CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Internship Report

Manuel Schmid

Summer 2014

EPFL Engineering Internship at CSCS

1 Introduction

With the problems of global climate change becoming more and more visible, climate research has become an important field of studies. By refining climate models and improving their resolution, we hope to improve our understanding of the climate's behavior and development.

However, the amount of data created by these models is tremendous, which is becoming a limiting factor for new climate models. Several approaches for reducing the data output are currently being explored, each of which has advantages and drawbacks.

Lossless compression retains all information but the data savings are minimal. Lossy compression can reduce the data significantly but important information might get lost. Other approaches are based on saving only a limited number of checkpoints and rerunning the simulation from those whenever the results are accessed. This requires special care to ensure reproducible results as small differences in hardware, system configuration, and compiler options can lead to very different simulation runs and the energy efficiency is questionable.

During my internship, I worked with a new lossy compression algorithm based on principle component analysis. I improved and expanded a first implementation of the algorithm and started analyzing its performance for climate data.

2 Overview of the Algorithm

The algorithm is based on the work of Prof. Ilia Horenko from the Università della Svizzera italiana (USI). It is an extended version of principal component analysis (PCA), a technique commonly used for exploratory data analysis. By treating the climate data as a set of vectors that form the columns of a matrix, we can use this technique for data compression.

Principal component analysis is used to find the dominant directions in a set of N-dimensional vectors centered around the origin. If we think of each vector as a point in an N-dimensional vector space, the first direction found by PCA is the direction along which the variance of the points is maximal, i.e. along which the data is most spread out. The second direction is perpendicular to the first direction and again the variance is maximized. It can be shown that these directions correspond to the eigenvectors of the covariance matrix XX^T when X is a matrix with the data as column vectors.

These directions found by PCA can be used for data compression. If we pick the first M dominant directions, we can project each data vector into the subspace spanned by these directions. Instead of all the original, N-dimensional vectors, we only save the M eigenvectors and the M-dimensional projections of the data vectors. If M is significantly smaller than N, the size of the data can be reduced drastically.

The present algorithm uses an expanded form of principal component analysis. Instead of solving the problem globally, it groups the vectors into K different clusters and compresses each cluster separately using the first M vectors of the principal component analysis. K and M are parameters of the algorithm.

Grouping the vectors into clusters is done iteratively. Initially, all vectors are arbitrarily assigned to a cluster. Then, a PCA is performed to find the first eigenvector of each cluster. In a next step, all vectors are reassigned to the cluster where they are best described by the cluster's eigenvector. After this, the algorithm goes back to performing a PCA for each cluster and repeats the process until the cluster configuration converges. In the end, a final PCA is performed for each cluster, calculating M eigenvectors instead of just one.

3 Overview of the Implementation

The current implementation has the goal of testing the algorithm's performance for different data and analyzing the compression losses. While it does not act as an operational compression module yet, it should be possible to extend the code for integration into climate simulations.

The program currently does the following steps: First, the data for several climate variables such as temperature and humidity is read from a NetCDF file. As there are several variables and each variable can have several dimensions such as longitude, latitude, and time, the data has to be reorganized into a single, two-dimensional matrix. Each variable is scaled to have values in the interval $[-1, 1]$ to prevent variable with large variability from dominating the principal component analysis. The data is then compressed with the procedure described in the previous section. For the analysis of compression losses, the program reconstructs the full matrix from the compressed data and compares it to the original, uncompressed matrix, calculating several statistics. The reconstructed data is then written to a NetCDF file in the same form as the original file. This file can be used for further comparison with the original data.

The program is written in C++ and parallelized with MPI. Each MPI rank reads only a part of the data corresponding to some columns of the data matrix. Whenever an operation involves the full matrix, information has to be exchanged between the processes. Some operations, most notably the re-clustering of the columns, can run locally.

For the linear algebra operations, the program can use different back ends. The first version is based on Eigen, a mature, CPU-based linear algebra framework for C++. The other two versions are based on minlin, a small library based on Thrust which can use both OpenMP or CUDA for running operations on either a CPU or GPU.

As the algorithm is based on principal component analysis, the code has to solve an eigenvalue problem. More precisely, it has to find the eigenvectors corresponding to the M largest eigenvalues of the correlation matrix XX^T. This matrix can be very large, so using a direct eigensolver is not feasible. Instead, the program uses the iterative Lanczos algorithm to find the eigenvectors. This also has the advantage that the full matrix XX^T never has to be built, as the algorithm only needs the matrix-vector multiplication XX^Tv, which can be computed as $X(X^Tv)$.

4 Expanding the Program

During my internship, I spent most time working on the code to make it more robust and add features. I worked on getting all versions to run and behave the same way, changed the build system to CMake, restructured the code to make it more modular, added a full command line interface and extensive documentation, and improved the functionality for reading and restructuring data from NetCDF files. As this last point added significant complexity to the code, I will describe it in some more detail here.

4.1 Details of the NetCDF Functionality

As the compression algorithm runs on a 2D matrix, the climate data has to be restructured before it can be compressed. In addition, each column of the matrix has to be assigned to one of the MPI ranks.

These tasks are carried out by the class "NetCDFInterface". The constructor sets up the data structures describing this mapping based on the input parameters and information read from the NetCDF file. The member function "read_matrix()" then uses this information to read the assigned part of the data from the NetCDF file and write it to a matrix in the correct order.

4.1.1 Types of Dimensions

For each run of the program, the dimensions of all variables are split into three categories. "Compressed dimensions" will be placed along the matrix columns. The values along these dimensions will be replaced with just M coefficients for the eigenvectors in the compressed representation of the matrix. "Distributed dimensions" will be placed along the matrix rows. Each MPI rank only gets some of the entries along these dimensions. This is equivalent to building the full matrix and assigning some of the columns to each MPI rank. "Indexed dimensions" are dimensions along which only one entry at a specific index is chosen while all other entries are discarded. This has been implemented specifically for the CESM data set which included two time steps (initial conditions and average values), only one of which was meant to be compressed.

4.1.2 Combining Variables

If we rearrange the data for each variable, we get a series of rather small matrices. For the compression, we want to combine these matrices so we can use the correlation

between different fields. We can stack the matrices horizontally, adding some columns for each variable, or vertically, adding some rows for each variable.

We can even stack variables with different sizes, as long as they only differ in the direction along which we stack them. Thus, we can combine 2D and 3D variables, e.g. by placing latitude, longitude, and time along the rows (i.e. as distributed dimensions) whereas the vertical levels are placed along columns. This results in a series of matrices with the same width and only one (2D variables) or several (3D variables) rows, which can then be stacked vertically.

4.1.3 Selection of Data Ranges

The NetCDFInterface class first sets up two arrays, "start" and "count", with the first index and the number of entries for each variable and dimension. For compressed and indexed dimensions, this is trivial. For distributed dimensions however, this is a bit more complicated, as each MPI rank has different values. The selection should be flexible enough to deal with different numbers of MPI processes and distributed dimensions.

For each variable, we do two steps. First, we collect the number of distributed dimensions and decide how many parts each of them should be split into. This corresponds to the mathematical problem of finding N numbers — N being the number of distributed dimensions — the product of which is equal to the number of MPI processes. We simplify the problem by enforcing that the number of MPI processes is a power of two, which should not be a problem in practice. With this, we can use the following algorithm to get a reasonably balanced process distribution:

```
p = number_of_mpi_processes
i = 0
number_of_processes_along_dimension = [1] *
   number_of_distributed_dimensions
while p > 1:
  p /= 2
  number_of_processes_along_dimension[i] *= 2
  i = (i + 1) % process_distribution.size()
```

The second step consists in assigning an index along each distributed dimension to each MPI rank. We can use the following algorithm for this:

```
r = my_rank
p = mpi_processes
for i in range(number_of_distributed_dimensions):
  p /= number_of_processes_along_dimension[i]
  index_along_dimension[i] = r / p
  r %= p
```

Once we have this, we can easily calculate the start and count for each dimension, variable, and MPI process. We simply divide the number entries along a dimension by the number of processes along the dimension to get "count". For "start", we multiply this number by the index along the dimension.

4.1.4 Mapping to 2D-Martix

While each MPI process now reads the correct subset of values, the data still consists of a 1D-array with the original dimensions in the original order. As the algorithm should be able to place each dimension along the rows or columns of the matrix depending on the desired configuration, the values have to be rearranged such that the fastest varying indices correspond to the dimensions that should be placed along columns. This way, the array can be used to create a column-major matrix, which is the default for both Eigen and minlin.

When I initially added this remapping to the program, I used the functionality provided by the NetCDF library. The library provides a function for reading data called "nc_get_varm_double()", which has an argument "imap" in addition to "start" and "count". This is an array with one entry per dimension specifying the distance between two elements of this dimension in the output array. However, using the functionality of the NetCDF library proved to be extremely slow, so I rewrote the mapping to occur in memory after the data has been read from the file.

The approach used for this is very similar to the approach of the NetCDF library. Again, we calculate the distance between two elements in the output array for each dimension. However, we now do this separately for the distance in row- and column-direction, as we write directly to a matrix instead of a 1D-array.

For the inter-element distances along the rows, we go through all distributed dimensions. The first dimension gets a distance of 1, as the elements are consecutive. The other dimensions get the product of the number of elements along all dimensions before them. The compressed dimensions (placed along columns) get an inter-element distance of 0 along the rows.

For the inter-element distances along the columns we do the same, except that the compressed dimensions get a non-zero distance and the distributed dimensions get a distance of 0. The inter-element distances of indexed dimensions are not used and therefore can be set to any value.

Once these arrays describing the mapping are built, restructuring the data becomes easy. We simply go through all values in the original data, keeping track of the indices along each dimension (starting from all zero), and multiply the indices with the inter-element distance along the dimensions. This way, we obtain the new row and column index and we can write the values to the correct location in the rearranged matrix.

5 Evaluating the Algorithm

The second major part of my internship was starting the evaluation of the compression algorithm. I ran the compression with different configurations to get an idea of its overall performance and to find configurations that work well.

The analysis was based on annual average data from the Community Earth System Model (CESM). As this data had no time dimension and only one horizontal dimension (no separate latitude/longitude dimensions), the possible configurations for distributed and compressed dimensions were manageable. Apart from the horizontal dimension "ncol" with 48602 entries, there was a vertical dimension "lev" with 30 entries or "ilev" with 31 entries, depending on whether the variables were defined within a vertical layer or at its interface. For 2D variables, the vertical dimension was missing.

The data set had 97 3D variables and 101 2D variables. As some of the 2D variables posed numerical problems, probably due to missing values, we only used 89 of them.

5.1 Horizontal Stacking

With the algorithm in its current form, stacking variables horizontally did not seem to work very well. The clustering tended to converge to a configuration with one very large cluster while all other clusters were more than an order of magnitude smaller. This made it difficult to find a number of final eigenvectors that leads to acceptable compression for both the large cluster and the small clusters. If a future version of the algorithm has additional logic to deal with different cluster sizes, it might make sense to look at horizontal stacking again.

5.2 Selection of Compressed Dimensions

If horizontal stacking is excluded, it is evident that variables have to be combined along the columns of the matrix. Thus, it made sense to place the large dimension "ncol" along rows, as both K and M would have to be very small otherwise.

This means that there were two options left for the configuration of the dimensions. Either we can place the vertical dimension ("lev" or "ilev") along the columns or we place it along the rows. In the latter case, we can only combine variables with the same number of levels.

Looking at the relationship between the compression ratio and the associated RMS errors, it is clear that placing the vertical dimension along the columns leads to much better performance. This is convenient, as we can combine variables with different numbers of levels this way.

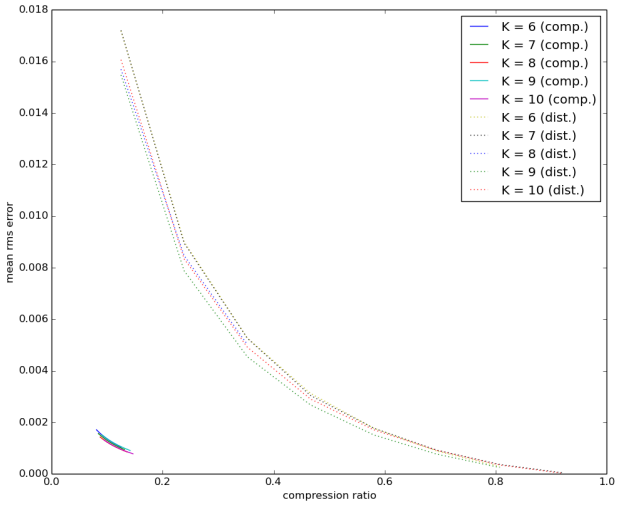

Figure 5.1: The scenarios with vertical levels as compressed dimensions (solid lines in the bottom left corner) perform much better than when they are placed as distributed dimensions (dotted lines).

5.3 Optimal Number of Clusters (K)

Once the configuration of the dimensions is fixed, the parameters for the number of clusters (K) and for the number of final eigenvectors (M) remain to be determined. As we are interested in strong compression with small errors, it makes sense to plot the error against the compression ratio. If we plot curves for a fixed K and a varying M, we see that these curves generally follow the same form and do not overlap. This means that there is an optimal value for K. For the dataset and dimension configuration used, this optimal value seems to be around K=10. This leaves a single parameter M for the choice between stronger compression and smaller errors.

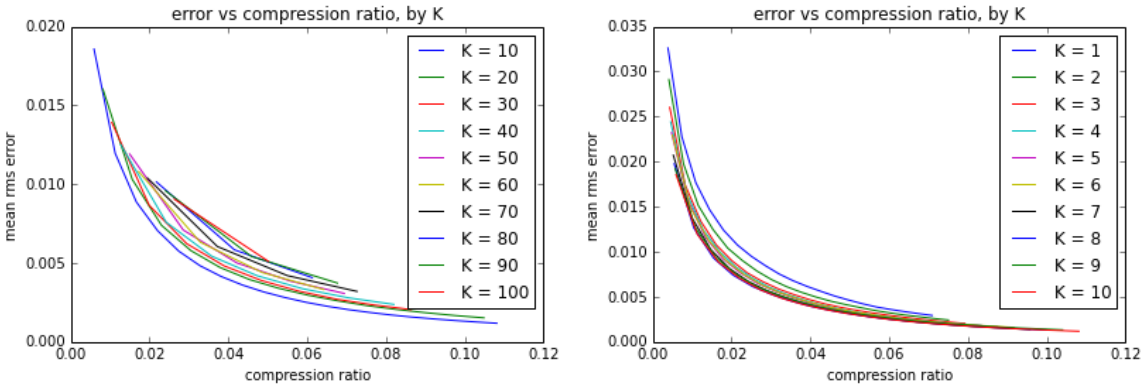

Figure 5.2: There seems to be an optimum for K at around K=10.

5.4 Excluding 2D Variables

Combining 2D and 3D variables can potentially be problematic. As 3D variables have much more values, it could be possible for them to dominate the compression. However, this did not seem to be the case. When combined, 2D and 3D variables seemed to have similar relative RMS errors while maximum errors tended to be smaller for 2D variables. It is therefore best to combine all variables in a single compression, which is also preferable from a practical point of view.

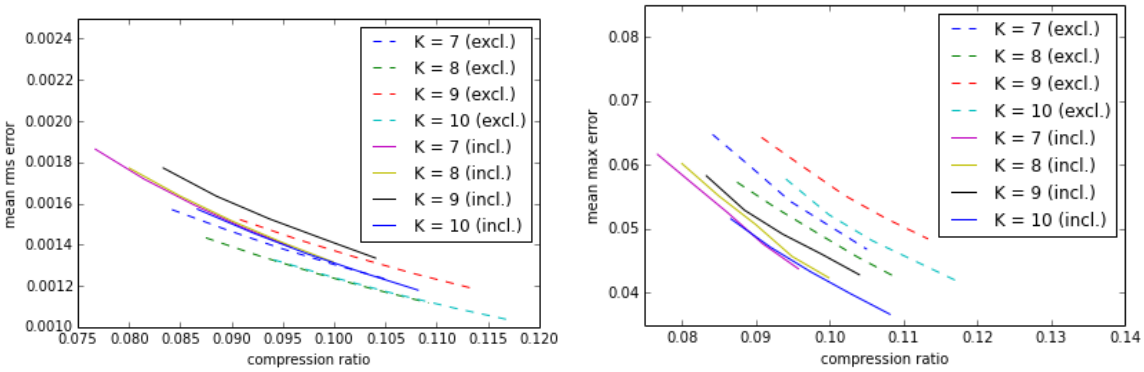

Figure 5.3: While the RMS errors are comparable with or without 2D variables, the maximum errors tend to be smaller when they are included.

5.5 Comparison to Other Algorithms

Comparing the algorithm to other lossy compression algorithm is currently difficult as the implementation does not yet have the functionality for adding lossless compression on top of the lossy compression algorithm. This means that the compression ratios do not represent the actual compression strength of the algorithm yet, which would be necessary for a fair comparison.

An initial comparison with values from a publication by Baker et al.[1] using the same climate data shows that performance is comparable to other lossy compression algorithms. However, most of the examples from this and other publications use weaker compression. As weaker compression leads to longer running times for our algorithm, I have focused my evaluation on rather strong compression.

[1]Baker, A H, Haijing, X, Dennis, J M, Levy, M N, Nychka, D, Mickelson S. A.: 2014, A Methodology for Evaluating the Impact of Data Compression on Climate Simulation Data (in preparation)

6 Next Steps

The current implementation only allows for an incomplete comparison with other compression algorithms. While the core functionality of the compression is working, the fact that the compressed data cannot be written to a file yet prevents us from adding lossless compression. Therefore, future work should focus on this functionality.

For the purpose of evaluation, it would be enough to just save all data from the class CompressedMatrix to a file. This way, the file can be compressed with a standard lossless algorithm and the overall compression ratio can be compared to other lossy compression algorithms.

A more complete implementation would write a file where the data from all MPI processes is combined as if there was just a single process. This would make it possible to read or decompress the file with a number of processes other than what was used for the compression. However, this would involve much of the complexity of the NetCDFInterface class both when reading and writing the file.

Once compressed files can be written, the algorithm can be compared to other lossy algorithms by imposing the same compression ratio for all of them and measuring the error. As I have mostly been using very strong compression, it would be interesting to look at the errors for weaker compression ratios too, presuming that the algorithm still achieves acceptable running times.

While comparison with other algorithms is interesting as a first result, the main question will be whether the lossy compression achieved by this algorithm is usable for climate data. This question is much more difficult to answer as it depends on a definition of what is acceptable as compression error. The paper by Baker et al. mentioned earlier[1] proposes a technique to relate the error introduced by lossy compression to the error produced by slight perturbations in the initial conditions of the climate model. A script to calculate these so-called RMSZ values has also been written as part of this internship and can be used for further analysis.

[1] Baker, A H, Haijing, X, Dennis, J M, Levy, M N, Nychka, D, Mickelson S. A.: 2014, A Methodology for Evaluating the Impact of Data Compression on Climate Simulation Data (in preparation)

7 Conclusion

The algorithm I worked on during my internship provides strong lossy compression for climate data. While it is computationally expensive, it improves over simple PCA with smaller errors and better compression ratios. Comparisons with other algorithms are still difficult but the first results look promising. However, it remains to be seen whether the compression performance outweighs the computational cost.

During my internship, I have been able to improve the robustness of the algorithm's implementation and add new features. I also have started evaluating the algorithm and presented some first results. Obviously, this evaluation has to be expanded to a wider range of usage scenarios but my work can serve as a basis for further evaluations.

Personally, I have been very satisfied with my internship at CSCS. I have been able to get an impression of something in between working at a regular company and working in academia and I could see myself working in such a place later. Most importantly for me, it has been an experience of intense learning. I therefore feel that my internship has been a great success.

Acknowledgement

I would like to thank the CSCS staff for their assistance and interest in the project. In particular, I want to thank Will Sawyer and Ben Cumming for their supervision. I would also like to acknowledge Prof. Illia Horenko from USI for his input on the evaluation and Prof. Daniel Kressner who has been the EPFL responsible for the internship.