ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

IAC Institute for Atmospheric and Climate Science

Parallelization of an Isentropic Model with mpi4py

High Performance Computing for Weather and Climate

Spring Term 2024

Supervised by:
Oliver Fuhrer

Authors:
Tim Mutkala
Gianni Ardielli
Nadine Gut

Abstract

With the need to run climate and weather models on finer and finer grids, solutions for the high computational demand are needed. One possible measure is running a model in parallel across multiple nodes. In this report, we refactored an isentropic model that simulates a flow over a mountain ridge, enabling parallelization in the main simulation loop. To investigate the performance of this parallelization measure, we compared the execution time of the original model with our restructured model. Our results show that, for a pre-defined configuration on a supercomputer, the computational time can be reduced by 97.2%, making the model 36 times faster.

Contents

1 Introduction

Weather models are essential to understand and predict weather phenomena. They provide a way to approximate physical processes and enable forecasts, as well as detailed investigations. Achieving a high spatial resolution is crucial for detecting localized, small-scale phenomena. However, even a resolution of 1 km, which is considered very fine, may not capture all local and dynamic effects. Today's weather and climate models must solve differential equations for each grid point at each time step. As the demand for more accurate weather forecasts increases, especially for small-scale events, the grid spacing becomes smaller and smaller. As a result, the computational demand grows rapidly. Therefore, in order to run finer-grided atmosphere or climate models within a reasonable time frame, optimization techniques such as parallelism are essential.

In the course "Climate and Weather Models" by Christoph Schär, we utilized a two-dimensional isentropic model to simulate both dry and wet flows over a mountain ridge. This model, originally developed by Christoph Schär himself, has been extended and refined over the years. Recognizing the time consuming nature of the simulations, especially with increasing domain size and timesteps, our group aimed to contribute by parallelizing this model. We implemented the parallelization using 'mpi4py', a tool that allows Python applications to use multiple threads, processors and - on supercomputers - even multiple nodes, blades and cabinets. (Dalcin and contributors, 2024).

The goal of our project was to parallelize the code provided in the tutorial of the "Climate and Weather Models" course and to compare the execution time of the original simulation with our optimized, parallelized version.

2 Methods

The isentropic model simulates a 2D flow of a stream over a mountain ridge on an xz-plane, and uses isentropic coordinates $\phi(x, \theta, t)$. As it is built on a c-grid, u is horizontally and $\dot{\theta}, \theta$ and p are vertically staggered. For discretization, centered differences in time and space are used. As this scheme is explicit, parallelization becomes much easier.

2.1 Domain Expansion

The default grid of the isentropic model consists of 100 grid points in x-direction with 60 layers. As a result, the model already runs very fast. In reality, models are parallelized when the computation of a step is really expensive. Therefore, such a model was imitated by increasing the number of grid points to 5'040. This specific value was chosen to enable a uniform distribution of gridpoints among a large variety in the number of workers. To meet the CFL criterion, the domains had to be increased by the same factor. Since no mountains extend over such a large area, we changed the topography to a mountain range with three distinct peaks.

2.2 Parallelization

2.2.1 Analysis and Profiling of the Model

In a first step, we manually analyzed each function invocation inside the main loop, to identify where parallelization is possible and where problems might occur. We concluded that only the horizontal diffusion and the two-moment microphysics scheme could cause problems because they need information of the neighbouring values of prognosed fields (ϕ_{i-1}^{n+1}). For simplicity, we decided to only use the Kessler micorphysics and apply the horizontal diffusion after exchanging the boundaries. Figure 2.1 shows the performance profile of the unoptimized simulation for the used simulation configuration. With the aforementioned decisions, no problems should arise and assuming the Kessler scheme runs well in parallel, a performance improvement could become noticable.

2.2.2 Parallelized Model Structure

The optimized model starts with the initialisation of the global fields, followed by the scattering to local fields. Then, in a time loop, prognostic steps of s, u, qc, qv and qr are calculated. After exchanging the boundary values between neighbouring workers, boundary relaxation and horizontal diffusion are applied and the diagnostic variables are calculated. Every 360 timesteps global fields are gathered at the main process to generate an output. Figure 2.2 depicts a flowchart of the parallelized simulation.

2.2.3 Scattering

Before the main time loop starts, the dataset ($nx = 5'040$ columns) is divided into chunks along the x-axis, corresponding to the slices needed by each worker. Accordingly, each chunk ends up with $nx/\#chunks + 2 \times nb$ x-coordinates and 60 layers, where nb refers to the number of boundary points on each side. The number of chunks, i.e. the number of workers, must divide nx. We renamed the initial variable definitions to have a "$\_g$" suffix and the new partial, worker-specific chunks to have "$\_p$" suffix, in order to identify them as global and local fields. Since the horizontal velocity 'u' is staggered in the x-direction, a

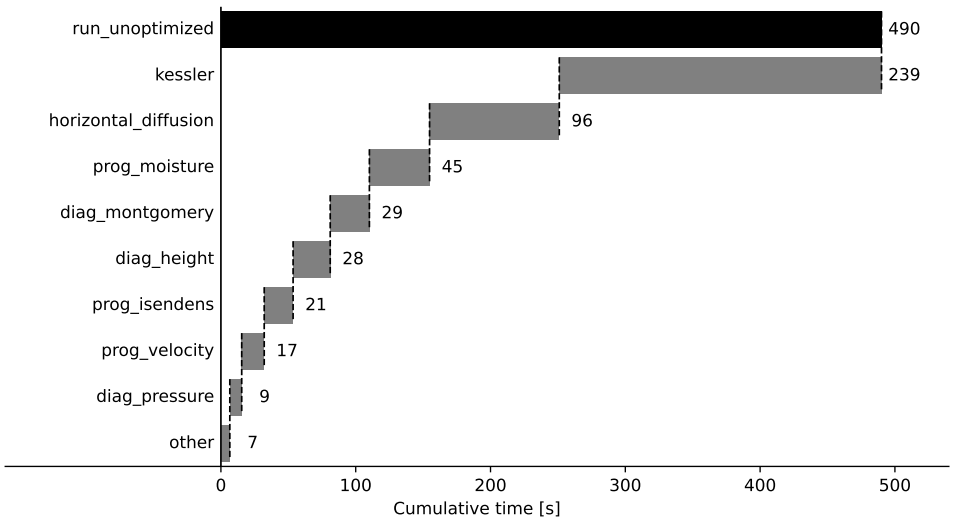


Figure 2.1: Breakdown of the simulation profile, illustrating how the total computation time of the model ('run_unoptimized') is distributed among the function invocations inside the main simulation loop. Running the Kessler scheme in parallel should bring the largest performance improvement.

prognostic step needs at least the information of two boundary points of the neighbouring junk. Therefore, we provided each worker with the two boundary points of the neighbouring workers on both sides. For simplicity, we also provided two boundary points for all other variables (s, qc, qv, qr), even if they are unstaggered.

2.2.4 Sending and Receiving Data

To improve the efficiency of parallelization, the data is scattered at the beginning and gathered only for iterations where an output is generated. Therefore, in the time loop, the domain remains divided and the workers use the MPI 'Sendreceive' method to exchange the boundaries of their newly computed fields ($snew$, $unew$, $qcnew$, $qvnew$, $qrnew$). At no point after the initialization is data scattered to all nodes at once. The two right/left boundary points are always sent to the right/left worker as new left/right boundaries (see Figure 2.3). To damp the right and left boundary of the global field, we have adapted the relaxation method from the initial model. The left- and rightmost workers always relax their left/right boundaries if the relaxation flag is set.
There is only one small difference between the initial and the parallelized model. Unlike the parallelized model, the initial model does not exchange its horizontal boundaries when relaxation is activated. But this difference should not strongly affect the results.

2.2.5 Gathering

The model generates an output every 360 time steps. Therefore, at the end of these time steps, the variables are gathered again to create global arrays. The two overlapping boundary points must be taken into account in the gathering to ensure that no column is duplicated. For the staggered variable 'u', only one boundary point has to be dropped. The gathered global fields are used only for the output, the next time step uses the local fields again.

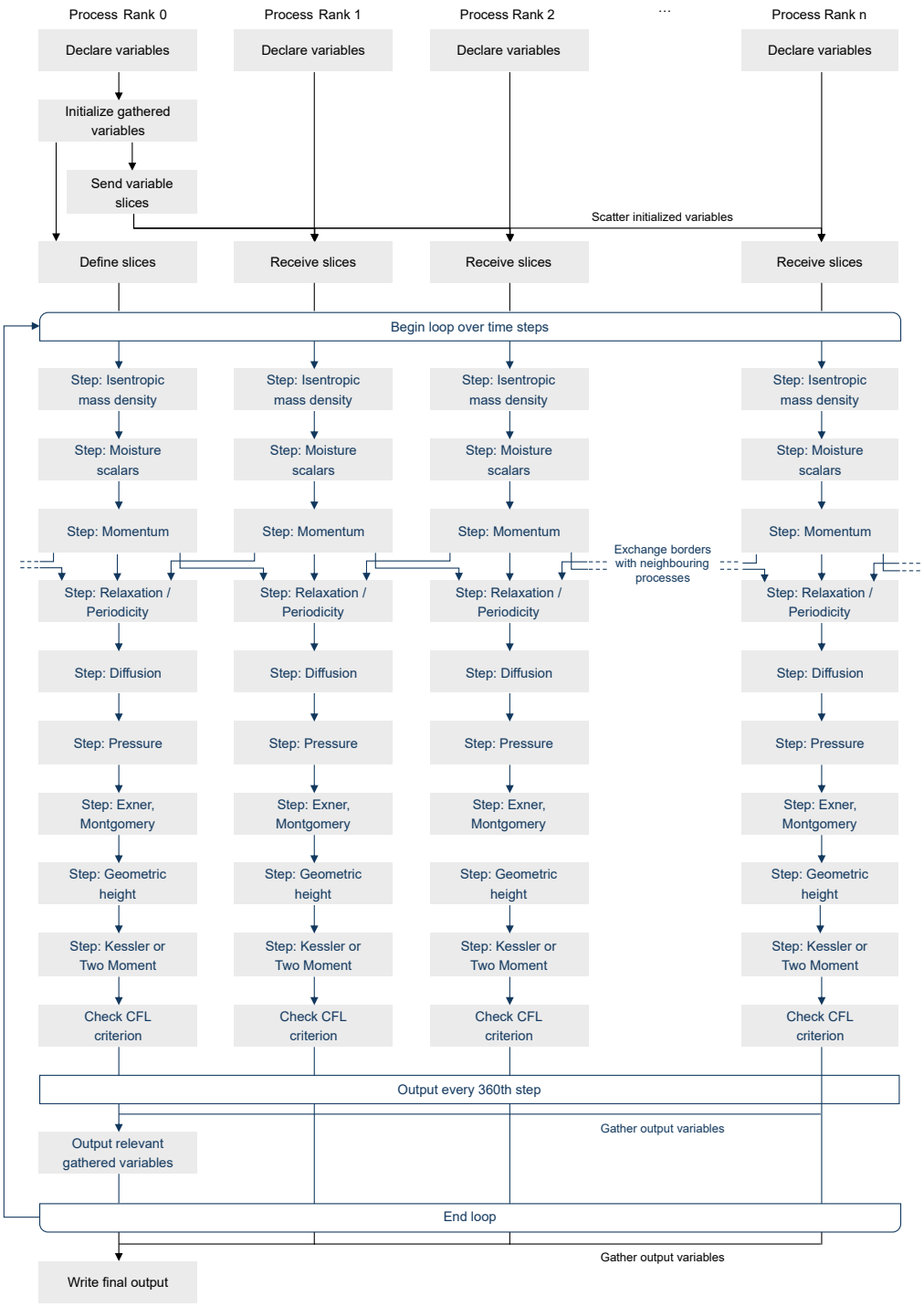


Figure 2.2: Flowchart of the parallelized model. After initializing all variables, the main process distributes the variable slices to all worker processes. During the main simulation loop, each worker computes in parallel and the relevant boundary values are exchanged once per iteration. Whenever an output is written, all needed variable values are collected at the main process. After a set number of iterations, the main process finalizes and the simulation is terminated.

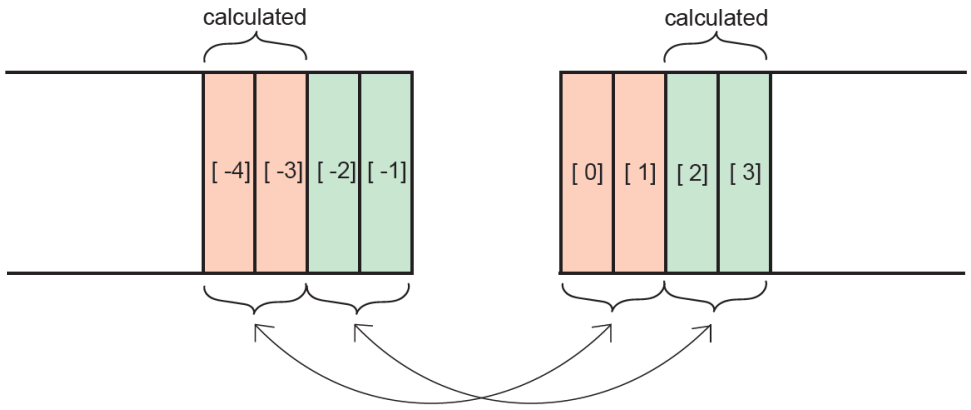


Figure 2.3: Modified scattering. Each worker receives two overlapping points from its neighbour (red, green). Here the left worker calculates the red columns and the right worker calculates the green columns. This way each column is calculated in a time step. To avoid deadlocks, in practice each worker first sends its border values to its left neighbour, awaiting those coming from the right neighbour and then sends its border values to the right neighbour, awaiting those from the left neighbour.

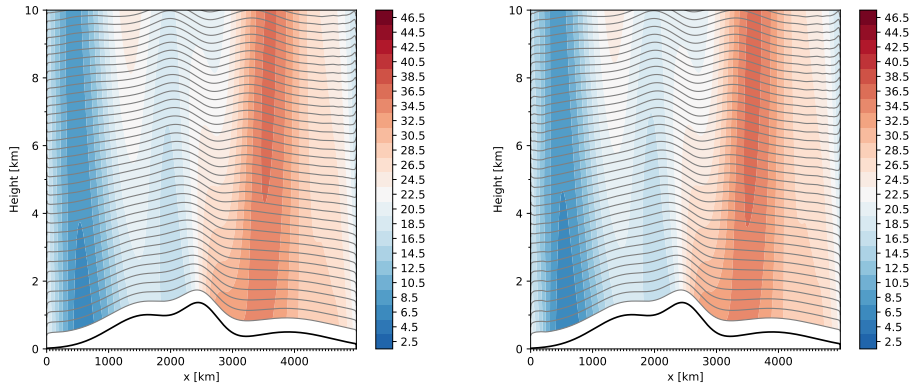
2.3 Performance Evaluation

In order to evaluate the performance of the model, a comparison was conducted between the total computation time of the initial model and the parallelized model on one of our computers, as well as on the supercomputer Piz Daint. In order to perform the comparison, an integration time of 7'200 seconds with a time step of 2 seconds was selected. Moreover, the number of workers was selected so that the grid points in the x-direction were divisible by the number of workers. First, the original model was run on one of our consumer-grade computers and then on Piz Daint. The final computation time used is an average of four model runs. The same procedure was then followed for the parallelized version with a different numbers of workers (2,3,4,...,24). Again, this was done as well as on our personal computer, as on Piz Daint. A maximum number of workers of 24 was chosen, corresponding to the number of threads on one node on Piz Daint. In a second measurment, the parallelized version was run across multiple nodes on Piz Daint. float

3 Results and Discussion

3.1 Model Validation

Before we start to compare the performance of the initial simulation and the optimized version, we need to ensure that both actually produce the same result. Figure 3.1 depicts the xz-plot for both the original, and the adapted version, after having completed the simulation for an identical configuration. No difference is noticable.



(a) Unoptimized Model

(b) Optimized Model

Figure 3.1: The resulting xz-plots shows a two dimensional flow over a mountain range in a vertical crosssection. For both the unoptimized and optimized model executions for the same configuration of 3'600 time steps and 5'040 grid points.

3.2 Single-node Performance

The initial model has a computation time of 601 ± 4 seconds on Piz Daint and $1'160 \pm 40$ seconds on a personal computer (four simulation runs each, see Figure 3.2). When using the parallelized version, the computation time is approximately halved on Piz Daint when using two workers (reduction of around 30% on the personal machine). The lowest computation time of 70.67 ± 0.16 seconds is achieved on the supercomputer for 12 workers, which corresponds to the number of cores on a node. This corresponds to a decrease in computation time of 88.2%. It should be noted that this is when running in an environment that only has one node at its disposal. For the personal machine, it was 14 workers at 221.7 ± 2.2 seconds (80.9% decrease compared to the unoptimized version). Here again 14 corresponds to the total number of cores.

3.3 Multi-node Performance

In order to assess the effect of using multiple nodes, a Jupyter sessions was started on Piz Daint with five nodes. Subsequently, the computation time was measured for the unoptimized model and the optimized

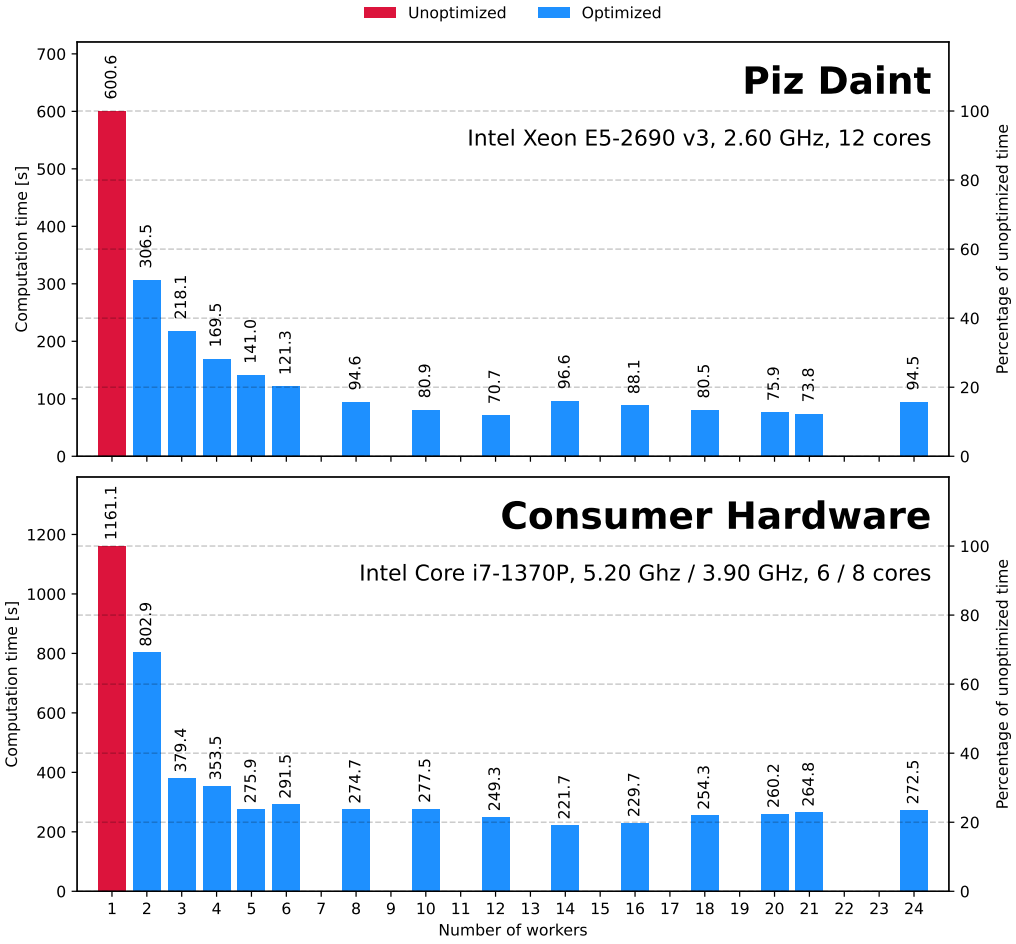


Figure 3.2: Comparison of the model performance on a single node at 3'600 time steps and 5'040 grid points. The unoptimized model was run on both a personal laptop and the Piz Daint supercomputer. Subsequently, the optimized version was run for different numbers of workers. Each value is an average of four simulation runs. For each run, the time inside the main loop of the slowest worker was taken. The times across different workers for a given run were always within 0.05 seconds of each other. The consumer-grade computer has six performance cores at 5.20 GHz and eight efficiency cores at 3.90 GHz, while the XC50 compute nodes in Piz Daint have twelve cores each at 2.60 GHz.

version. For the latter, each possible number of workers that divides the number of gridpoints (5'040) was chosen, up to 180. Figure 3.3 depicts the measurement results.

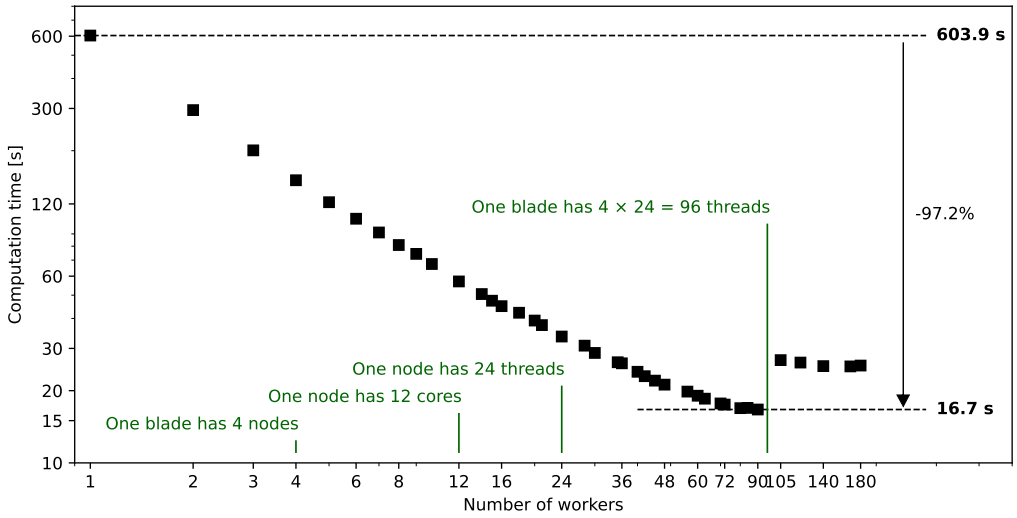


Figure 3.3: Log-log plot of the number of workers vs. the total computation time of the simulation running at 3'600 time steps with 5'040 grid points.

We observe a steady decline in computation time, all the way until 90 workers, where we encounter a sudden increase when running with 105 workers. As an XC50 compute node has four nodes on a blade and each node has 24 threads, we observe that this jump happens exactly at the point where we start using a second blade, assuming one worker per thread. This increases the cost of transferring borders between processes and we become strongly i/o-bound. The computation time at 90 workers is just 16.7 seconds, which is a 97.2% decrease from the 603.9 seconds when running the unoptimized version.

3.4 Potential for Improvement

Even though our model already reduces the computation time significantly, there are some improvements that could be considered:

- We used two boundary points on both sides for all variables, even though all horizontally unstaggered variables would only need one grid point. Taking this into account would further reduce the amount of data sent during the 'Sendreceive' calls.
- An individual non-uniform partitioning of the chunks could further improve the model speed. In particular, chunks that are close to the mountain have to calculate more operations because their microphysics calculations are more expensive. On the other hand, chunks at the boundary can often ignore the microphysics. By applying a higher resolution of partitioning near the mountain, the computation time would be more evenly distributed.
- The parallelization could be even faster when using Fortran instead of Python.
- In our evaluation, we focused only on parallelization with 'mpi4py'. Applying cache optimisation could further improve the model's time performance.

4 Conclusion

In this project, we aimed to parallelize the model of an isentropic flow over a mountain ridge that is utilized in the course "Climate and Weather Models". With 'mpi4py' we had a way to distribute the computational demand onto different workers by dividing the dataset into several chunks along the x-axis.

By comparing the execution time of the original code with our optimized parallelized version we found that for 90 workers, spread across multiple nodes, the reduction of computation time was highest at 97.2% compared to the unoptimized model, making the model evaluation 36 times faster. This exceeded our expectations and shows that parallelization with 'mpi4py' is useful and decreases the computational time significantly. For all measurements above 90 workers the computation time is higher and plateaus. This is due to the fact that we become strongly i/o bound as we are now sending data across two blades. Nevertheless, the parallelization could possibly be even faster when using Fortran instead of Python and accounting for caching behaviour and the fact that workers near the mountains might take longer than those at the boundaries.

Further, we compared the reduction of computation time when running on a single XC50 compute node, of the Piz Daint supercomputer and a consumer-grade Lenovo ThinkPad X1 Carbon, where we found a reduction around in computation time of around 80% in both cases. This revealed that not only is parallelization useful for both super- and regular computers, but in both cases the computation time became minimal when the number of workers corresponded to the number of cores.

The main challenge in this project was understanding the simulation on a physical level and ensuring that any changes made are not just syntactically correct, but do not reduce the quality of the model. We also ran into deadlocks a few times, but this was straight-forward to solve.

Future experiments with this code could benefit from the parallelization by being able to run the isentropic model with much more complicated topography and finer grids.

5 Acknowledgments

We would like to acknowledge and give thanks to our supervisor Oliver Fuhrer who guided us through our project. Further, we would also like to thank Christoph Schär to allow us to parallelize his code and Tuule Mürsepp who provided us with the most recent version of it.

Bibliography

L. Dalcin and contributors, "Mpi for python (mpi4py)," 2024, accessed: 2024-09-01.

A Appendix

Tables A.1 and A.2 contain the measured computation times for single- and multi-node measurements respectively.

Machine	Model Version	Number of Workers	Duration [s]
x1carbon	unoptimized	1	$(1.16 \pm 0.04) \times 10^3$
x1carbon	optimized	2	$(8.0 \pm 1.9) \times 10^2$
x1carbon	optimized	3	$(3.8 \pm 0.4) \times 10^2$
x1carbon	optimized	4	$(3.5 \pm 0.4) \times 10^2$
x1carbon	optimized	5	275.9 ± 3.1
x1carbon	optimized	6	291 ± 20
x1carbon	optimized	8	275 ± 14
x1carbon	optimized	10	277 ± 11
x1carbon	optimized	12	249.3 ± 1.6
x1carbon	optimized	14	221.7 ± 2.2
x1carbon	optimized	16	229.7 ± 1.0
x1carbon	optimized	18	254 ± 13
x1carbon	optimized	20	260.2 ± 2.6
x1carbon	optimized	21	264.8 ± 2.3
x1carbon	optimized	24	272.5 ± 1.6
pizdaint	unoptimized	1	601 ± 4
pizdaint	optimized	2	306.5 ± 2.7
pizdaint	optimized	3	218.1 ± 1.0
pizdaint	optimized	4	169.5 ± 0.9
pizdaint	optimized	5	140.97 ± 0.19
pizdaint	optimized	6	121.3 ± 0.6
pizdaint	optimized	8	94.6 ± 0.4
pizdaint	optimized	10	80.95 ± 0.27
pizdaint	optimized	12	70.67 ± 0.16
pizdaint	optimized	14	96.6 ± 0.6
pizdaint	optimized	16	88.11 ± 0.18
pizdaint	optimized	18	80.5 ± 0.4
pizdaint	optimized	20	75.94 ± 0.12
pizdaint	optimized	21	73.84 ± 0.28
pizdaint	optimized	24	94.5 ± 0.5

Table A.1: Computation time of the unoptimized and optimized models on Piz Daint and a personal machine for different numbers of workers. Each configuration was measured four times and the displayed values are the mean and standard deviation. The Jupyter server on Piz Daint was started with one XC50 compute node.

Machine	Model Version	Number of Workers	Duration [s]
pizdaint	unoptimized	1	603.88
pizdaint	optimized	2	295.27
pizdaint	optimized	3	200.66
pizdaint	optimized	4	150.63
pizdaint	optimized	5	122.00
pizdaint	optimized	6	104.11
pizdaint	optimized	7	91.25
pizdaint	optimized	8	80.90
pizdaint	optimized	9	74.30
pizdaint	optimized	10	67.42
pizdaint	optimized	12	57.04
pizdaint	optimized	14	50.54
pizdaint	optimized	15	47.41
pizdaint	optimized	16	45.00
pizdaint	optimized	18	42.24
pizdaint	optimized	20	39.19
pizdaint	optimized	21	37.52
pizdaint	optimized	24	33.62
pizdaint	optimized	28	30.80
pizdaint	optimized	30	28.72
pizdaint	optimized	35	26.27
pizdaint	optimized	36	26.00
pizdaint	optimized	40	23.97
pizdaint	optimized	42	22.98
pizdaint	optimized	45	22.02
pizdaint	optimized	48	21.19
pizdaint	optimized	56	19.82
pizdaint	optimized	60	19.05
pizdaint	optimized	63	18.52
pizdaint	optimized	70	17.69
pizdaint	optimized	72	17.50
pizdaint	optimized	80	16.91
pizdaint	optimized	84	16.95
pizdaint	optimized	90	16.69
pizdaint	optimized	105	26.81
pizdaint	optimized	120	26.19
pizdaint	optimized	140	25.31
pizdaint	optimized	168	25.24
pizdaint	optimized	180	25.45

Table A.2: Computation time of the unoptimized and optimized models on Piz Daint for different numbers of workers. The Jupyter server on Piz Daint was started with five XC50 compute nodes.