine learning to improve upon this approach (Stevens et al., 2015; João Monteiro, 2017; Lin et al., 2011), often combining many different sources of ancillary information (e.g., satellite imagery of night-time lights (Stevens et al., 2015; Briggs et al., 2007), LIDAR-derived building volumes (Sridharan and Qiu, 2013; Zhao et al., 2017), or even density maps derived from geo-referenced tweets (Goodchild et al., 1993), from OpenStreetMap points-of-interest (Bakillah et al., 2014), or from mobile phone data (Deville et al., 2014)).

2.1 Tools for Scalable Applications Development in R

R is a system for statistical computation and graphics. It provides, among other things, a programming language, high level graphics, interfaces to other languages and debugging facilities. R's native support for efficient and scalable computation is limited, although there is a list of CRAN packages[1] design to extend the existing base functionalities. This section discusses some of the tools used in this project, regarding parallel computation on multi-core environments and out-of-memory data manipulation.

2.1.1 Multi-Core Computing in R

In regard to explicit parallel computing, starting from release 2.14.0, R includes a package called *parallel*[2], incorporating revised copies of CRAN packages *multicore* and *snow* (Simple Network of Workstations). The *parallel* package is designed to exploit multi-core and distributed environments, although this project foucus is aimed at a multi-core sotution.

Two types of clusters are available in the *parallel* package. One, the *FORK* type cluster is only available for Unix like system, and the second one, the *PSOCK* type cluster can be used across all platforms, including Windows.

The *FORK* type cluster comes with the advantage of nodes linking to the same address space as the master node, as it creates multiple copies of the current R session, greatly simplifying memory management in the cluster. Only when a node performs a write operation into an object living in the master node memory space, is the content effectively copied to that node's memory space. However, for this project the *FORK* type cluster was not an option aiming at a multi-platform application.

The used option was, the *PSOCK* type cluster, where every node is started as a new R session. This requires an initialization process where all the required libraries and data structures necessary for the parallel computations must be first loaded.

Multiple methods are offered in the *parallel* package regarding data manipulation and function calling in the context of the cluster distributed environment.

One of such methods is a parallelized version of the *lapply* method, named *parlapply*, were it is possible to apply a given user defined function f to an array x in a parallel fashion, obtaining *length(x)* results, processed by the different nodes in the cluster.

Another possibility to utilize the R cluster capabilities is available through the R package *foreach*[3], where a substitute for the traditional *for* loop is presented. In this scenario, each iteration of the loop is performed in parallel by the available R sessions in the cluster, where the output of each iteration can be combined in various forms by using the *.combine* argument. Caution must be made in the usage of the *foreach* package regarding memory usage, as all the variables used in the loop are first loaded to all nodes in the cluster, including arrays, vectors and lists where only a few elements are relevant to a given node computation. This can be avoided by correctly initializing the cluster, with only the necessary data.

In the initialization procedure, methods like *clusterExport* or *clusterEvalQ* can be very useful. The *clusterExport* method takes any kind of R object as argument and assigns their values to variables of the same name in the global environment of each node. The *clusterEvalQ* evaluates a literal expression on each cluster node, useful to library initialization and debugging purposes for example.

2.1.2 Secondary memory in R

Secondary memory, besides providing access to arbitrarily large data structures, potential larger than RAM, also provides shared access to different R session, in order to support efficient parallel computing techniques. Nonetheless, attention must be made in the use of such packages, as using a disk storage layout different from R's internal structure representation can render most CRAN packages incompatible.

[1] `https://cran.r-project.org/web/views/HighPerformanceComputing.html`
[2] `https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf`
[3] `https://CRAN.R-project.org/package=foreach`

Packages like *ff*[4] and *bigmemory*[5] (Kane et al., 2013) provide efficient memory manipulation by partitioning the data and swapping it in and out of the main memory, when required. The *RSQLite*[6] package enables the usage *SQLite* database engine in R and can also be used as secondary memory.

The *bigmemory* package is aimed at the creation, storage, access and manipulation of large matrices whose interactions are exposed via the R object *big.matrix*. The *big.matrix* object points to data structures implemented in C++, however the usage is very much like a traditional R matrix object effectively shielding the user from the implementation. The application domain of the *big.matrix* object can be restrictive, however in problems of spacial data analysis, this data representation can be very useful.

The *ff* package, extended by the *ffbase*[7] package is a much more general purpose tool in regard to secondary memory usage. It makes available objects supporting data structures for vectors, matrices and arrays, as well as a ffdf class, mimicking the traditional R *data.frame*. While the raw data is stored on a flat file possibly in disk, meta information about the atomic data structure such as it's dimension, virtual storage mode i.e., *vmode*, or internal length are stored as an ordinary R object. The raw flat file data encoding is always in native machine format for optimal performance and provides several packing schemes for different data types such as logical, raw, integer and double. The extension added by the *ffbase* package, brings a lot of functionality, mostly aimed at the *ffdf* objects. This functionalities include among others, math operations, data selection and manipulation, summary statistics and data transformations.

All ff objects have associated a finalizer component. This component manages the life cycle of permanent and temporary files. When (1) the garbage collector is called, (2) the rm() command is executed or (3) the current R session is terminated the finalizer determines if the allocated resources are eliminated. The scenario where only the in-memory components are removed and the files in persistent memory are preserved is also contemplated.

The use of secondary memory tools in R alone does not avoid the traditional out-of-memory problems associated with large data computations. One might wrongly assume that in a given application, by only replacing the existing data structures with secondary memory capable substitutes all the problems will disappear. However, without a partitioning strategy where the data in disk is swapped to ram in a sustainable manner, the same out-of-memory errors will emerge in a similar fashion. The *ff* package however, contains chucking methods design to this purpose in which a *ffdf* can be accessed via multiple indexes pairs, generated in the basis of a user defined chunk size. This allows the in-disk data to be loaded in chunks to memory. Nevertheless is the user's responsibility to adapt existing computations to implement this data partition scheme and guarantee that the final results are consistent with a all-in-ram approach.

3 Baseline Dissever Implementation

The dissever algorithm is a hybrid approach that combines pycnophylactic interpolation and regression-based dasymetric mapping(Monteiro, 2016). In brief, we have that pycnophylactic interpolation is used for producing initial estimates from the aggregated data, which that are then adjusted through an iterative procedure that uses regression modeling to infer population from a set of ancillary variables. The general procedure is detailed next, trough an enumeration of all the individual steps that are involved:

1. Produce a vector polygon layer for the data to be disaggregated by associating the population counts, linked to the source regions, to geometric polygons representing the corresponding regions;

2. Create a raster representation for the study region, with basis on the vector polygon layer from the previous step and considering a given target resolution, (e.g., 30 meters per cell or 1 arcsec). This raster, referred to as T^p, will contain smooth values resulting from a pycnophylactic interpolation procedure (Patel et al., 2017). The algorithm starts by assigning cells to values computed from the

[4]`https://CRAN.R-project.org/package=ff`
[5]`https://CRAN.R-project.org/package=bigmemory`
[6]`https://cran.r-project.org/package=RSQLite`
[7]`https://CRAN.R-project.org/package=ffbase`

original vector polygon layer, using a simple mass-preserving areal weighting procedure (i.e., we redistribute the aggregated data with basis on the proportion of each source zone that overlaps with the target zone). Interactively, each cell's value is replaced with the average of its eight neighbors in the target raster. We finally adjust the values of all cells within each zone proportionally, so that each zone's total in the target raster is the same as the original total (e.g., if the total is 10% lower than the original value, we increase the value of each cell by a factor of 10%). The procedure is repeated, until no more significant changes occur. The resulting raster is a smooth surface corresponding to an initial estimate for the disaggregated values;

3. Overlay n different rasters P^1 to P^{n-1}, also using a resolution of 30 meters per cell, on the study region from the original vector layer and from the raster produced in the previous step (e.g., information regarding modern population density, historical land coverage or terrain elevation). These rasters will be used as ancillary information for the spatial disaggregation procedure. Prior to overlaying the data, the n different raster data sources are normalized to the resolution of 30 meters per cell, through a simple interpolation procedure based on taking the mean of the different values per cell (i.e., in the cases where the original raster had a higher resolution), or the value from the nearest/encompassing cell (i.e., in the cases where the original raster had a lower resolution);

4. Create a final raster overlay, through the application of an intelligent dasymetric disaggregation procedure based on disseveration, as originally proposed by Malone et al. (2012), and leveraging the rasters from the previous steps. Specifically, the vector polygon layer from Step 1 is considered as the source data to be disaggregated, while raster T^p from Step 2 is considered as an initial estimate for the disaggregated values. Rasters P^1 to P^{n-1}, are seen as predictive covariates. The regression algorithm used in the disseveration procedure is fit using all the available data, and applied to produce new values for raster T_p. The application of the regression algorithm will refine the initial estimates with basis on their relation towards the predictive covariates, this way *dissevering* the source data;

5. We proportionally adjust the values returned by the downscaling method from the previous step for all cells within each source zone, so that each source zone's total in the target raster is the same as the total in the original vector polygon layer (e.g., again, if the total is 10% lower than the original value, increase the value of each cell in by a factor of 10%).

6. Steps 4 and 5 are repeated, iteratively executing the disseveration procedure that relies on regression analysis to adjust the initial estimates T^p from Step 2, until the estimated values converge (i.e., until the change in the estimated error rate over three consecutive iterations is less than 0.001) or until reaching a maximum number of iterations (i.e., 100 iterations).

Notice that the previous enumeration describes the procedure in general terms. Also, the presented steps do not follow the exact same order as the original implementation in R for clarity purposes.

In Algorithm 1 a detailed pseudo-code is presented outlining the principal logical components in the baseline Dissever implementation. Here, the *pycno* method stands for pycnophylactic interpolation procedure, the *fineAuxData* variable is the raster stack containing all the available auxiliary variables and the *shapefile* variable is the set of polygons representing all the geographical regions containing the information to be desegregated.

3.1 Pycnophylactic Interpolation Baseline Implementation

Pycnophylactic Interpolation, also referred to as pycnophylactic reallocation, is a spatial disaggregation method that solely relies on the source data itself and on some intrinsic properties of population distributions, such as the fact that people tend to congregate, leading to neighboring and adjacent places being similar regarding population density (Tobler, 1979b). Tobler's (1979b) *pycnophilatic interpolation* method starts by generating a raster map i.e., *gridCounts*, containing a new grid representation based on the original data set, analogous to the first iteration represented in Figure 1. The creation of *gridCounts* takes

Algorithm 1: Baseline Dissever Implementation

```
input  : A Shapefile and a target resolution
output: A smooth count surface raster produced at the same extent of the shapefile at the target
        resolution
pycnoRaster ← pycno(shapefile, fineAuxData)
coarseRaster ← rasterize(shapefile, targetResolution)
coarseIdsRaster ← rasterize(shapefile, targetResolution)
trainingTable ← preProcessingTrainingData(pycnoRaster, coarseRaster, coarseIdsRaster)
repeat
    fit ← train(trainingTable$results, fineAuxData)
    trainingTable$results ← predict(fit, fineAuxData)
    trainingTable$results ← massPreservation(trainingTable)
until;
```

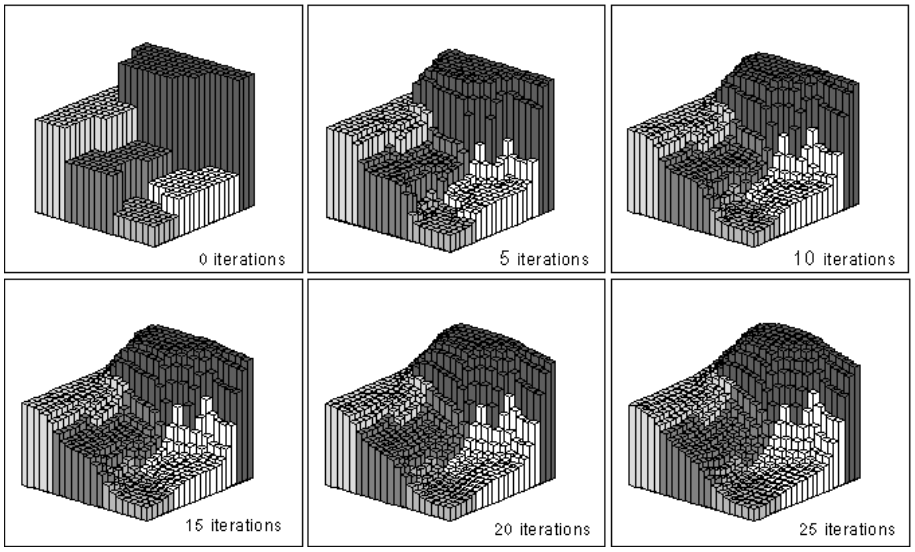


Figure 1: Illustration of Tobler's pycnophylactic method for geographical regions.[8]

into account the property of mass-preserving areal weighting, where the sum of all the cells (i.e., target zones) representing a source zone must equal to the amount of that original source zone. The values of the *gridCounts* are then smoothed, by replacing them with the average of their four neighbours, resulting in a raster such as the one from tle last iteration in Figure 1. The predicted values in each source zone on the newly generated raster map are compared with the actual values from the original source zones, and adjusted to meet the pycnophylactic condition of mass-preservation. This is done by multiplying a weight with all the cells belonging to the source zone deviating from the original values. Several iterations of the previous referenced algorithm may be required, until a smooth surface is achieved. Starting from the replacing of the cells with the average of their four neighbours, the whole process can be repeated until some convergence criterion is met (i.e., the overall value of the cells remains fairly unmodified in two consecutive iterations).

To better understand the actual implementation, the pseudocode in algorithm 2 illustrates this process in grater detail.

The first steep aims at creating a matrix representation of the variable to be desegregated. This matrix *gridCounts* is scaled to the target resolution, defined by *cellSize* spanning the entire target region. The *Extent* method returns the size of the frame containing the entire geographical region defined in the *shapefile*, typically in decimal degrees. This means that each cell in *gridCounts* represents the smallest

Algorithm 2: Baseline Pycnophilatic Interpolation

```
input  : A Shapefile and a target resolution
output: A smooth count surface raster produced at the same extent of the shapefile at the target
        resolution
emptyGrid ← SpatialGrid(cellSize, extent(shapefile))
gridIds ← IdPopulate(emptyGrid, shapefile)
gridCounts ← populateCounts(shapefile, gridIds)
stopper ← max(gridCounts) $\times 10^{-converge}$
repeat
    oldGridCounts ← gridCounts
    avgSmoothing(gridCounts)
    correctionA(gridCounts)
    negativeRemoval(gridCounts)
    correctionB(gridCounts)
until max(abs(oldGridCounts − gridCounts)) <= stopper;
```

spatial unit in the desegregation procedure.

As the geographical region defined in the *shapefile* is divided into several sub-administrative regions, each one associated with a different count value, it is necessary to keep track of which cells belong to which regions. To this end a twin matrix named *gridIds* must also be generated.

In *gridIds*, the number of cells is identical to the *gridCounts* however the stored values are the ids identifying the zones to which a given cell is associated.

The process to access and manipulate data in *gridCounts* is done in two steps. First a logical comparison between *gridIds* and a specific *zoneId* is performed, resulting in a boolean mask matrix i.e. *zoneSet* where only cells associated with *zoneId* are set as a true boolean value. Second the values in *gridCounts* can be accessed in the following fashion: *gridCounts*[*zoneSet*], setting up the scheme for data manipulation across the different sub-routines in this process.

Despite being an easy to read and straightforward method, this data access scheme incurs into high overhead as the logical comparison between *gridIds* and a specific *zoneId* must be performed for every read or write operations in order to obtain a set of logical indices (i.e. boolean mask matrix) to subset the relevant data. Modifications to this data access scheme are further discussed in Section 4.1

In order to populate the *gridCounts* matrix the *PopulateCounts* method, using the previously described data access scheme, sets the value of the cells associated to a given *zoneId* to the division of the total count regarding the *zoneId* by the total number of cells contained in that area, i.e.:

$$gridCounts[zoneSet] = \frac{shapefile.counts[zoneId]}{numberCells(zoneSet)} \tag{1}$$

This ensures that the pycnophylactic condition of mass-preservation is fulfilled, concluding the first step of the interpolation.

The second step consists of an iterative process of smoothing i.e., replacing the values of each cell in *gridCounts* with the average of their four neighbours. This is represented in algorithm 2 by the *AvgSmoothing* method.

One inevitable consequence of the smoothing procedure is the deviation from the original count in the different sub-regions, violating the pycnophylactic mass-preservation condition. Due to this deviation corrections must be performed.

The first correction i.e., *CorrectionA* aims at summing or subtracting the difference in counts from the newly smoothed regions to the original values:

$$correction = \frac{shapefile.counts[zoneId] - sum(gridCounts[zoneSet])}{numberCells(zoneSet)} \tag{2}$$

$$gridCounts[zoneSet] = gridCounts[zoneSet] + correction \tag{3}$$

One consequence from *CorrectionA* is the possibility of some cells storing negative values. To this end, the method *NegativeRemoval* defined in algorithm 2 is applied to the *gridCounts* matrix replacing all negative values with zero. This once again compromises the pycnophylactic mass-preservation condition, requiring a second correction procedure, i.e., *CorrectionB* defined in the following way:

$$correction = \frac{shapefile.counts[zoneId]}{sum(gridCounts[zoneSet])} \tag{4}$$

$$gridCounts[zoneSet] = gridCounts[zoneSet] * correction \tag{5}$$

The iterative process comes to an end when the maximum value from the difference between two consecutive iterations, reaches the stopping criterion.

3.2 Pre-Processing Training Data Baseline Implementation

This sub-section describes the data processing phase that takes place before the training of the regression model. The input data in this stage consists in: (1) the result from the pycnophylactic interpolation (i.e., a raster layer data structure) and (2) a raster stack (i.e., a list of multiple raster layers) composed up to n auxiliary variables. Both (1) and (2) are scaled to the same resolution, and have matching extents, meaning a perfect overlap between the rasters. Usually the resolution of (1) and (2) is defined by the minimum resolution available by the auxiliary variables i.e., the target resolution.

The first step in this process is to map the available data into groups, where each group contains all the relative data regarding a sub-geographical region associated to a given *zoneId*.

In order to achieve this, the referenced data (i.e., the raster layers) is parsed into a table data-frame, named *trainingTable*. This table contains an entry for each cell of the fine resolution data. The columns code information associated with each cell, including geographical coordinate position, dependent and independent variables, and a sub-geographical region identifier i.e., the *zoneId* to which that cell belongs. More specifically the columns are respectively: the x and y coordinates, the value of the pycnophylactic interpolation (i.e., the dependent variable), one column for each auxiliary variable (i.e., the independent variables) and the grouping id *zoneId*. The procedure into which the data contained in the raster layers is extracted to the *trainingTable* is by means of directly conversion to R *data.frames*. Then, the relevant columns to be exported are selected and copied to the *trainingTable*. This requires that the referenced raster layers possibly in disk do not exceed the available ram memory.

After all available data is lined up, data points containing NA data, (i.e., entries in the *trainingTable* that contain one or more none-existing value) are removed from.

In order to populate the *zoneId* column a rasterization process (described in grater detail on subsection 3.2.1) is performed so that a raster layer is created from the *zoneIds* specified in the *shapefile*. Then, a extraction process takes place, aiming to populate the column *zoneId*. It is through this aggregation component that the mass preserving process is performed in a *group by sum* SQL style computation.

If the data sampling option is chosen by the user, this is done so by a percentage parameter p, ranging in values from $[1, 0[$. Internally $p \times numberRows$ random entries from the previously described table are selected, defining the data subset that will actually be used to train the regression model in the next steps.

3.2.1 Rasterization Baseline Implementation

In the rasterization process, values associated with spatial data object (i.e., points, lines or polygons) are transferred to raster cells. When the source data comes in form of a *SpatialPolygonDataFrame*, the polygon value is transferred to a raster-cell if it covers the center of the cell. In the *dissever* algorithm this is achieved by an off the shelf solution i.e., an atomic method available in the *raster* package named *rasterize.* This method does not support parallel execution in its native implementation. In a side note, a distinction between the rasterization process and the pycnophylactic interpolation algorithm must be made. The rasterization process is not a sub-component of the pycnophylactic processes as one may think. Despite their output similarities (e.i., their both raster layers describing a geographical region) the cell values of each raster vary in significance. In the rasterization result, each cell contains the overall value associated with the sub-region to which that cell belongs. An analogy can be made to a snapshot taken from the original shapefile to which the rasterization process is derived. On the other hand, in the pycnophylactic raster result, each cell holds meaning on its own. This means that the value contained in each cell is specific only to the area covered by that same cell.

3.3 Regression Baseline Implementation

In terms of the implementation of the regression models, internally the dissever baseline version is using the caret[9] package. The caret package, short for classification and regression training, contains numerous tools for developing different types of predictive models, facilitating the realization of experiments with different types of regression approaches in order to discover the relations between the target variable and the available covariates.

In standard linear regression, a linear least-squares fit is computed for a set of predictor variables (i.e., the covariates) to predict a dependent variable (i.e., the disaggregated values). The well known linear regression equation corresponds to a weighted linear combination of the predictive covariates, added to a bias term. On the other hand, models based on decision trees correspond to non-linear procedures based on inferring a flow-chart-like structure, where each internal node denotes a test on an attribute, each branch represents the outcome of a test, and each leaf node holds a target value. Decision trees can be learned by splitting the source set of training instances into subsets, based on finding an attribute value test that optimizes the homogeneity of the target variable within the resulting subsets (e.g., by optimizing an information gain metric). This process is repeated on each derived subset in a recursive manner. Recently, ensemble strategies for combining multiple decision trees have become popular in machine learning and data mining competitions, often achieving state-of-the-art results while at the same time supporting efficient training and inference through the parallel training of the trees involved in the ensemble (Banfield et al., 2007).

Random forests operate by independently inferring a multitude of decision trees at training time, afterwards outputting the mean of the values returned by the individual trees (Breiman, 2001). Each tree in the ensemble is trained over a random sample, taken with replacements, of instances from the training set. The process of inferring the trees also adds variability to each tree in the ensemble by selecting, at each candidate split in the learning process, a random subset of the features. The gradient boosting approach operates in a stage-wise fashion, sequentially learning decision tree models that focus on improving the decisions of predecessor models. The prediction scores of each individual tree are summed up to get the final score. Chen and Guestrin introduced a scalable tree boosting library called XGBoost, which has been used widely by data scientists to achieve state-of-the-art results on many machine learning challenges (Chen and Guestrin, 2016). XGBoost is available from within the caret package, and we used this specific implementation of tree boosting.

All three aforementioned procedures (i.e., linear regression modeling, random forests, and tree boosting) process each grid cell independently of the others. Those values (i.e., the dependent and independent variables) available in the *trainingTable* are then used to fit the required model and compute the predicted desegregation. Then, the prediction is proportionally adjusted to maintain the pycnophylactic mass preservation condition and used again as an initial estimate in the next train-prediction iteration until the estimated values converge (i.e., until the change in the estimated error rate over three consecutive

[9]`http://cran.r-project.org/web/packages/caret`

Algorithm 3: Scalable Pycnophilatic Interpolation

```
input  : A Shapefile and a target resolution
output: A smooth count surface raster produced at the same extent of the shapefile at the target
        resolution

parIndicesComputation(shapefile, cellSize,)
bigGridCounts ← parPopulateCounts(shapefile, gridIds, DB)
stopper ← parMax(bigGridCounts) $\times 10^{-converge}$
repeat
    bigOldGridCounts ← bigGridCounts
    parAvgSmoothing(bigGridCounts)
    parCorrectionA(bigGridCounts)
    parNegativeRemoval(bigGridCounts)
    parCorrectionB(bigGridCounts)
until parMax(abs(bigOldGridCounts − bigGridCounts)) <= stopper;
```

iterations is less than 0.001) or until reaching a maximum number of iterations (i.e., 100 iterations).

4 Scalable and Memory Efficient Dissever

In the baseline Dissever algorithm, limitations regarding scalability are mainly associated with the usage of data sets larger than RAM. This renders the utilization of some data sets an impossibility as the execution will stop due to system restrains in memory allocation.

The single thread implementation used by the standard R libraries can also be addressed as a bottleneck although not a critical one, as it does not prevent the conclusion of the process. However, as the size of the datasets used increase so does the time it takes to finish all the necessary computations. In this scenario, the non-utilization of all the available cores in a machine can be a huge waste of time and resources.

In the process of replacing key code components in any software, attention must be made so that the meaning of the output is not compromised in any manner. To this end, a good practice is to probe all intermediate results at every relevant step of the process and store them for later comparison. A good comparison strategy must also be devised so that any difference in the results can be made visible to the programmer. This section describes all the changes performed to the baseline version of the dissever algorithm as well as some implementation choices that lead to the final result of the S-Dissever.

4.1 Salable Pycnophylactic Interpolation

This new implementation is described in general terms by the Algorithm 3.

The first step to achieve scalability in this process was to replace the *gridCounts*, previously in-memory to an in-disk representation, named *bigGridCounts*. This was done via the *big.matrix* data structure, supported by the *bigmemory* package. This new data representation serves not only to handle the load of larger-than-ram datasets but to enable shared data manipulation across several R sessions (i.e. workers in a R cluster).

Since the *big.matrix* underlying data structure is represented as an array of pointers to column vectors, it is preferable to read and write fewer columns segments than row shorter segments, in order to avoid random file access. This is an important detail that will be taken into account when manipulating this new data structure.

The next step to achieve a scalable and efficient pycnophylactic interpolation was by means of some architectural changes in the initial version of the algorithm. This modification is associated with the

Algorithm 4: Scalable Indices Computation

```
input  : A Shapefile and a target cellSize fixing the resolution
output: A in-memory database mapping zoneIDs to their respective cell indices

Master:
chunkSizeBytes ← $\frac{\text{availableMemory}}{\text{numberWorkers}}$ × conservativeWeight
maxCellsToLoad ← chunkSizeBytes / cellSizeBytes
sectionsList ← computeSections(maxCellsToLoad, cellResolution, numberWorkers, extent(shapefile))
foreach worker ∈ cluster do
    worker ← workerSection ←sectionsList [$i$]
end

Worker:
emptyGrid ← SpatialGrid(cellSize, extent(workerSection))
gridIds ← IdPopulate(emptyGrid, shapefile)
foreach zoneId ∈ setZoneIds do
    storeInDB(zoneId, logicalToNumericIndice(gridIds == zoneId))
end
```

previously described data access scheme regarding how the cell values of a given *zoneId* were written or read in the *gridCounts* matrix, now replaced by the *big.matrix* data structure.

In the first version, in order to index the cells containing the values of a given *zoneId* in the *gridCounts* matrix, a matrix of boolean values (i.e., the logical set of indices *zoneSet*) obtained by means of a logical comparison between the matrix *gridIds*, and the integer *zoneId* was computed for each read or write operations. This process is used in the methods *PopulateCounts*, *CorrectionA* and *CorrectionB*. Effectively incurring into repeated work, aggravated in the situations where the required convergence needs more than one iteration to be achieved.

The implemented alternative consists of a new initial phase named *parIndicesComputation*, detailed in the Algorithm 4. In this new phase, indices for each cell associated with a given *zoneId* are computed and stored in a *SQLite* database for later use. For storage efficiency purposes, the logical index representation is ditched in favor of a numerical representation. This process is leveraged by parallel computation. In order to allow parallel computation in this procedure, the entire geographical region must be first splitted into N different segments, where only the geographical coordinates of each segment are uploaded to the nodes in the cluster. Following this partitioning, each node will generate their own *gridIds* matrix, spanning only its assigned partition of the geographical region. Then, the mapping associating *zoneIds* values and their respective x, y indices is performed and stored in the database.

In this process of populating the database, the criteria into which the geographical region is divided i.e., the size of N, takes into account three different factors: (1) The size in bytes that a single cell in the *gridId* requires to be computed; (2) The number of nodes in the cluster; (3) The available RAM memory. This ensures that each worker can safely create its own *gridId* without exceeding the available memory, as stated in Algorithm 4 in the *chunkSizeBytes* initialization.

A conservative weight, taking values $\in [1, 0[$ is used no increase the size of N, effectively decreasing the size that each geographical partition takes in memory, ensuring a sustainable use of the available memory. This splitting of the geographic region takes no regard for the boundaries of the sub-administrative regions associated to the different *zoneIds*, since it is not necessary that each workers knows were a given region starts or ends.

Furthermore, the memory allocation procedures (i.e., when a worker try's to generate a *gridIds*) are protected by a try-catch clause in the eventuality of memory allocation failure, in which case the process is repeated after the garbage collector is executed. When all indices the *zoneId* indices are effectively stored in the database, all unnecessary data structures used in the process (like the *gridId* matrix itself) are dereferenced by the *rm()* command, in order to increase the available memory.

This new database in disk enables a faster data manipulation scheme in the *bigGridCounts* matrix. Since the time to retrieve a set of indices from disk is less than the time to compute the same set from the *gridIds* matrix this approach has proven itself to be a preferable option.

All the sub-components in this new version of a scalable parallel pycnophilatic interpolation will make use of this new data access scheme as well as the new in-disk data representations.

Regarding parallel computation, and how it was performed, there are effectively two distinct types of data partitioning, in order to distribute work among the available workers in the cluster.

The first partition scheme is used in the *parPopulateCounts*, *parCorrectionA* and *parCorrectionB* sub-components. These sub-components need to perform operations in cell sets regarding specific *zoneIds* in the *bigGridCounts* matrix. To achieve this, the newly developed data access scheme, leveraged by the pre-computed mapping stored in the database is used. In order to distribute work among the cluster in this scenario, the total set of *zoneIds* (i.e., *setZoneIds*) is splitted into n different sets, where n is the number of workers in the cluster.

Then, each set is attributed to a given node, effectively assigning a collection of sub-geographical regions to each worker. Considering that all sub-geographical regions have well-defined borders and don't overlap each other it is safe to assume that there is no need for synchronicity in the writing procedure since each worker will be writing in their well-defined set of cells in the *bigGridCounts* matrix.

It is possible, however, that the number of cell regarding a specific *zoneId* is larger than the available ram memory. This might be a problem if a worker is trying to perform some computation that requires reading all cells (e.g., performing corrections in the set of cells of a particular zoneId) as it happens in sub-components *parCorrectionA* and *parCorrectionB*.

To avoid problems like this, a new level of partitioning must be introduced at the level of the *zoneId*. This means that only a n number of cells are loaded into memory at any given time, until all cells have been processed. For example in sub-components *parCorrectionA* and *parCorrectionB* it is necessary to sum all cells values in a given zone in order to compute the necessary correction. This is achieved by only fetching a n number of row at each time from the database until all rows have been fetched. The procedure for determining n is similar to the procedure for population the database. In this case n only allows fetching a number of cells that prefers a certain size in memory.

The second partition scheme is used by the sub-components *parMax*, *parAvgSmoothing* and *parNegativeRemoval*. This scheme follows a y-axis oriented partitioning. In other words the *bigGridCounts* is sliced along the y-axis, resulting in M different segments, where each segment is only defined by its initial an final column value. The y-axis orientation is preferred to make a better use of the underlying representation of the *big.matrix* data structures, due to reasons previously appointed. The value of M is also computed in function of the maximum size that a worker is allowed to upload from disk to memory.

For the sub-components *parMax* and *parNegativeRemoval* not much change from the perspective of a worker in relation to the first version of the algorithm. However, for the component *parAvgSmoothing*, this is not the case. Consistency and synchronization issues arise when splitting the *bigGridCounts* matrix in sequential continues partitions. More specifically, to compute the neighbors average of the cells contained within the border column of a partition, the values of the of the next partition border column will also be needed. As there is no way to tell if the worker responsible for the next partition has already performed the average operation or not, errors may be introduced in the process. The solution is to create a new empty *bigGridCounts* matrix and delegate the *oldBigGridCounts* to a read-only status. This allows the workers to safely read the content of the previously iteration. In addition, each worker partition will have to overlap the next workers partition by one column, in order to preserve consistency in the final results.

4.2 Parallel Rasterization

The rasterization method available in the *raster* package used by the *dissever* algorithm is an atomic component.

Algorithm 5: Scalable Rasterization

```
input  : A Shapefile and a target resolution
output: A raster representation of the shapefile at the target resolution

Master:
polygonsList ← features(shapefile)
polygonSets ← split(polygonsList, numberWorkers)
foreach worker ∈ cluster do
    worker ← polygonSet ←polygonSets [i]
    worker ← targetResolution
end

Worker:
rasterize(shapefile [polygonSet], targetResolution)
```

In order to achieve a scalable solution in this segment, a data partition scheme was devised so that each worker in the cluster could perform the atomic rasterization process independently. This was achieved by subsetting the total number of features (i.e., polygons) in the *shapefile*. This process is exemplified in the algorithm 5. Then, each subset is attributed to a given worker. The final result of this partition scheme is that so each worker was a shapefile of its own whose individual polygons do not overlap any of the polygons held by other workers, representing a unique subset of the original *shapefile*. After each worker applies the rasterization method to its shapefile subset, a number of rasters equal to the number of workers is generated. The final step in this process is performed by the master node. After all the workers result rasters have been collected, a merging phase is initiated, culminating in the final raster.

4.3 Scalable Pre-Processing of the Training Data

When dealing with large data, the direct conversion from the raster file possibly in disk to an in-memory R data-frame is not feasible.

Due to this constraint, the ff-data-frame structure, or *ffdf* for short, available in the *ff* package was the selected option to overcome memory issues. The ffdf data structure will be used as a replacement for the underlying *data.frame* structure used by the *trainingTable* in the non-scalable implementation. For differentiation purposes this new implementation of the *trainingTable* will be referenced as *ffTrainingTable*. In order to effectively populate the *trainingTable* a chucking procedure was devised in order to (1) read smaller-than-ram segments form the available rasters and (2) write the chunks into the *ffTrainingTable*.

The *raster* package provides a method named *blockSize* that given a user-defined chunk size, outputs a list of three elements: (1) *rows* the suggested row numbers at which to start the blocks for reading, (2) *nrows*, the number of rows in each block, and, (3) *n*, the total number of blocks, effectively allowing reading or writing smaller than RAM raster chunks.

Three different components are extracted from the raster chunks. These components are the x y geographical coordinates as well the respective values. This is done by using three different methods from the *raster* package. The methods *yFromRow* and *xFromCol* are used for the the x y geographical coordinates extraction, and the *getValues* method for the spatial data.

Once this data is stored in RAM, the next step is populating the *ffTrainingTable*, a process leveraged by the *write.ff* method, a simple low-level interface for writing vectors into ff files.

In order to achieve parallelization in this component, the *foreach* method was applied, so that each iteration regarding a different chunk is performed by one available worker in the cluster. This is possible due to the well defined regions both on the raster layer and on the *ffTrainingTable* on which the workers are allowed to perform reading and writing operations.

The sampling procedure if defined by the user follows the same lines as the previous version where a set of entries from the *ffTrainingTable* is randomly selected for training purposes.

4.4 Scalable Training and Prediction

In regard to the train-prediction procedure, new scalability considerations had to be addressed in this project. For instance, the caret package used for training in the baseline version does not support batch training. Instead, caret assumes resampling as the primary approach for optimizing predictive models in regard to larger-than-memory datasets. To do this, many alternate bootstrapped samples of the training set are used to train the model and predict a hold-out set. This process is repeated many times to get performance estimates that generalize to new datasets although, this training system was not implemented. To improve processing time of the multiple executions of the *train* function, *caret* supports parallel processing capabilities of the *parallel* package, if a cluster is previously registered and the flag *allowParallel* in the trainControl object is enabled. Although this capability was introduced in the S-Dissever, it should be noted that (1) only some models will take full advantage of the cluster parallelization (including the *randomForest*) and (2) as the number of workers increases, the memory required also increase as each worker will keep a versions of the data in memory[10]. Due to some of this constraints, a new package named *biglm* [11] was implemented as a replacement for the caret linear model and as prof-of-concept for the newly implemented data partitioning scheme. This model can be updated with more data using the method *biglm::update*, allowing linear regression on data sets larger than memory. This is done by chunking the *ffTrainingTable*, using the previously referenced method *chunk* from the *ff* package.

In order to preserve the initial baseline functionality, if one regression method is selected from the caret package a sampling procedure is applied to the training set. This enables loading a subset from training data that will fit the available RAM. If the sampling parameter is set by the user and is not enough to load the necessary entries of the *ffTrainingTable* into RAM then, this parameter is decreased and a warning message is issued.

The prediction component was reformulated in order to make use of the new out-of-memory partitioning scheme and the cluster multicore capabilities, independently of the used model (i.e., one of the *carat's* available models or the *biglm* regression model). This was done via the parallel *foreach* method, where the fitted model is distributed among the cluster and each worker is responsible to predict the results based on smaller-than-ram chunks of the *ffTrainingTable*.

4.5 Mass Preserving Areal Weighting Leveraging External Memory

The mass preserving sequence is performed in a similar fashion to the pycnophylactic interpolation. However, the source data is in the *ffTrainingTable*, a ffdf object. The first step in this process is to aggregate all predictions regarding the same *zoneId*. In a *ffdf* object this can be performed by the *ffdfdply* method. The *ffdfdply* performs a split-apply-combine on an *ffdf* data structure by splitting it according to a user defined function (i.e., a sum function in this particular case). Then, the results are storage in an new *ffdf* object named *aggregationTable*. This process does not actually split the data for efficiency purposes. In order to reduce the number of times data is put into RAM for situations with a lot of split levels, the *ffdfdply* method extracts groups of split elements which can be put into RAM according to initial configuration of the batch size. The number of elements of each *zoneId* is also counted by the *ffdfdply* method (i.e., using a *count* function for each split level) and stored as well in the *aggregationTable*. Then, for each entry in the *aggregationTable*, the correction values for *correctionA* are calculated as stated in Equation 2. The previously chunking partition scheme (i.e., using the *chunk* method) is applied in this step, complemented by parallelization where each chunk is processed by a node in the cluster. The final result in this intermediate step is a resulting value for each *zoneId* (i.e., the *correctionA* value) to incremented in every cell.

[10] `http://topepo.github.io/caret/parallel-processing.html`

[11] `https://CRAN.R-project.org/package=biglm`

The two remaining steps, in the mass preserving process (namely: *negativeRemoval* and *correctionB*) follow the same parallel-partition as described here in *correctionA*.

5 Experimental Evaluation

To better access the improvements of the new *S-Dissever* application and its sub-components, multiple evaluations were carried out, analyzing execution times across datasets of multiple resolutions in regard to a multiple number of cores, completed by correspondent speedup assessments. All tests presented in this Section were performed on the basis of (1) a shapefile describing the geographical region of England and Wales, including 1486 different administrative regions and their respective population counts from which the the desegregation was performed and (2) two auxiliary variables (i.e., elevation and land cover information) at a resolution of 0.0008DD.

All testes in this machine were performed in a machine with the following specification:

- 2 x Intel(R) Xenon(R) E5-2648L v4 @ 1.80GHz
- 32Gb DDR4 ECC @ 2.4GHz

This hardware configuration has a total of 14 physical cores (8 per CPU). Each core has 16 total threads, summing up to 56 threads in the hyper-threading mode. This detail might explain some performance degradation verified in the evaluation of the developed implementation beyond the 14 cores region.

The rest of this Section is as follows: In Section 5.1 a profiling the baseline dissever algorithm is presented. Section 5.2 presents the different sub-components in the new scalable pycnophylactic interpolation algorithm. Section 5.3 presents the new implementation of the parallel rasterization process. Section 5.4 addresses the performance of the pre-processing phase for the training data. Finally, Section 5.5 exhibits the results for the training-prediction iteration.

5.1 Profiling the Baseline Dissever Algorithm

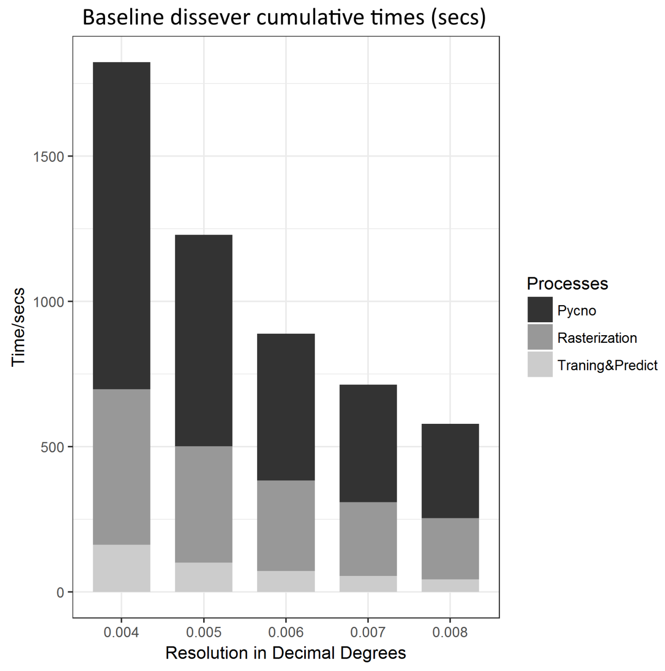
To better understand which components from the first version of the dissever algorithm are responsible for the bulk of the execution time, a execution time profiling using two dummy auxiliary variables at multiple target resolution was performed. The training model in this particular case was a linear regression. The results are displayed in Figure 2 (a).

The profile regarding execution time revealed that more than 80% of the total execution time, across all different target resolutions was spent in the pycnophylactic interpolation and rasterization components alone. The percentage of time corespondent to each component remains constant independent of the target resolution used.

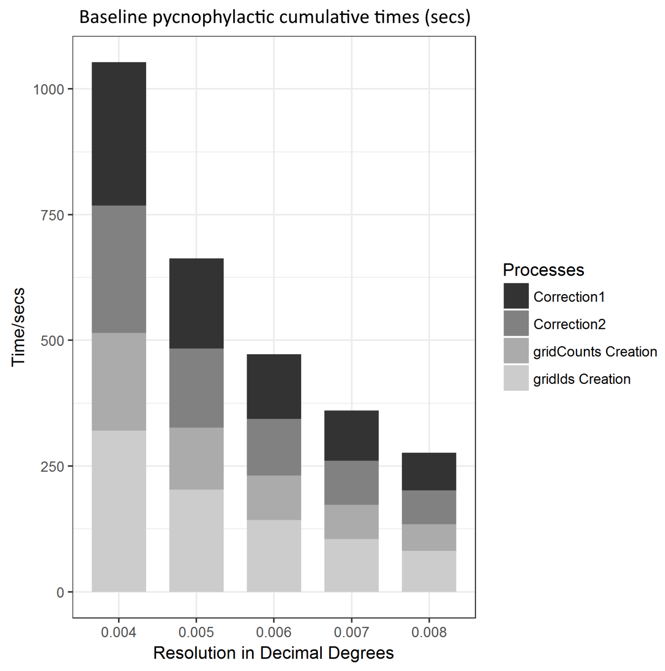
As previously described, the rasterization process is an atomic component, as such, it can not be broken down into sub-components for further analysis. In regard to the pycnophylactic interpolation process, a deeper inspection was performed into the existing sub-components. This analysis is depicted in Figure 2 (b). The conclusion was that more than 99% of the time was spent in the components that required access to the entire set of *zoneId* like populating the count and id grids or performing mass preserving corrections. On the other hand, the core process of the pycnophylactic interpolation (i.e., the neighbor cell average calculation) had a minimal significance in the overall time, representing less than 1% of the total time.

5.2 Experimental Results for the Scalable Pycnophylactic Interpolation

Experiments into the evolution of the different components in the new scalable version of the Pycnophylactic Interpolation algorithm were carried out using a multi-core environment. In Figure 3 (a) the cumulative times of the principal components are presented in a graph bar for the execution of the algorithm to a target resolution of 0.004 decimal degrees (DD). Figure 3 (b) presents speedups for a target resolutions



(a) Dissever profiling for two auxiliary variables at multiple resolutions using a linear model



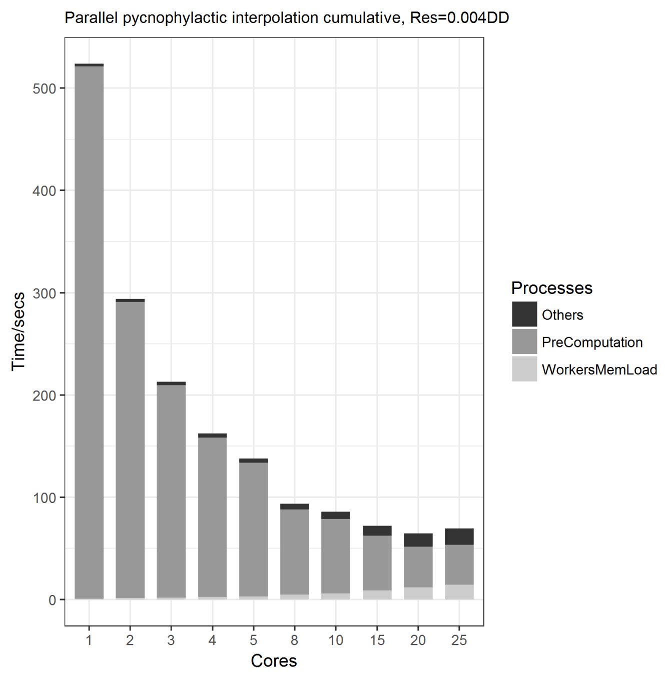
(b) Pycnophylactic interpolation code profiling at multiple target resolutions

Figure 2: Code profiling at different components and sub-components of the baseline dissever algorithm

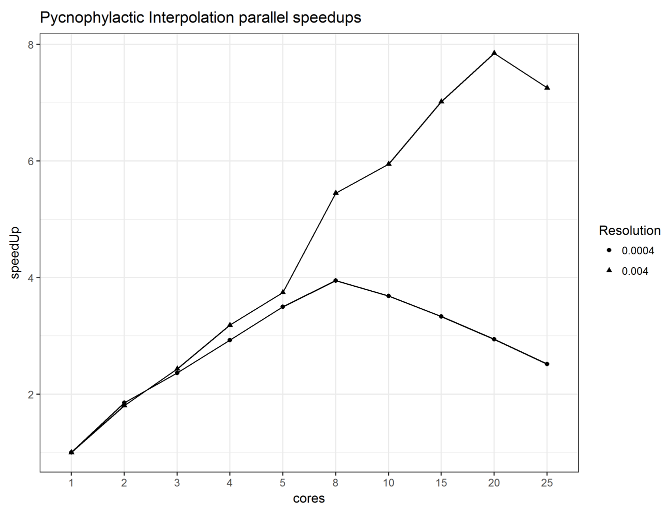
of 0.004 DD and 0.0004 DD compared to a single-core execution of the same algorithm. It should be noted as well that the convergence factor for this tests was set to 0, meaning that the cycle regarding neighbour average replacement and respective correction was performed only once, represented in Figure 3 (a) by the others component. This component also aggregates the cluster initialization times. The component *WorkerMemLoad* describes the time it takes to load the necessary data into the cluster. Form the observation its possible to conclude that 90% to 98% of the time is spent on the newly implemented pre-computation phase, allowing for a very fast execution of the averaging cycle, possible to many iteration without significant penalties. In Figure 3 (b) the speedups for a medium and high target resolutions are described. It is interesting to note that the optimal speedup peaks at about 8 cores for the high resolution. This decrease in efficiency is associated with the merging phase on which disk-based rasters returned by the different workers in the cluster are aggregated back in the main process. Also, as the number of workers increases, the memory available to each worker decreases, resulting into an increasing overhead originated from a higher number of continues read-write operations (i.e., the batch size decreases). For the medium resolution, the decrease in efficiency happens at about 20 cores. This is simply due to the times for the cluster initialization, which compared to the total execution time represent an increasing percentage of the total time, although negligible for higher resolution.

5.3 Experimental Results for the Parallel Rasterization

Regarding the parallel rasterization process, in Figure 4 (a) the total execution time across different cores is presented to a target resolution of 0.0008DD. the component others, aggregates all other components described in Section 4.1. In Figure 4 (b), the speedups for the process execution to resolutions of 0.008DD and 0.0008DD in comparison to a single-core execution of the same process are displayed. This component shows a fairly good speedup increase with the usage of more available cores. The optimal point for the used target resolutions sits between 15 an 20 cores at a speedup of 12 times. The only reason for the observed



(a) Parallel pycnophylactic interpolation total cumulative times for a resolution of 0.004 Decimal degrees across multiple cores



(b) Parallel pycnophylactic interpolation for a resolutions of 0.008 and 0.0008 Decimal degrees across multiple cores

Figure 3: Multi-core evaluation for the rasterization pycnophylactic interpolation process

decrees in efficiency beyond this point is due to the merging presses from aggregating results regarding the different workers.

5.4 Experimental Results for the Pre-Processing of the Training Data

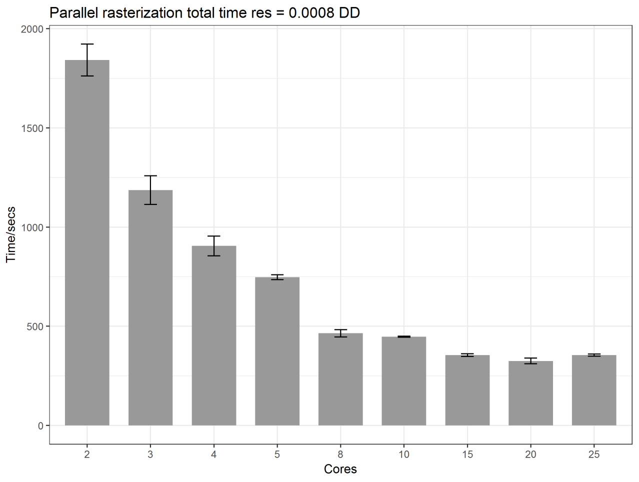
The parallelization in this process was performed in such way that each worker populates a well defined segment of the *ffTraningTable*. As presented in Figure 5 this strategy escalates fairly good up to 5 times speedup increase around the 15 cores area. One of the reason for the tipping point in this area is a well defined on. Given the fact that the *chunk* method in the *ff* package for splitting the table was set to a automatic BATCHBYTES value, it happen so that the number of splits in the *ffTraningTable* was 16, due to the size of the table in regard to the used auxiliary variables.

5.5 Experimental Results for the Training and Predict Cycle

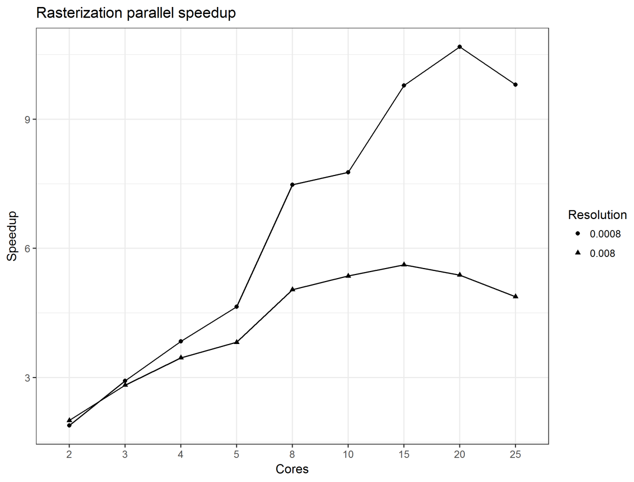
As in Section 5.4 this component was evaluated in regard to the same training data at the same extent and resolution. Multi-core assessment in the training and predict cycle is bound to show only superficial improvements, due to the marginal implementation of multi-core utilization. The focus on this component was in the utilization of out-of-memory training data. However, as depicted in Figure 6 the optimal point is in the the 4 cores region. The reason why efficiency degradation follows this point is due to the added overhead in the rather small performed parallel computation.

6 Conclusions and Future Work

In this paper, I reported the development of a new variant of the dissever algorithm advanced by João Monteiro. This new variant called S-Dissever was design to (1) make use of new multi-core architectures

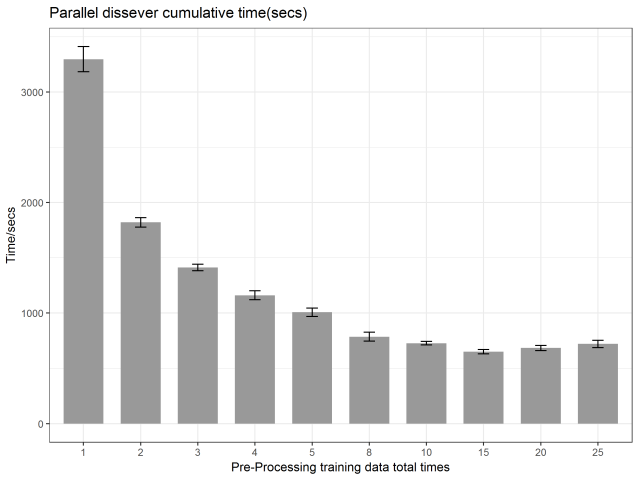


(a) Parallel rasterization total times for a resolution of 0.0008 Decimal degrees

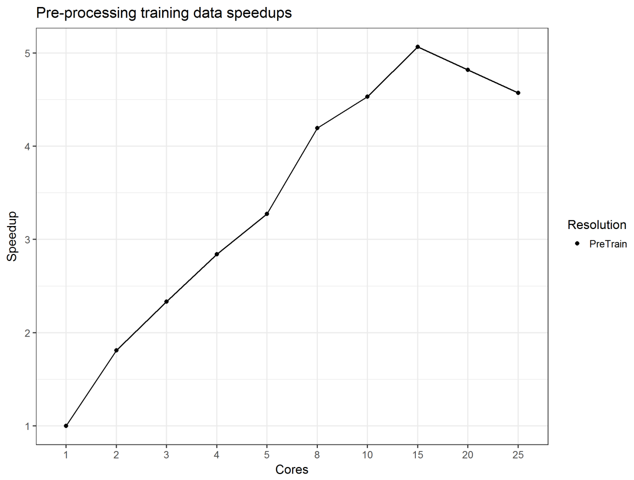


(b) Parallel rasterization speedups for a resolutions of 0.008 and 0.0008 Decimal degrees

Figure 4: Multi-core evaluation for the parallel rasterization process

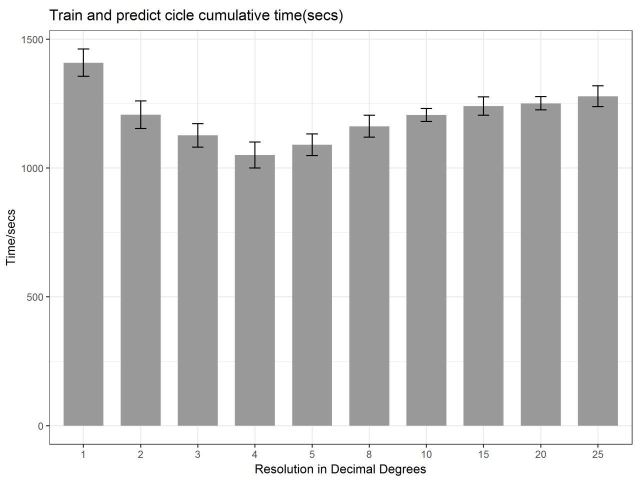


(a) Pre-Processing of the Training Data total times

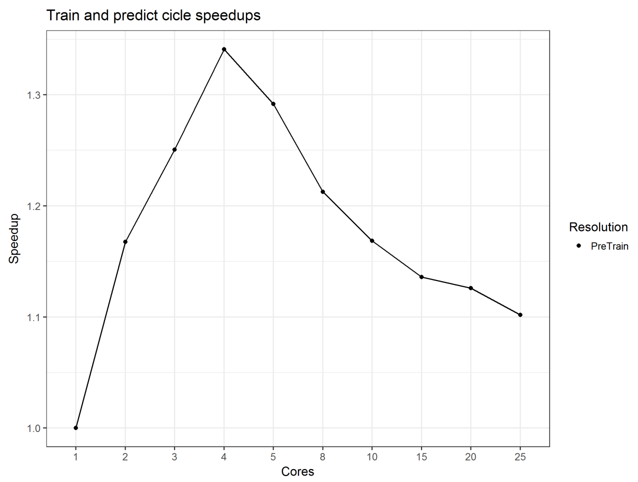


(b) Pre-Processing of the Training Data speedups

Figure 5: Multi-core evaluation for the Pre-Processing of the Training Data



(a) Parallel training and predict cycle total times



(b) Parallel training and predict cycle total times speedups

Figure 6: Multi-core evaluation for the parallel rasterization process

while (2) taking advantage of secondary memory, enabling very large data sets of data with fine resolution to be used as auxiliary information in the desegregation procedure.

For future work it would be interesting to continue improving the number of components benefiting from parallelization. For instance, some sub-process (e.g., pycnophylactic interpolation) in the S-Disseverrel rely on a merging phase in order to aggregate results produced by the multiple workers in the cluster (i.e., from multiple raster to a single raster). By exploring methods available in the raster package like the *writeValues* method one cold possible devise a more efficient parallel merging procedure.

Other important contribution yet to be made is the employment of the newly developed out-of-memory data access scheme into other machine learning frameworks like for instance the keras[12] or tensorflow [13]. This frameworks allow the usage of deep convolutional neural networks and cold be implemented utilizing techniques such as batch gradient descent or bini-batch gradient descent (Bottou, 2010).

References

Bakillah, M., Liang, S., Mobasheri, A., Jokar Arsanjani, J., and Zipf, A. (2014). Fine-resolution population mapping using openstreetmap points-of-interest. *International Journal of Geographical Information Science*, 28(9):1940–1963.

Banfield, R. E., Hall, L. O., Bowyer, K. W., and Kegelmeyer, W. P. (2007). A comparison of decision tree ensemble creation techniques. *IEEE transactions on pattern analysis and machine intelligence*, 29(1):173–180.

Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer.

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1).

Briggs, D. J., Gulliver, J., Fecht, D., and Vienneau, D. M. (2007). Dasymetric modelling of small-area population distribution using land cover and light emissions data. *Remote sensing of Environment*, 108(4):451–466.

[12]`https://rstudio.github.io/keras/`
[13]`https://tensorflow.rstudio.com/`

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM.

Deville, P., Linard, C., Martin, S., Gilbert, M., Stevens, F. R., Gaughan, A. E., Blondel, V. D., and Tatem, A. J. (2014). Dynamic population mapping using mobile phone data. *Proceedings of the National Academy of Sciences*, 111(45):15888–15893.

Goodchild, M. F., Anselin, L., and Deichmann, U. (1993). A framework for the areal interpolation of socioeconomic data. *Environment and planning A*, 25(3):383–397.

João Monteiro, Bruno Martins, J. a. M. P. (2017). A hybrid approach for the spatial disaggregation of socio-economic indicators.

Kane, M. J., Emerson, J., and Weston, S. (2013). Scalable strategies for computing with massive data. *Journal of Statistical Software*, 55(14).

Lin, J., Cromley, R., and Zhang, C. (2011). Using geographically weighted regression to solve the areal interpolation problem. *Annals of GIS*, 17(1):1–14.

Lloyd, C. D. (2014). *Exploring spatial scale in geography.* John Wiley & Sons.

Malone, B. P., McBratney, A. B., Minasny, B., and Wheeler, I. (2012). A general method for downscaling earth resource information. *Computers & Geosciences*, 41(2).

Monteiro, J. (2016). Spatial disaggregation using geo-referenced social media data as ancillary information. Master's thesis, Instituto Superior Técnico.

Patel, N. N., Stevens, F. R., Huang, Z., Gaughan, A. E., Elyazar, I., and Tatem, A. J. (2017). Improving large area population mapping using geotweet densities. *Transactions in GIS*, 21(2):317–331.

Sridharan, H. and Qiu, F. (2013). A spatially disaggregated areal interpolation model using light detection and ranging-derived building volumes. *Geographical Analysis*, 45(3):238–258.

Stevens, F. R., Gaughan, A. E., Linard, C., and Tatem, A. J. (2015). Disaggregating census data for population mapping using random forests with remotely-sensed and ancillary data. *PloS one*, 10(2):e0107042.

Team, R. C. (2000). R language definition. *Vienna, Austria: R foundation for statistical computing.*

Tobler, W. R. (1979a). Smooth pycnophylactic interpolation for geographical regions. *Journal of the American Statistical Association*, 74(367):519–530.

Tobler, W. R. (1979b). Smooth pycnophylactic interpolation for geographical regions. *Journal of the American Statistical Association*, 74(367).

Zhao, Y., Ovando-Montejo, G. A., Frazier, A. E., Mathews, A. J., Flynn, K. C., and Ellis, E. A. (2017). Estimating work and home population using lidar-derived building volumes. *International Journal of Remote Sensing*, 38(4):1180–1196.