Climate HPC-HPC4DS

Pietro Demurtas*
pietro.demurtas@studenti.unitn.it
University of Trento
Trento, TN, ITA

Mostafa Haggag*
mostafa.haggag@studenti.unitn.it
University of Trento
Trento, TN, ITA

KEYWORDS

HPC, FESOM. NetCDF, Open MPI, OpenMP

ACM Reference Format:
Pietro Demurtas and Mostafa Haggag. 2018. Climate HPC-HPC4DS. In *Trento '22: Dataming a course about leveraging data, December 08–20, 2022, Trento, IT*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

High-Performance Computing (HPC) has become an increasingly essential part when working with high dimensional data. The main goal is to aggregate computing power in a way that delivers much higher computational power compared to a typical desktop computer. HPC is used for solving large complex problems in different fields from engineering, finance, biology, and many others. We can say that the goal of HPC is to provide the necessary computational power, memory and storage capacity, and network bandwidth to handle extremely large-scale data and processing requirements. When having a lot of resources, it becomes more important to introduce parallelism into your code to allow efficient data processing. Different libraries could be used to introduce parallelism into the code. Open-MPI (Message Passing Interface) is a library designed to provide a standardized interface for inter-process communication on a variety of different parallel computing systems, including clusters. Another library useful tool when dealing with parallelization is OpenMP which is an API for shared-memory parallel programming. OpenMP allows the user to specify the part to paralyze the code to get be executed in parallel, and the OpenMP runtime system automatically handles the details of thread creation, work assignment, and synchronization. OpenMP provides a simple and portable way to write parallel code.

2 DATASET

Our dataset is composed of oceanic data from the FESOM model [1]. FESOM implements the concept of using unstructured meshes with variable resolution. Due to the flexibility of the mesh, it is possible to maintain relatively coarse resolution in less dynamic areas while increasing resolution in those that are dynamic. The data are provided on the native unstructured triangular grid in order to avoid loss of information. We have variables called "lon" and "lat" that store the coordinates of the triangle vertices(nodes). The provider for this grid contains a file for the sea surface data and two files for the horizontal and vertical velocity for all the different levels in the sea. The grid is consisting of 8852366 points.

2.1 Sea Surface Elevation Data

The sea surface elevation data is stored in the `ssh.fesom.2010.nc` file. The data consists of samplings taken at 1-day temporal resolution across all the points in the grid for a total of 365×8852366 data points. Sea surface elevation is expressed in terms of meters.

2.2 Velocity Data

We had two separate files for the vertical `vnod.fesom.2010.nc` and horizontal velocity `unod.fesom.2010.nc` for the complete grid. In contrast, to the sea surface level file, the velocities files do not contain a single level rather 69 levels indicate the speed at the same points but on different heights. Not to mention, these readings contain data sampled over 12 months with a month resolution. This indicates that the file contains 12x69x8852366 data points with 3 different dimensions.

3 PROBLEM ANALYSIS

We start working with Netcdf-4 which does not allow parallel reading which means that we have 1 Netcdf instance to service on all the MPI Processes. In this section, we describe the serial algorithm that we used. Our final goal is to average the three different files across the time and level dimensions to a common resolution in order to plot the data and visually check and compare them. In order to do so we average to a common final dimension of one data point per month for every node in the grid for a total of 12×8852366 data point for each file.

3.1 Sea Surface Elevation Data

3.1.1 Serial Algorithm. In order to reduce the sea surface elevation data to a dimension of 12×8852366 we need to average across the time dimension every 30 days. In order to do we can write a simple serial algorithm (pseudo-code 1) using 2 for and 1 while loops to sum and average the data. The serial algorithm has a complexity of

$$O(3 \times n \times m) \tag{1}$$

$$O(nm) \approx T_S \tag{2}$$

where n and m are respectively the number of days and the number of grid points.

3.2 Horizontal Velocity Data

A serial Algorithm is used to reduce the horizontal velocity data for all of the levels for 12 months to have a final dimension of $12 \times$

*Both authors contributed equally to this research.

Trento '22, December 08–20, 2022, Trento, IT

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

Algorithm 1: Serial SSH average

```
Input  : SSH dataset
Output : Averaged data

average[12][8852366] = 0;
day = 0;
for node ∈ SSH.gridpoints do
    for month ∈ [0:11] do
        while day < 30×(month+1)) do
            average[month][node] += SSH[day][node]/30;
            day++;
        end
    end
end
return average
```

8852366. We show a simple pseudo-code (pseudo-code 2) explaining how a serial process to do this kind of data reduction can be done. The serial algorithm has a complexity of

$$O((n * m * k)) \tag{3}$$

where n, k, and m are respectively the number of months, the number of levels, and the number of grid points.

Algorithm 2: Serial VELOCITY average

```
Input  : Velocity dataset
Output : Averaged data
 average[12][8852366] = 0
for time ∈ [0:12] do
    for level ∈ [0:69] do
        for node ∈ [0:8852366] do
            average[time][node] +=
              velocity[time][level][node]/69
        end
    end
end
return average
```

4 INTRODUCED PARALLELIZATION APPROACHES

4.1 Sea Surface Elevation Data

4.1.1 Semi-sequential approach. In order to parallelize the computation for the sea surface elevation we have considered different approaches. Our first implementation involved computing the average of one month at a time by partitioning each month into each process, computing the local average, collecting the partition averages on process 0, and summing them up. We realized that this approach has several major flaws in the communication pattern as it performs multiple unnecessary communications, in addition computing one month at a time is a very simple way to parallelize the computation and can be done in a more clever way.

4.1.2 Multi-communicator approach. We started developing an approach that would focus on reducing the number of communications between processes to achieve a higher speed up. In order to do so we decided to split the main communicator into groups (or colors) of equal size. The total number of months is then divided among colors. Every process of each color will then get assigned a partition of each month assigned to each color proportional to the number of processes per color. (i.e. with 12 processes and 4 colors process 0 would receive the first 10 days of the first 3 months). Next, each process computes the average of each assigned partition. These averages are then reduced by the sum on the first process of each color (i.e. the process with group rank 0) to obtain one value per node per assigned month. Finally, we create the last communicator consisting of all the processes with group rank 0. We then gather with this communicator all the final values for each month from the process with group rank 0 of every group. The number of groups is a tunable hyperparameter that has to be chosen in accordance with the number of processes. Specifically we have to impose $|groups|\%12 == 0$ and $\frac{12}{|groups|}\%30 == 0$ Pseudo-code 4 illustrates the computational part of the algorithm and it assumes that each process takes as input `partition_SSH` of dimensions $month\_per\_color \times (30/group\_size) \times 8852366$. In the implementation, such an array is provided by parallel access to the input file in such a way that each process reads only the data assigned to its partition. An example of the communication infrastructure with 12 processes and 4 groups can be found in Figure 5 and 7.

4.2 Velocity Data

We tried to introduce more than one method of parallelizing and try to compare them. As we defined previously, our main goal is to reduce the 69 levels of velocities into 1 level averaged over them all, so we have 8852366 grid points and we are summing up each point in the 69 levels and dividing by the total to get the average. Not to mention, we have a similar operation over 12 different time steps.

4.2.1 Simple paralism. We started approaching the problem by implementing an algorithm that runs serially along the time dimension but parallelizes along the levels. This means that we're trying mainly to distribute for each process a given number of levels. These processes should work on summing and then later use an MPI to reduce the sum into process 0. We divided the number of levels per process equally so that we have a distributed load among all of the processes. Process 0 works mainly to collect everything and write it into the NetCDF file to have the final results that we use to plot the maps. We can see in fig.8 the layout of 16 processes that read separately from different levels and start summing them up. We use an MPI reduce operation, in the end, to reduce all the levels into a single one. The pseudo-code for this simple algorithm can be found in pseudo-code 5.

4.2.2 Splitting communication. We noticed that in the previous version, there are major drawbacks since we are still running serially along the time dimension. We start thinking about a way to parallelize both dimensions. We introduce the concept of splitting the communication. We have a set of processes with the same color that gets assigned different partitions of the levels while we also have different colors that get assigned different time instances. With this

design, we are able to work efficiently on both dimensions of the file. We can see in fig.9 that this is the layout of 16 processes where we have 4 colors where each color is managing the data for 3 months and each process in the color is managing 18 levels. We can find pseudo code for this simple algorithm in Pseudo-code 6.

4.2.3 Hybrid communication. We wanted to investigate a different approach in contrast to the approach shown in the previous subsection. We thought about keeping the serialization over time but integrating Openmp into the process. We no longer have groups but we have a number of processes with a different numbers of cores. We are mapping for each process a number of cores and since we have more than 1 core we start threading the code where each thread works on a specific number of grid points or specific levels. In general, here we experiment with two approaches to do this. As we mentioned previously, the NetCDF implementation that we have here is serial one so we have 1 client of NetCDF servicing all the processes. In general, there are 2 approaches that can be used where approach 1 we can read a large chunk of data consisting of Nlevels by gridpoints and loop upon it which takes more memory but fewer calls to the NetCDF client or the second approach is to read 1 level at a time causing more calls of the NetCDF client but less memory usage. Up till now, all the previous approaches are based on reading 1 level at a time without having large chunks of data because we noticed that reading large chunks of data was more time-consuming for the over all experiment. However, since we are having more than 1 threading per process, we compare the two methods here. In version one we are threading over the grid points using *omp for* over the gridpoints so we divide all the points over all of the threads equally. In the benchmarking, we experiment with different scheduling approaches. In the second version, we use *omp for* over the levels in contrast to what was mentioned previously. In this case, we have 2 nested for loops indicating that we could have used collapse functionality but the results were not better so we do not mention it here. We can see in fig.10 that this is the layout of 4 processes where we have 4 threads for each process. We can find pseudo code for this simple algorithm in Pseudo-code 7.

5 RESULTS AND BENCHMARKING

We use 3 metrics to compare our results to understand if a method outperforms another method for specific configurations. We first calculate the wall time over the program and it will be explained that the method of calculating the wall time differs between the sea surface elevation data and the velocity data. Secondly, we use both metrics of speed up and efficiency to see how well we parallelize our code using the following formulas shown in eq.4 and eq.5. Not to mention, we calculate the approximation T_P by dividing T_S by the number of CPUs N to know that this is the lower bound that we should be able to achieve.

$$Efficency = \frac{T_S}{T_P * N} \tag{4}$$

$$Speedup = \frac{T_S}{T_P} \tag{5}$$

where T_S stands for the serial time and T_P stands for the parallel time and N stands for number of processes used.

5.1 Sea Surface Elevation Data

Due to the design of the algorithm and the nature of the data set, we have decided to benchmark the algorithm with fixed input size and variable processes and a number of groups. We benchmarked both own-time and wall-time on process 0 since due to the structure of the communication pattern it is the slowest process to run. For this reason, wall time and own time were identical and thus we report only wall time.

Number of processes	number of groups	reading wall-time(s)↓	computation wall-time(s)↓	speedup↑	efficiency↑
1	1	158.757	34.635	1.000	1.000
6	2	135.998	8.932	3.878	0.646
6	3	84.244	7.586	4.566	0.761
8	2	78.495	4.682	7.397	0.925
8	4	69.583	4.436	7.808	0.976
9	3	76.025	3.732	9.281	1.031
10	2	71.523	3.833	9.036	0.904
12	2	73.571	3.223	10.746	0.896
12	4	91.003	2.695	12.852	1.071
15	3	74.062	2.588	13.383	0.892
18	2	63.684	2.735	12.664	0.704
18	3	59.058	2.197	15.765	0.876
20	2	19.822	4.986	6.946	0.347
20	4	20.901	4.151	8.344	0.417
24	2	12.996	5.002	6.924	0.289
24	4	113.037	2.654	13.050	0.544

Table 1: SSH algorithm run-time

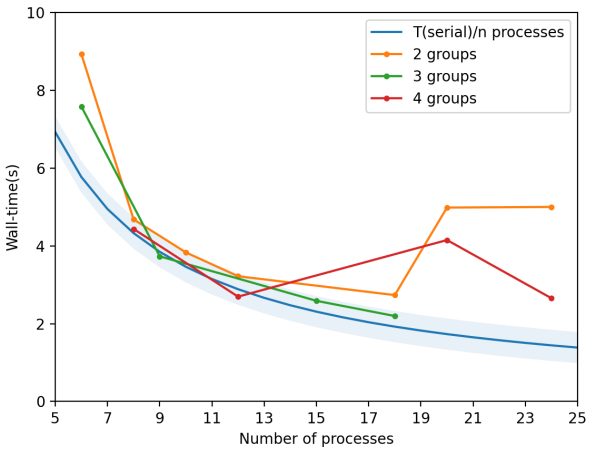


Figure 1: Computational wall-time of SSH algorithm with respect to the number of processes

Table 1 reports the performance for different configurations in terms of reading wall-time(s), computation wall-time(s), speedup and efficiency. Figure 1 visualize the wall-time of the different configurations compared to the theoretical lower bound T_S/N corresponding to the ideal run-time where communications take no time. As we can see we achieve very good speedup and efficiency for N spanning from 8 to 18. For a higher number of processes, we see a drop in efficiency which would be expected as we are not varying the input size and as such $T_{overhead}$ becomes the dominant element of the timing. We don't observe a significant change in performance when varying the number of groups, however, the configurations with 3 groups yield a more stable performance. It is very important to say,

however, that even when explicitly sepcifying placement options on the cluster there is high variability between run-times of different runs with the same configuration. This explains why we are able to achieve some efficiency ≥ 1 which should not be achievable. It is also worth mentioning that for the number of processors greater than 20 we were not able to reserve nodes in a packed exclusive configuration which resulted in better performance. For this reason, we were not able to benchmark for 20 and 24 processes under the same conditions.

5.2 Velocity Data

We calculate a wall time here including the I/O reading time which is indeed a disadvantage but due to the fact that we are reading the data as we go and not all at once. The wall time used here is the total time to average all the data over the 12 months excluding the time of writing the new data into the NetCDF file. We ran the results for different CPUs and we tried to understand how increasing the number of CPUs impacts my results for the different implementations that we are presenting here. We ran the benchmarking three times and we reported the setting with the worst performance. T_S used in all of the calculations was 491.3 seconds as it was the worse performance achieved by the serial algorithm after 12 runs. Table 2 reports the performance for different configurations in terms of speedup and efficiency calculated over the wall time and fig.2 shows the wall time for the different setups that we ran. As we can see in the results, it seems that working with a smaller number of CPUs was better in terms of efficiency and speed-up so even if the wall time is decreasing as we are using more processes, the performance is getting worse. In terms of which implementation is better than the other, we can for sure say that splitting the communication helped to achieve better results in contrast with the rest for all the number of CPUs.

Since the splitting of the communication seemed to be the most efficient implementation, we start benchmarking over different placement configurations to see the impact of changing the configuration. As seen in fig.3, we can see that scatter was able to achieve lower wall time in contrast to pack, and using excl allowed for even better results. We calculated the average reading time over all the processes over the 12 months and we noticed that NetCDF is reading on average faster when using scatter in contrast to pack.

Not to mention, for the hybrid approach, we test different strategies for the scheduling of the OMP like using static, dynamic, and guided configurations for the hybrid v2 version of the algorithm. In addition, we introduce a collapse setting to collapse the 2 nested for loops and check their performance. As we see in fig.4, using different scheduling techniques did not impact so much the wall time although dynamic scheduling achieved the best configurations. It is worth mentioning that using the collapse of the nested loop is not needed and leads to worse results.

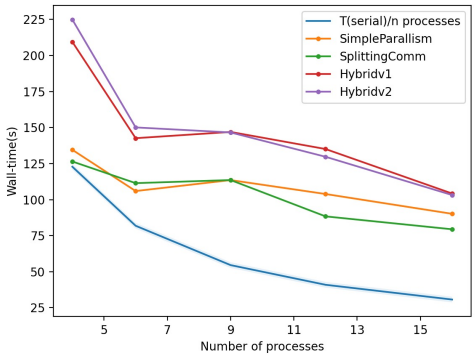


Figure 2: Wall-time of velocity algorithm with respect to the number of processes

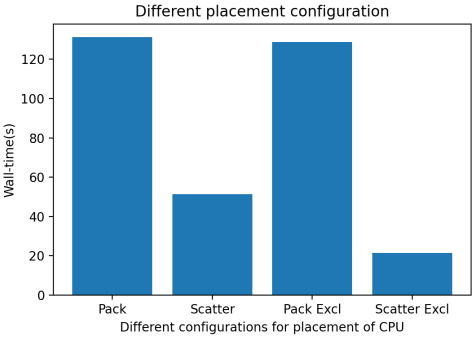


Figure 3: Wall-time of velocity split communication algorithm with respect to different placement configurations.

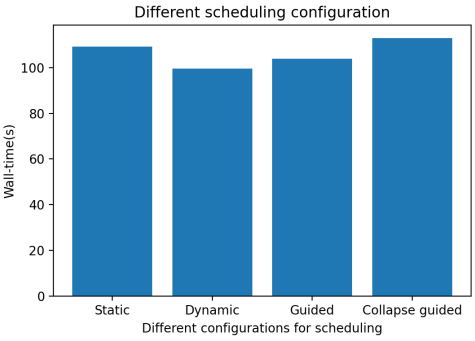


Figure 4: Wall-time of velocity split communication algorithm with respect to different scheduling configurations.

6 CONCLUSION

To conclude, we have implemented a parallel algorithm to reduce each file in the FESOM data set to a visually comparable common shape. The different shapes of the constituent files of the data set required different approaches. We have experimented with different design choices and reported the most promising results. While benchmarking our implementations we noticed that running in a `pack` configuration resulted in faster computational times while running in `scatter` configuration resulted in faster reading time. This explains why the different algorithms, which are timed differently, behave differently according to the configuration. Finally, we were

Number of processes	Simple Parallism		Splitting Comm		Hybrid v1		Hybrid v2	
CPU#	Efficiency	Speed up	Efficiency	Speed up	Efficiency	Speed up	Efficiency	Speed up
4	0.912	3.651	0.971	3.884	0.586	2.345	0.546	2.184
6	0.772	4.636	0.735	4.408	0.574	3.444	0.545	3.272
9	0.480	4.325	0.480	4.325	0.371	3.342	0.372	3.350
12	0.394	4.728	0.463	5.558	0.303	3.636	0.316	3.786
16	0.340	5.448	0.386	6.184	0.295	4.712	0.298	4.760

Table 2: Average velocity speed-up and efficiency results.

able to observe and experiment with the power and utility of parallel programming which allowed us to read and process a huge amount of data in a very with a very short run-time.

REFERENCES

[1] DANILOV, S., KIVMAN, G., AND SCHRÖTER, J. A finite-element ocean model: principles and evaluation. *Ocean Modelling 6*, 2 (2004), 125–150.

7 APPENDIX

7.1 Images

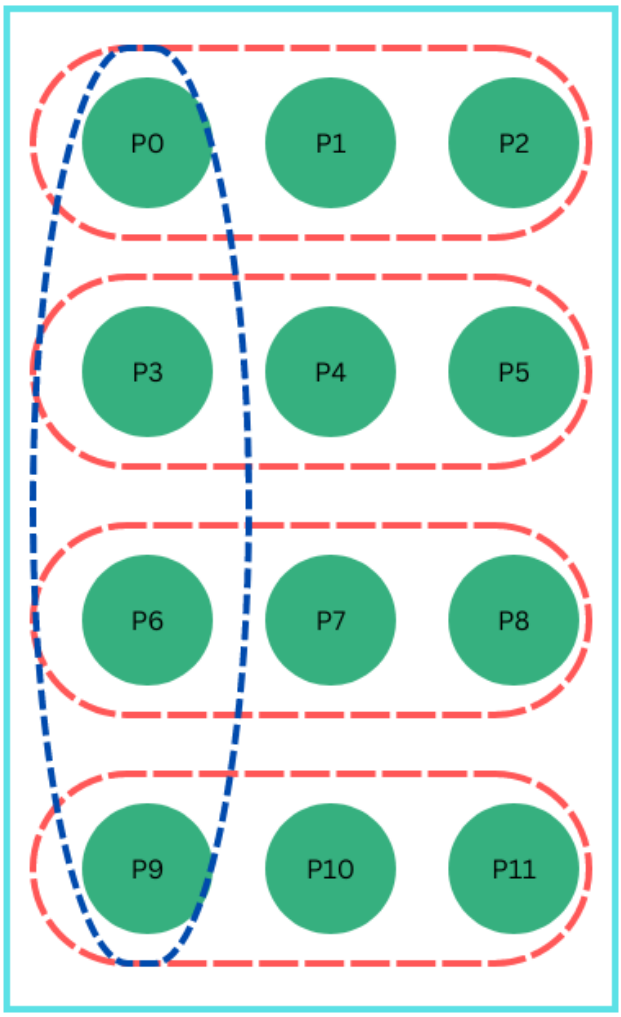


Figure 5: Communication infrastructure for SSH multi-communicator approach. Red lines represent the group communicators, and dark blue line represents row_rank=0 communicator

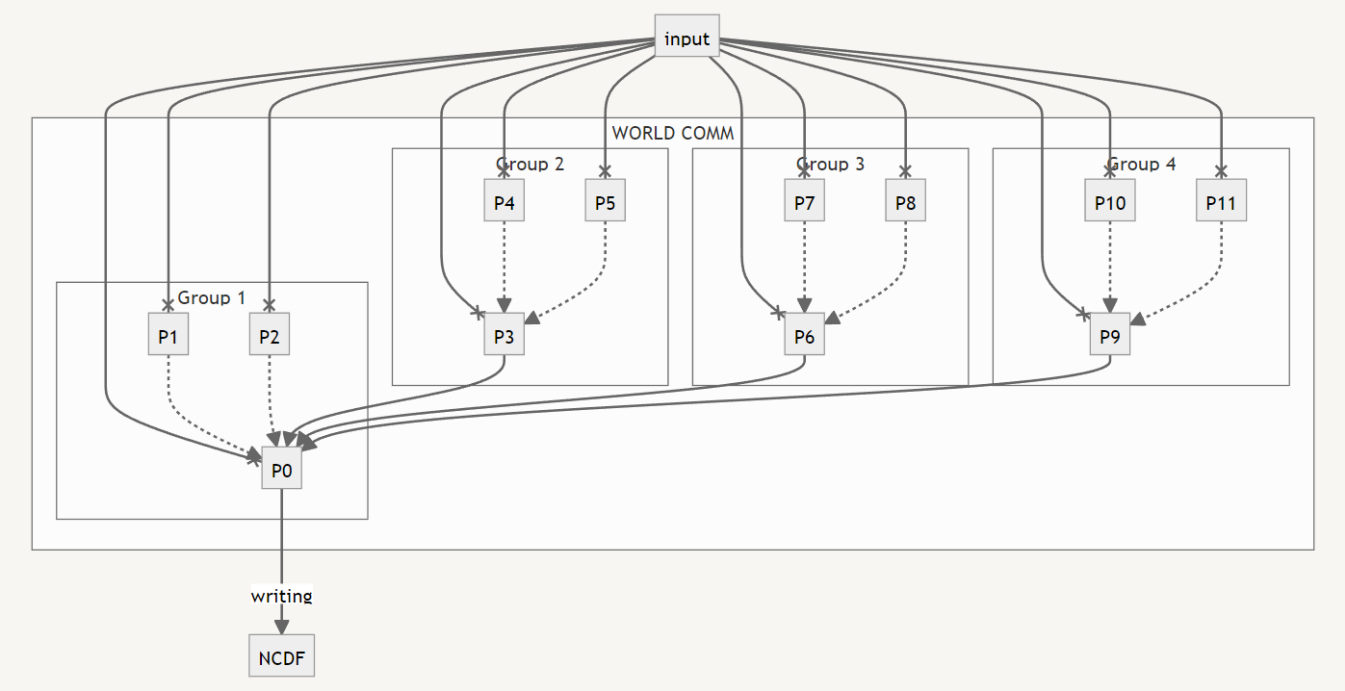


Figure 6: Communication diagram for SSH multi-communicator approach. x-lines represent input, dashed lines represent communication inside groups and solid lines represent communication outside groups.

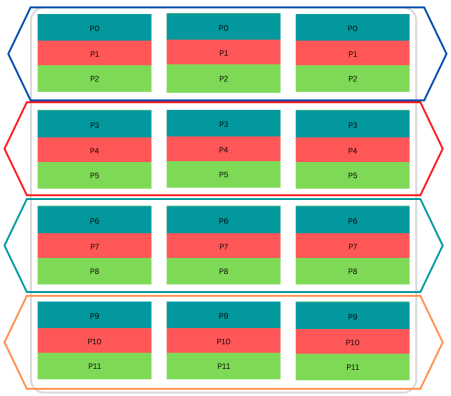


Figure 7: Image showing the partition of twelve month across colors, represented by colored contour, and processes. Case shown has 4 groups and 12 processes

7.2 Pseudo code for reading the data

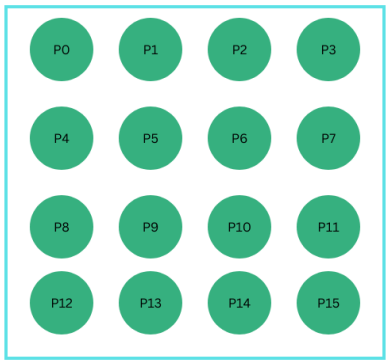


Figure 8: Communicaiton infrastructure for simple velocity multi-communicator approach.

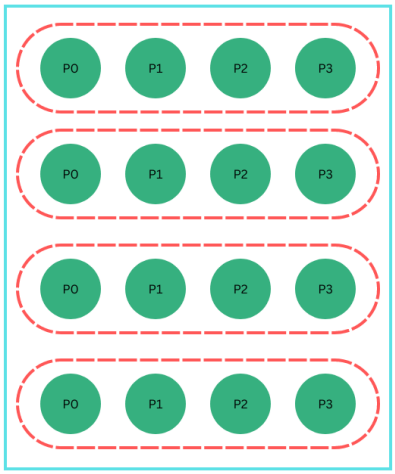


Figure 9: Communicaiton infrastructure for velocity multi-communicator approach with splitting the communication. The red lines indicate that these 4 processes communicate with each other.

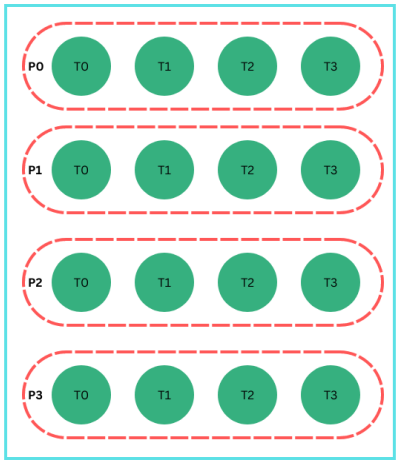
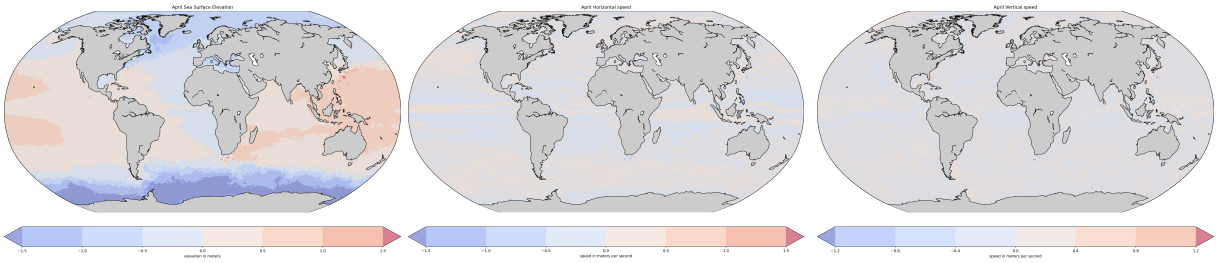


Figure 10: Communicaiton infrastructure for velocity hybrid approach. The red lines indicate all the CPUs are under the same process indicating 1 process with multiple threads.



(a) Sea Surface elevation **(b) Horizontal Speed** **(c) Vertical Speed**

Figure 11: Map plots of January

Algorithm 3: Parallel SSH average

```
Input  : partition_SSH dataset
         n_colors
Output : Averaged data
month_per_color = 12/n_colors;
partition_sum[month_per_color][8852366] = 0;
average[12][8852366] = 0;
local_average[month_per_color][8852366] = 0;
day = 0;
for node ∈ partition_SSH.gridpoints do
    for month ∈ range(month_per_color) do
        while day < 30/row_size do
            partition_sum[month][node] += SSH[month][day][node]/30;
            day++;
        end
    end
end
for i∈ range(month_per_color) do
    MPI_reduce(partition_sum[i]→local_average[i], to process: group_rank = 0, communicator : group);
end
MPI_gather(local_average→average, to process: rank = 0, communicator: group_rank = 0);
return average
```

Algorithm 4: Parallel SSH reader

```
Input  : SSH dataset
         n_colors
Output : partition_SSH dataset

partition_ssh[month_per_color][30/group_size][8852366] = 0;
month_per_color = 12/n_colors;
partition_size = 30/row_size;
color_start_index = color × month_per_color × 30;
color_end_index = (color + 1) × month_per_color × 30;
row_start_index = color_start_index + (row_rank × partition_size);
row_end_index = row_start_index +partition_size ;
for month ∈ month_per_color do
    color_counter = 0 for month ∈ range((month*30)+row_start_index; (month*30)+row_end_index ) do
        partition_ssh[month][day]←NCRead
    end
end
return partition_ssh
```

Algorithm 5: Parallel Velocity average

```
Input  : partition_Velocity dataset
         n_colors
Output : Averaged data
level_per_process = ceil((double)N_NZ1 / size);
for time ∈ 12 do
    total_sum[8852366] = 0;
    local_read[8852366] = 0;
    local_average[8852366] = 0;
    day = 0;
    limit = (rank + 1) * level_per_process;
    if limit > 69 then
        limit = 69
    end
    for process_rank*level_per_process ∈ limit do
        local_read=netcdf_READ();
        for node ∈ 8852366 do
            local_average[node] += local_read[node]/69;
        end
    end
    MPI_reduce(local_average[i]→total_sum, to process: group_rank = 0, communicator : group);
    netcdf_WRITE(total_sum,time);
end
```

Algorithm 6: Parallel Velocity average with comm splitting

```
Input  : partition_Velocity dataset
         n_colors
Output : Averaged data
color = rank/(size/SPLIT_COMM);
MPI_Comm_split();
time_per_proc =ceil((double)N_TIME/SPLIT_COMM);
limit_time = (color + 1) * time_per_proc;
if limit_time > 12 then
    limit_time = 12
end
level_per_process = ceil((double)N_NZ1/row_size);
for color*time_per_process ∈ limit_time do
    total_sum[8852366] = 0;
    local_read[8852366] = 0;
    local_average[8852366] = 0;
    day = 0;
    limit = (row_rank + 1) * level_per_process;
    if limit > 69 then
        limit = 69
    end
    for row_rank*level_per_process ∈ limit do
        local_read=netcdf_READ();
        for node ∈ 8852366 do
            local_average[node] += local_read[node]/69;
        end
    end
    MPI_reduce(local_average[i]→total_sum, to process: group_rank = 0, communicator : group);
    netcdf_WRITE(total_sum,time);
end
```

Algorithm 7: Parallel Velocity average with hybrid implementation

```
Input  : partition_Velocity dataset
         n_colors
Output : Averaged data
level_per_process = ceil((double)N_NZ1 / size);
for time ∈ 12 do
    total_sum[8852366] = 0;
    local_read[level_per_process][8852366] = 0;
    local_average[8852366] = 0;
    day = 0;
    limit = (rank + 1) * level_per_process;
    if limit > 69 then
        limit = 69
    end
    count_levels_per_proc=levels_per_proc;
    if rank == size − 1 then
        count_levels_per_proc = N_NZ1 − levels_per_proc ∗ rank
    end
    for process_rank*level_per_process ∈ limit do
        local_read=netcdf_READ();
        PRAGMA OMP PARALLEL START;
        temp_private[8852366] = 0;
        for level ∈ count_levels_per_proc do
            for node ∈ 8852366 do
                temp_private[node] += local_read[count_levels_per_proc][node]/69;
            end
        end
        PRAMGA OMP CRITICAL START;
        for node ∈ 8852366 do
            local_average[node] += temp_private[node];
        end
        PRAMGA OMP CRITICAL END;
        PRAGMA OMP PARALLEL START;
    end
    PRAGMA END;
    MPI_reduce(local_average[i]→total_sum, to process: group_rank = 0, communicator : group);
    netcdf_WRITE(total_sum,time);
end
```