Parallel climate analysis using structured data

Francesco Laiti - 232070
Davide Lobba - 232089

Abstract—This report explains the development and evaluation of a parallel climate analysis tool using structured data. We develop the application to exploit the increasing hardware resources. To distribute the workload across multiple resources, we deploy the tool using MPI for multi-processing and OpenMP for multi-threading. Our tool is able to reduce drastically the time involved to process precipitation flux data using a parallel approach compared to a serial one.

1 Introduction

Climate change is an ongoing topic and refers to long-term shifts in temperatures and weather patterns. These shifts may be natural, such as through variations in the solar cycle. But since the 1800s, human activities have been the main driver of climate change, primarily due to burning fossil fuels like coal, oil and gas [1].
In order to extract and analyze vast amounts of climate data, it is important to exploit parallel computing to achieve in an efficient way and in short time the desired results.
Our work wants to explore the implementation of a parallel climate analysis tool using structured data to exploit the provided hardware resources. To test out our algorithm, we used a benchmark precipitation flux dataset provided by the CMSS research center. Our objective is to output the average precipitation flux across the years, applying a reduction method on the `time` dimension.
We structured this report as follows: we give an overview of the dataset, and then we describe the serial implementation of the algorithm and its parallel version. Finally, we perform benchmarking and analysis, and we draw conclusions.
The code is written in C and for memory debugging we used the `valgrind` tool. The cluster HPC@Unitrento has been provided by the University of Trento. The source code is available at `github.com/davidelobba/Parallel-Climate-Analysis`

2 Dataset

For this project, we decided to use precipitation flux datasets available in a shared folder of the HPC@Unitrento cluster. The data are stored in a NetCDF (Network Common Data Form) format. The dataset is composed of many attributes, but we report the most important one for our task:

- *Dimensions* are used to define the size and shape of the data arrays in a NetCDF file. Organized by `time`, which for a single file is based on 25 years (9125 days, or called records), `latitude` and `longitude`, which are, respectively, based on 192 and 288 points. In the end, we obtain a grid of latitudes and longitudes of 55296 points and for each point we have the precipitation flux of the given day.
- *Variables* are used to store the actual data in a NetCDF file. Organized by `time, latitude, longitude` and `precipitation flux`. Specifically, the variable that we analyze is the precipitation flux, which is a function that returns a float number for a given time, longitude and latitude value, which tells how many precipitations for each day, including both liquid and solid phases, have been happened for that point of the grid.

To increase the size of the dataset to reduce, and to have a reasonable performance to analyze between the serial and parallel versions, we wrote a script to use the `cdo` library, which helps us to merge the `time` dimension from several NetCDF files that contain similar data structure. The final result is a dataset with the precipitation flux for each point of the grid from 1850-01-01 to 2014-12-31, so instead of having only the data of 9125 days, we deal with a dataset with a precipitation flux of 60225 days. Our goal is to reduce the 3-dimensions data `time, latitude, longitude` as input to 2-dimensions data `latitude, longitude` as output, so by reducing along the `time` dimension.

3 Serial implementation

We decided to adopt the official NetCDF library to read and write data. The implementation is straightforward. In the following subsections, we describe the three main parts executed sequentially by the tool:

3.1 Reading

In the beginning, the algorithm takes as input the `.nc` file, that has been already merged over the years, as described in Section 2, and opens it. After that, we retrieve the dimension IDs and variable IDs to properly call them when reading the data. In order to get access to the precipitation flux of a specific record, two arrays of size 3, equal to the number of variables `time, latitude, longitude` are created:

- `count` specifies the number of elements to be read along each dimension;
- `start` specifies the starting point of the matrix to be read.

Also, we created two 2-dim matrices of size `latitude` $\times$ `longitude`, one to store the temporal data and one for the output data of the writing part.

3.2 Reducing

In this phase, we start looping along the `time` dimension where, for every day, we retrieve the matrix regarding the precipitation flux of that day using `nc_get_vara_float` function provided by NetCDF. We pass to the function the NetCDF ID (which identifies the open dataset), the precipitation flux variable ID, the `start` and `count` parameters, and a pointer to the temporal matrix memory location where the data is going to be stored with dimensions defined by `count`. A nested loop sum up to the output matrix the temporal one. In this way, we are summing up the contributions of every record to the output matrix.

3.3 Writing

At the end of the loop, we divide every point of the output matrix by the number of days to obtain a precipitation flux average. Finally, we generate a new NetCDF file which contains only two dimensions, the `latitude` and `longitude` ones, with the precipitation flux average value of every point of the grid.

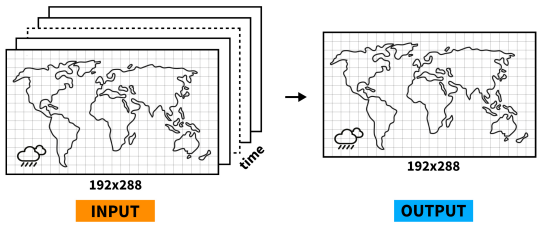


Fig. 1: Serial implementation graphically represented.

In Figure 1, we wanted to give an intuitive idea of what we have done. 1 process is involved in this example, where we loop along the `time` dimension to retrieve precipitation flux data of one slice (a slice represents a day with a full precipitation flux data) and, in the end, the algorithm outputs a new NetCDF file with the average of all the days.

4 Parallel Implementation

We decided to implement two strategies to parallelize the serial code: one using only MPI and one using hybrid parallelization, which consists in adopting both MPI and OpenMP capabilities.

We decided to follow a simple but effective structure: (1) multiple processes should work on a subset of the dataset by splitting the `time` dimension, (2) multiple threads of the same process should work on different parts of the matrix, where the matrix is full-fill with precipitation flux of a specific record.

Most operating systems allow multiple processes to open the same file in reading mode concurrently, as reading a file does not modify its contents and therefore does not create conflicts between processes. For this reason, every process opens and reads the data independently from the other process and works on the records range assigned. However, when it comes to writing to a file, in order to prevent data corruption and ensure data consistency, we decided to write the data by using the root process.

Threads are not affected by data transfer tasks because they work in a shared memory environment, but the costs of invoking a parallel loop and executing the barrier, cache and synchronization effects can greatly increase the cost [2]. The log data regarding time, nodes, number of processes, number of threads, etc. are saved in `.csv` file, shared across the different jobs.

4.1 MPI

Initially, we implemented MPI. To parallelize the serial version, we introduced several changes to the code to work correctly with MPI. We decided to parallelize along the `time` dimension. For every MPI process, we split the time in a uniform way across the available processes, managing the non-zero remainder division by giving the remaining records to one process. In this way, every process has a subset of data to read and sum on a local matrix defined as `local_pr_out` parameter. To return the matrix of output elements to the root process, we used the function `MPI_Reduce`. Following the MPI documentation, we set the MPI operation as `MPI_SUM` to collect all the local matrix of each process, and sum up the results in a final matrix, `total_pr_out` parameter.

Back in the root process, we divided every point of the output matrix by the number of records/days and stored the new data in a NetCDF file on the cluster. These last instructions are very fast to compute, thus we have not parallelized them.

With Figure 2, we wanted to give an intuitive idea of what we have done. 3 processes are involved in this example, where every process takes a subset of slices and, in the end, the root process outputs a new NetCDF file with the average of all the days, as already seen in the serial version.

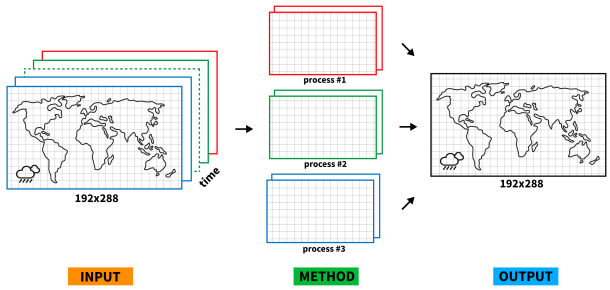


Fig. 2: MPI parallelization with 3 processes graphically represented.

4.2 Hybrid parallelization: MPI + OpenMP

To have a further analysis, we added to the MPI solution the OpenMP standard API. Specifically, we followed the same procedure as the MPI version with just a few key differences to allow multithreading. To add multithreading capabilities, we worked on the nested loop, after the collection part of a precipitation flux of a specific record, by dividing the grids into sub-grids which depend on the number of threads available. Using the parallel directive

```
#pragma omp parallel for collapse(2)
```

we are parallelizing two nested loops by collapsing them into a single loop then parallelize it. We ensured that the

`private` parameters, which are local to each thread, specifically the loop index variables `lat` and `lon`, are handled correctly. The `shared` variables include the local output matrix `local_pr_out` and the temporary matrix `local_pr_in`, which stores the current precipitation flux for a given record. There are no race conditions because each thread writes to a different part of the output matrix, and the temporary matrix remains unchanged inside the parallel loop. We observe that, without manually setting the private and shared clauses, the parallel directive automatically assigns the parameters to the correct categories. The number of threads is set with the key `num_threads(thread_count)`, where `thread_count` is computed to ensure that every thread has its dedicated core. At the end of the main loop, we proceed as we have already seen in Section 4.1.
With Figure 3, we wanted to give an intuitive idea of what we have done. 3 processes are involved in this example, where every process takes a subset of records and every process uses 4 threads to parallelize the nested loop. In the end, the root process outputs a new NetCDF file with the average of all the days, as already seen in the MPI version.

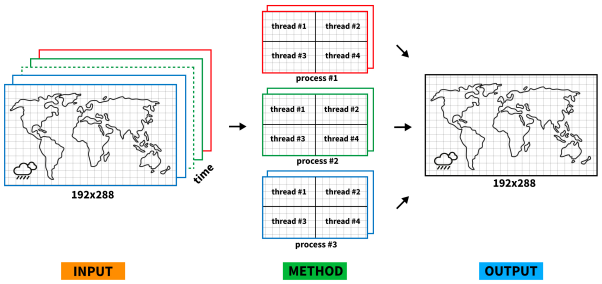


Fig. 3: Hybrid parallelization with 3 processes and 4 threads for each process graphically represented.

5 Performance and Benchmarking

In this section, we explain the setup that we adopted for benchmarking our solutions and discuss the performance evaluation of our algorithms. To benchmark the serial and MPI-only versions, we adjusted the configuration as follows: for the serial version we used an ad-hoc code; for the MPI-only version, we set the number of processes given via command line and a number of threads equal to 1. We start by evaluating the MPI version, finally the hybrid version.

5.1 Setup

All the tests were performed on the cluster HPC@Unitrento using the benchmark dataset built as described in Section 2. For the evaluation of the performances, we used the following metrics:

$$\text{Speedup} = \frac{\text{Time}_{Serial}}{\text{Time}_{Parallel(p)}} \tag{1}$$

$$\text{Efficiency} = \frac{\text{Speedup}}{p} = \frac{\text{Time}_{Serial}}{p \times \text{Time}_{Parallel(p)}} \tag{2}$$

where p is the number of processes used to run the parallel code.

To provide a more accurate measure of the overall runtime, we used the `MPI_Barrier` call to ensure that all processes start and finish the computation at the same time. To measure the elapsed time between two points in our code, we used `MPI_Wtime`.
In order to obtain a benchmark which was not subjected to race conditions or job overlapping, we submitted each job to the cluster separately.
We started computing the time required to run the serial code of our application and, following what was suggested in [3], we report the result of the minimum runtime. We have not done the average across all the runs because the interaction of the program with the rest of the system is unpredictable, and it will almost certainly not make the program run faster than it would run on a "quiet" system. Moreover, we decided to run each configuration 3 times to have more consistent benchmarks. We did this because we noticed that sometimes the I/O operations were impacting a lot the efficiency of our program.
We decided to test the problem with different sizes: `0.25` corresponds to a quarter of the total number of records/days, `0.5` to half records and finally `1.0` corresponds to the original problem, the totality of dataset's records.

5.2 MPI-only evaluation

We tested different settings using `PBS`, changing the number of nodes, cores, processes and placement strategies.
We ended up evaluating the algorithm using the `place=scatter:excl` and fixing the number of nodes to 4. We also tested `place=pack:excl` but found worse results. In order to ensure that every process has its own core, avoiding multitasking operations, and asking only the necessary resources to the cluster, we adopted the following relation for requesting the number of cores:

$$\text{Cores per node} = \frac{\text{Processes}}{\text{Nodes}} \tag{3}$$

where *Processes* and *Nodes* are given. We tested on a maximum of 128 processes because the available machines at our disposition have 24 cores each [4], and fixing the number of nodes leads to a maximum of 128 cores available. As expected, the MPI version outperforms the serial one.
As shown in Figure 6a, exploiting the parallel version we reduce the time needed to process the data. We reach the plateau after 8 processes, where increasing the number of processes does not seem to improve the performances. We can already notice how parallelization is well suitable for large problems rather than small ones. Regarding the speedup, we reach good performances when we are using 4 processes; after that points, the curve slightly decreases until we reach saturation. Regarding efficiency, we have great performances until 8 processes in half and original problem size, where we have an efficiency $> 70\%$. We can notice that the efficiency decreases as the number of cores increases. Also in this case, we can confirm that exploiting parallelization for small problems is not a good choice. The evident decrease in efficiency indicates that the algorithm cannot be entirely parallelized and that its performance is constrained by communication overhead.

5.3 Hybrid parallelization evaluation

We tested different configurations to exploit OpenMP capabilities. We tried to recreate the same configuration adopted before, thus we fixed the number of nodes to 4 and use a placement strategy `place=scatter:excl`. This type of configuration helped us to compare the two parallel versions. To assign the right number of cores to run our parallel program, we adopted the following relation:

$$\text{Cores per node} = \frac{\text{Threads} \times \text{Processes}}{\text{Nodes}} \tag{4}$$

where the number of *Processes*, *Nodes* and *Threads* are given. We fixed the number of threads to 4 to compare the results obtained with the previous section. As also specified in the MPI section 5.2, the relation (4) is useful to avoid multitasking operations. Threads are allocated on the same node and, if possible, on the same socket to guarantee fast communication between them. This has been achieved by using the MPI option `--map-by` and `--bind-to`.
The obtained results are disappointing. Indeed, we found that the hybrid parallelization with MPI and OpenMP does not improve the overall performances. We think that this behaviour is happening because we are invoking multiple threads at each record, and this is highly expensive in terms of computational resources, and it can slow down the overall computation. We have to deal with a very small grid. Probably, an increment of the resolution of the grid, built on the latitudes and longitudes, could improve the performances using hybrid parallelization.

5.4 Further analysis: MPI *vs* hybrid parallelization

In Figures 4 and 5, we can observe the differences between the MPI-only and hybrid parallelization. The problem size has been fixed to 1. We can observe that by using a hybrid parallelization there are no remarkable improvements. The performances of the two parallelizations are very close. Regarding the hybrid solution, we can notice that the best setting in terms of speedup, efficiency and time is reached using 24 processes and 4 threads per node. But it is clear that the performances slightly drop with respect to the MPI solution.

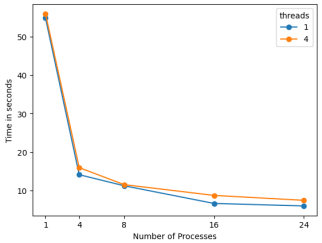
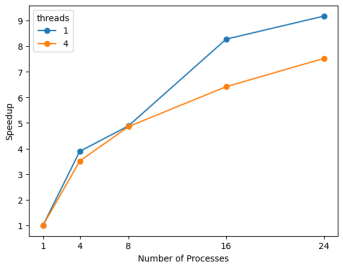
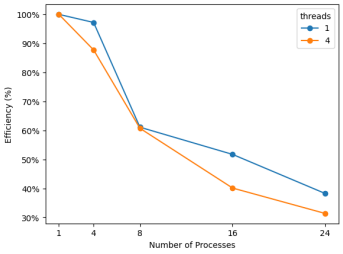


Fig. 4: Time vs. Number of Processes for different numbers of threads

As we can see from Figure 4, 5a and 5b, the parallel solutions MPI-only and MPI with OpenMP are comparable.
When we tested more than 8 processes, the hybrid parallelization works slightly worse with respect to the MPI-only solution. We think that invoking multiple threads at each record is highly expensive in terms of computational resources, and it can slow down the overall computation.



(a) Speedup vs. Number of Processes for different numbers of threads.



(b) Efficiency vs. Number of Processes for different numbers of threads.

Fig. 5: Speedup and efficiency for different numbers of threads.

6 Conclusions

In this report, we designed, implemented and tested three different solutions to perform climate analysis using structured data on precipitation flux records. As expected, parallel versions outperform the serial one.

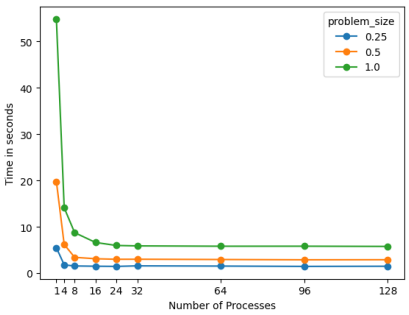
Overall, we are satisfied with the results obtained. In addition, it can be argued that simply increasing the number of processors does not always lead to improved performances. Rather, it is important to balance the allocation of resources, while also accounting for any potential overheads introduced. This assertion is supported by the results of the benchmark test.
For completeness, we processed the final NetCDF output file obtained by one of the configurations tested (the same output is expected from all the versions and configurations adopted) with Panoply data viewer [5], a tool developed by NASA, to plot the mean precipitation flux over 60225 days (original version of the dataset), reported in the Data visualization section, Figure 8.

References

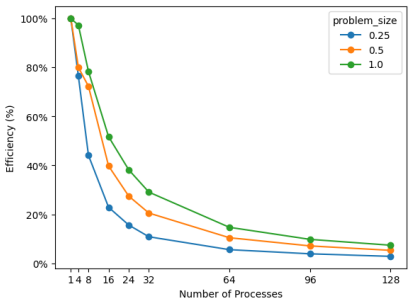
[1] "United Nations. What Is Climate Change?" https://www.un.org/en/climatechange/what-is-climate-change, accessed: 2023-04-05.
[2] "Parallel Programming with OpenMP," https://web.njit.edu/~shahriar/class_home/HPC/omp2.pdf, accessed: 2023-04-09.
[3] P. Pacheco, *An Introduction to Parallel Programming*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.
[4] "Detailed information on nodes and queues of the cluster HPC@Unitrento," https://docs.google.com/spreadsheets/d/1-n4q-nFSHiIugUh_PSmsFS1VwP_ZUitkCSyDvz2As0E/edit#gid=0, accessed: 2023-04-10.
[5] "Panoply netCDF, HDF and GRIB Data Viewer," https://www.giss.nasa.gov/tools/panoply/, accessed: 2023-04-09.

7 Data visualization

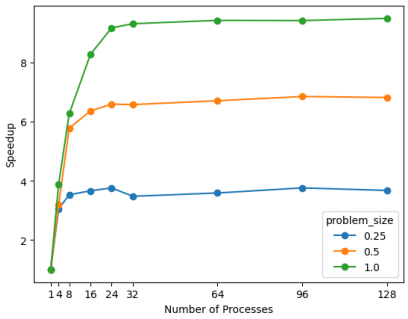
7.1 MPI



(a) Time vs. Number of processes for different problem sizes



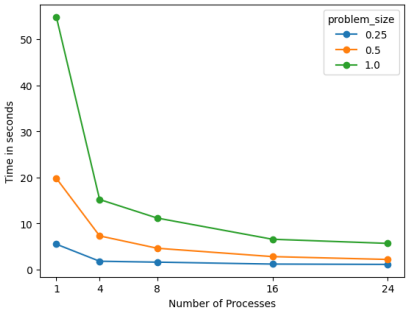
(b) Efficiency vs. Number of processes for different problem sizes



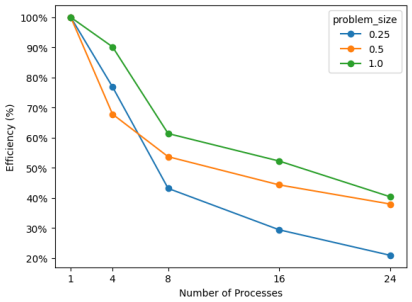
(c) Speedup vs. Number of processes for different problem sizes

Fig. 6: MPI benchmarks

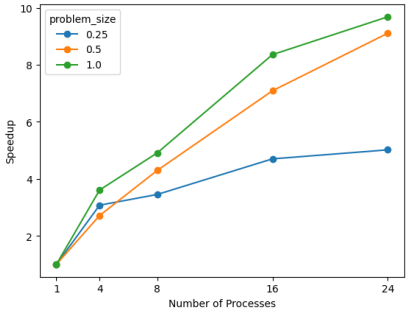
7.2 Hybrid parallelization



(a) Time vs. Number of processes for different problem sizes



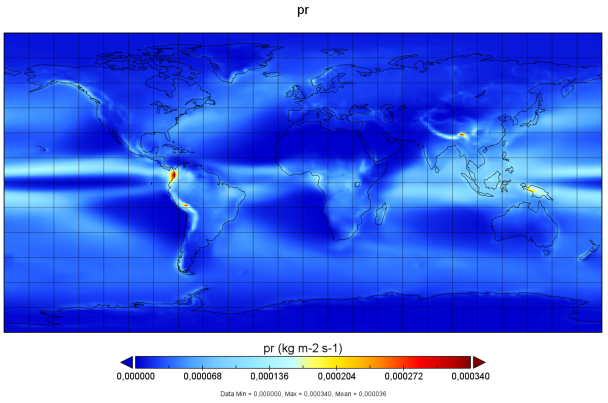
(b) Efficiency vs. Number of processes for different problem sizes



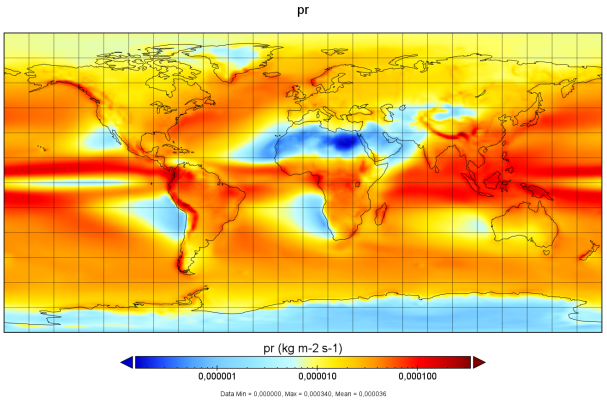
(c) Speedup vs. Number of processes for different problem sizes

Fig. 7: Hybrid parallelization benchmarks

7.3 Final output



(a) Standard scale



(b) Logarithmic scale

Fig. 8: Mean precipitation flux from 1850-01-01 to 2014-12-31 (60225 days).