An MPI Version of the GT4PY Stencil Code

Group 7 Project Work for the Course "High Performance Computing for Weather and Climate",
Institute for Atmospheric and Climate Science, ETH Zürich

Ruben Strässle (rubenst@student.ethz.ch)
Yilu Chen (yilchen@student.ethz.ch)
Marc Federer (marc.federer@env.ethz.ch)

Supervised by Tobias Wicky

Grid Tools For Python (GT4PY) is a domain-specific library used for grid based finite-difference Computations, also referred to as stencil codes, designed for Weather and Climate Models. In this Project, the GT4PY version of a stencil code for solving a diffusion equation was parallelized with the Message Passing Interface (MPI). It was ensured that all code was properly ported, including the halo-updates, by validating it with the base code without MPI and GT4PY. The code was analyzed and optimized by leveraging the full capacity of GT4PY data storage and management structures. Consequently, a strong and weak scaling analysis was done on Piz Daint supercomputer with both versions of the Code and different backends. The Laws of Amdahl and Gustafson could be recovered and explained by the obtained data and the resulting figures. Further, conclusions about both code versions and the different backends were drawn.

Introduction

Domain-specific languages (DSL) are gaining popularity due to their high level and ease of use. The Goal of a DSL is to increase productivity, performance and portability. Similar to TensorFlow for Machine Learning, LaTeX for text processing or OpenFoam in Fluid Mechanics, Grid Tools For Python (GT4PY) is a Python Application Programming Interface (API) for grid based finite-difference Computations, also referred to as stencil codes, primarily developed for applications in the area of weather and climate.

GT4PY generates high performance implementations of stencil kernels from a high-level user-definition using the GridTools framework. GridTools implements a DSL embedded in C++ and allows for the compilation with different backends to generate optimized code for a specific architecture at compile-time. Further, it also provides modules for boundary conditions, halo-updates, data storage and management, and bindings to other programming languages such as Fortran, on which most weather and climate models still run today. [1]

Weather and climate models require to solve a set of conservation laws called the Navier-Stokes-Fourier system as well as some additional equations accounting for the micro-physics. Atmospheric and ocean models often need some form of numerical filtering to control small-scale noise [2]. Xue [3] introduced a class of higher-order monotonic filters that are frequently used for this task. Discretizing and solving such a large system of equations has its own problems and peculiarities, which is not the scope of this work. Thus, during the block course [2] and in the present project, a two-dimensional 4th-order non-monotonic diffusion equation, as introduced by Xue [3], was used as a modeling equation and as a representation of a much more complex system for a weather an climate model. The modeling equation can be written as

$$\begin{aligned} \frac{\partial \phi}{\partial t} &= -\alpha_4 \nabla_h^4 \phi \\ &= -\alpha_4 \Delta \left(\Delta \phi \right) \ , \end{aligned} \tag{1}$$

where $\Delta_h = \nabla_h^2$ are the Laplace and Nabla Operators, respectively, acting on a any scalar quantity ϕ. Assuming an equidistant grid with $h = \Delta x = \Delta y$, the Laplace operator can be discretized as

$$\Delta_h \phi_{i,j}^n \approx \frac{\phi_{i-1,j}^n + \phi_{i,j-1}^n - 4\phi_{i,j}^n + \phi_{i+1,j}^n + \phi_{i,j+1}^n}{h^2} \tag{2}$$

with the central differences scheme (CDS). The resulting stencil from complete discretization of Equation 1 is marked in Figure 1 by the blue dots for the example of $\phi_{i=0,j=2}^{n+1}$.

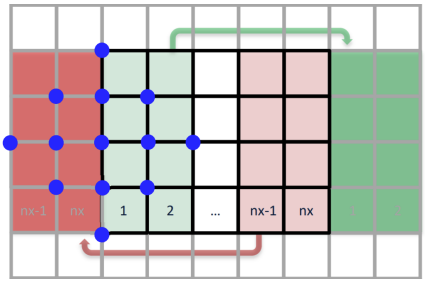


Fig. 1: *Schematic of the compute-domain indicated in black; the halo zones in gray; the halo-updates in red and green; and the stencil for of $\phi_{0,2}^{n+1}$ from Equation 1 with CDS in blue ([2], adapted).*

It is apparent that a part of the stencil is outside the compute-domain marked in black; this area is also referred to as the halo zone. Imposing periodic boundary conditions, one can now do the halo-updates by copying the last part of the compute-domain at the right side to the left halo zone, as indicated in red. The same can be done in all directions. When parallelizing the domain using the Message Passing Interface (MPI), the updates have to be made in the same manner not only at the real boundaries, but also inbetween the regions of individual MPI processes.

Parallelization of Halo-Updates

In order to achieve multi-node scalability of the halo-updating procedure, we implemented the MPI protocol for `stencil2d-gt4py-v0.py` [1]. The first necessary adaption is a reasonable partition of the initial input field. This is achieved using the `Partitioner` class [2]. The input field is first allocated as a `numpy.ndarray` on rank 0, scattered on all ranks using the `Partitioner` class and lastly the partitioned fields are allocated as `gt4py.storage` containers. A limiting factor of the `Partitioner` class is that the horizontal input field dimensions must be divisible by the number of ranks utilized, such that an even partition of the input field can be achieved. Suitable error messages were included in the code.
In a first approach, we utilize the `numpy.ndarray` functionality of the `gt4py.storage` module to perform the halo-updates. Doing so allows us to coerce a GT4PY array into a plain NumPy array, which can be used to perform send and receive operations using the `mpi4py` package. The received halo-updates are eventually assigned to the input field `gt4py.storage` containers using `numpy.ndarray` syntax, as shown in Listing 1. The code for this version is available on GitHub [3].

```
# bottom & top
field[num_halo:-num_halo, 0:num_halo, :] = b_rcvbuf
field[num_halo:-num_halo, -num_halo:, :] = t_rcvbuf
# left & right
field[0:num_halo, :, :] = l_rcvbuf
field[-num_halo:, :, :] = r_rcvbuf
```

Listing 1: *Assigning halo-updates to `gt4py.storage` containers using `numpy.ndarray`.*

Performance Optimization - Leveraging GT4PY

The implementation as described in the previous section is not fully leveraging the capabilities of GT4PY, since the halo-updates are assigned to `gt4py.storage` containers using `numpy.ndarray`. A more refined version would use a GT4PY stencil object to perform the halo-update. We implemented this functionality based on `stencil2d-gt4py-v3.py` [4]. First, the partition of the input field and the send/receive operations using MPI were implemented in analogy to the first version. Second, the received halo-update `numpy.ndarrays` were allocated as `gt4py.storage` containers. Lastly, the stencil object `copy_stencil` was used to copy the halo-update fields into the input fields as shown in Listing 2. This version fully leverages the capabilities of GT4PY in performing the halo-updates and ensures scalability across multiple nodes. The code for this version is available on GitHub [5].

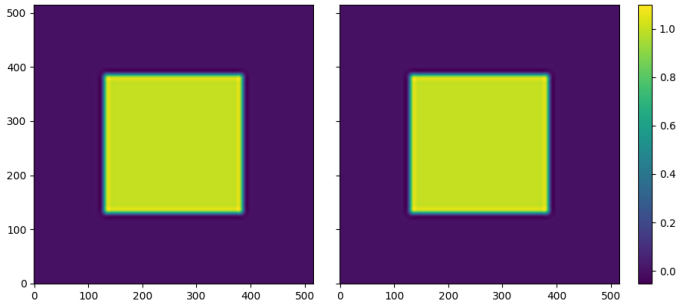
```
nx_p = field.shape[0]-2*num_halo
ny_p = field.shape[1]-2*num_halo
nz_p = field.shape[2]

# bottom
b_rcvbuf_storage =
    gt.storage.from_array(b_rcvbuf,
                          backend = backend,
                          default_origin = (0,0,0))
copy_stencil(
    src=b_rcvbuf_storage,
    dst=field,
    origin={"src": (0, 0, 0),
    "dst": (num_halo, 0, 0)},
    domain=(nx_p, num_halo, nz_p),
)

# top
t_rcvbuf_storage =
    gt.storage.from_array(t_rcvbuf,
                          backend = backend,
                          default_origin = (0,0,0))
copy_stencil(
    src=t_rcvbuf_storage,
    dst=field,
    origin={"src": (0, 0, 0),
    "dst": (num_halo, ny_p+num_halo, 0)},
    domain=(nx_p, num_halo, nz_p),
)

# left
l_rcvbuf_storage =
    gt.storage.from_array(l_rcvbuf,
                          backend = backend,
                          default_origin = (0,0,0))
copy_stencil(
    src=l_rcvbuf_storage,
    dst=field,
    origin={"src": (0, 0, 0),
    "dst": (0, 0, 0)},
    domain=(num_halo, ny_p + 2*num_halo, nz_p),
)

# right
r_rcvbuf_storage =
    gt.storage.from_array(r_rcvbuf,
                          backend = backend,
                          default_origin = (0,0,0))
copy_stencil(
    src=r_rcvbuf_storage,
    dst=field,
    origin={"src": (0, 0, 0),
    "dst": (nx_p+num_halo, 0, 0)},
    domain=(num_halo, ny_p+2*num_halo, nz_p),
)
```

Listing 2: *Assigning halo-updates to `gt4py.storage` containers using the stencil object `copy_stencil`.*

Validation

To ensure that the implementation was done correctly, specifically concerning the halo-updates, the generated codes were validated by comparing the results against the basic serial code used to adapt throughout the block course. The resulting output fields of the MPI-parallelized gt4py leveraging version and the original serial code are depicted in Figure 2. No difference can be detected after a runtime of 1024 timesteps

[1] https://github.com/ofuhrer/HPC4WC/blob/master/day5/solutions/stencil2d-gt4py-v0.py
[2] https://github.com/ofuhrer/HPC4WC/blob/master/day3/partitioner.py
[3] https://github.com/cloudfederer/hpc4wc_project/blob/main/stencil2d-gt4py-a1.py
[4] https://github.com/ofuhrer/HPC4WC/blob/master/day5/solutions/stencil2d-gt4py-v3.py
[5] https://github.com/cloudfederer/hpc4wc_project/blob/main/stencil2d-gt4py-a4.py



(a) *Solution of the original serial code, single process, no MPI, no GT4PY.*

(b) *Solution of the MPI-parallelized performance leveraging GT4PY version, 16 processes.*

Fig. 2: *Comparison of the solution of the original serial code with the solution of the MPI-parallelized performance leveraging GT4PY version after 1024 timesteps for a domain of size 512x512 grid points and a conventional top-hat initial condition.*

and there is no bug in communication of the halo-updates between the different processes visible in Figure 2b. Also, with the `numpy` function `allclose`, the fields were checked to be within a relative and absolute tolarance of 1×10^{-5} and 1×10^{-8}, respectively.

Strong Scaling Analysis

In the following, a strong and a weak scaling analysis were performed. The five available backends, i.e. `numpy`, `gt:cpu_ifirst`, `gt:cpu_kfirst`, `gt:gpu` and `cuda`, as well as both versions of the parallelized GT4PY code were compared. The codes were run for both 64 and 256 iterations on the Piz Daint supercomputer and the figures were created using the outcomes of 256 iterations, as the results appeared less noisy.

In the strong scaling analysis, the time to compute the solution with a varying number of processes for a **fixed problem size** is investigated. Thus, the different problem sizes of 64x64, 128x128, 256x256, 512x512, 1024x1024 and 2048x2048 grid points were examined separately, each for 1, 2, 4, 8, 16, 32 and 64 processes respectively. Strong scaling is usually used for compute-bound tasks in order to trade off and find the optimal runtime versus resource costs. Amdahl's Law [4] states that the speedup S is limited by the fraction of the serial part f_{ser} of the code. It can be read as

$$S_{Amdahl} = \frac{1}{f_{ser} + \frac{1-f_{ser}}{p}} \ , \tag{3}$$

where p is the number of processes. In the limit of $p \longrightarrow \infty$, we can see that the the speedup is given by $S = 1/f_{ser}$. The speed c of a task is defined as the associated work w, i.e. the problem size, divided by the time t needed to complete the work. The speedup S is the factor between the speed for parallel task c_p and the speed for serial task c_1. For the case of strong scaling, denoted by the subscript $_s$, this results in the strong scaling speedup

$$S_s = \frac{c_{p,s}}{c_1} = \frac{w/t_p}{w/t_1} = \frac{t_1}{t_p} \ . \tag{4}$$

Note that the work is the same and therefore cancels. The scaling efficiency E is defined as the speedup S divided by the number of processes p. The strong scaling efficiency gives

$$E_s = \frac{\frac{c_{p,s}}{c_1}}{p} = \frac{\frac{w/t_p}{w/t_1}}{p} = \frac{t_1}{t_p}\frac{1}{p} \ , \tag{5}$$

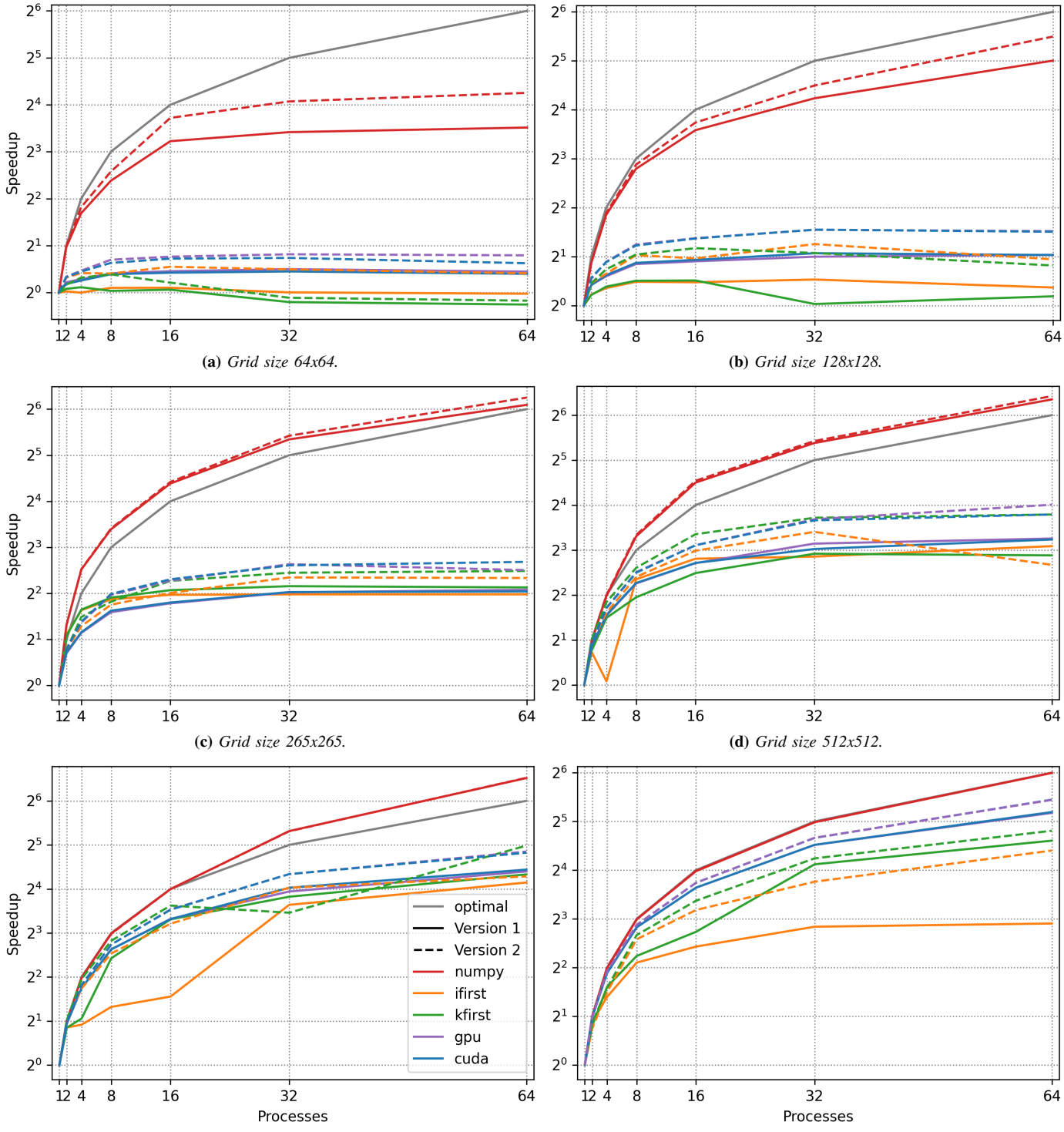
and therefore decreases witch increasing number of processes.

Results of the Different Grid Sizes

The resulting strong scaling speedups according to Equation 4 are depicted in Figure 3. Generally speaking, it can be seen what Amdahls Law (Equation 3) states: The asymptotic behavior for the limit of $p \longrightarrow \infty$ can be noticed depending on the grid size. For example, it starts to be visible at roughly 8 to 16 processes for the 64x64 sized grid (Figure 3a) or at 32 to 64 processes for 512x512 (Figure 3d). With this theoretical behavior, one can infer the serial fraction of the code. In reality, however, it may be that the speedup curve does not asymptotically converge to the value $1/f_{ser}$ but decreases again for higher parallelizations. This can be observed very nicely for the case of size 64x64 and 128x128 in the Figures 3a and 3b for certain backends. The reason is the ratio of the compute-domain versus the halo-domain: The compute-domain on each process decreases when going for larger parallelizations on the same small overall problem, while the halo-domain must remain the same size to fit the stencil. In this case the program starts to be dominated by the halo-updates and is thus increasingly more memory-bound. In addition, parallel overhead starts to become important the more processes are involved. For larger overall problem sizes, like the 1024x1024 and 2048x2048 grids depicted in Figures 3e and 3f, the workload per processor is more reasonable, which results in a much higher Speedup. There the curves are closer to the optimal speedup predicted by Amdahls law (Equation 3), which is obtained for a parallel fraction $f_{par} = 1 - f_{ser}$ of 100%.

Comparison of Both Versions and the Different Backends

- By comparing the runtime for different backends in Figure 4, it is immediately noticed that the `numpy` backend is the slowest among all five backends. For `numpy`, many intermediate arrays need to be created, which is expensive. The performance disadvantage becomes larger as the problem size increases. The backends `gt:cpu_ifirst` and `gt:cpu_kfirst` have the fastest runtime in Figure 4a, followed by `gt:gpu` and `gt:cuda`. However, Figure 4b indicates that the Graphics Processing Unit (GPU) based backends `gt:gpu` and `cuda` begin to gain an advantage for larger problems.
- For the speedup depicted in Figure 3, it is worth mentioning that the `numpy` backend shows a better scaling than the other ones, especially at lower grid sizes (Figures 3a-3c). This smaller overhead can be explained by the fact that communication comprises a relatively small part of the large total runtime that the `numpy` backend has. The remaining four backends eventually catch up for larger



(a) *Grid size 64x64.*

(b) *Grid size 128x128.*

(c) *Grid size 265x265.*

(d) *Grid size 512x512.*

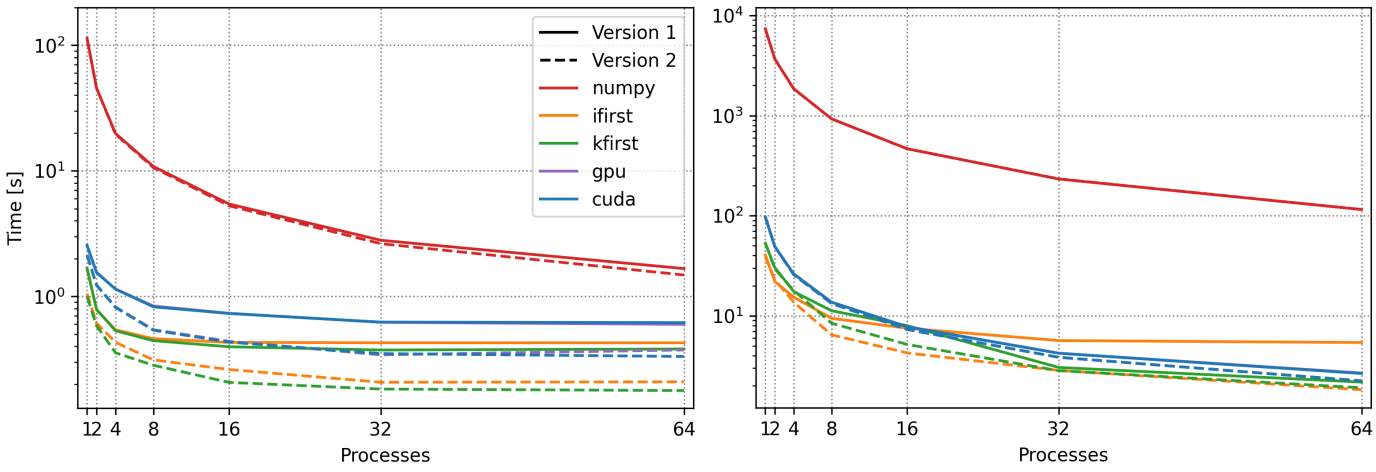
(e) *Grid size 1024x1024.*

(f) *Grid size 2048x2048.*

Fig. 3: *Strong scaling speedup S_s versus number of processes of the two versions of the GT4PY stencil code with different backends on six different Grid sizes. The legend holds for all subfigures. Version 1 (solid lines) is the performance leveraging and version 2 (dashed lines) the simple GT4PY code. All five backends are depicted in color for both versions.*

problems and are almost on an equal level for the grid size of 2048x2048 in Figure 3f. Also, it can be noticed that the backends `gt:gpu` and `cuda` are consistently a bit better than `gt:cpu_ifirst` and `gt:cpu_kfirst`.

- The third thing to notice is that the `numpy` backend is better than the optimal speedup for the Figures 3c-3e. Also the speedup curve lies directly on the optimal curve for the case of the 2048x2048 sized grid in Figure 3f,



(a) *Grid size 256x256.* **(b)** *Grid size 2048x2048.*

Fig. 4: *Strong scaling runtime versus number of processes of the two versions of the GT4PY stencil code for different backends. Two selected grid sizes are depicted. The legend holds for both subfigures. Version 1 (solid lines) is the performance leveraging and version 2 (dashed lines) the simple GT4PY code.*

which means the code would have a parallel fraction of exactly 100%. This behavior of outperforming the theoretical optimum is called super-linear speedup and it rarely happens. One possible explanation for the occurrence of super-linear speedup could be that in parallel programming, not only the number of processes change, but also the size of accumulated caches from different processes. With the larger accumulated cache size, more or even all of the working size can fit into the cache and the memory access time reduces dramatically, which could cause this extra speedup from the actual computation [5]. Further irregularities can be observed for the `gt:cpu_ifirst` backend in Figures 3d and 3e. There the obtained numbers for the speedup for 4 to 16 processes are a bit out of line.

- When comparing both code versions, it is apparent that both have very similar speedup, yet the simple code not leveraging GT4PY (version 2 in Figure 3) is consistently a bit better. By itself, this is not too special but only means the speedup of this version is more scalable by strong scaling. However, it is surprising to see the runtime of both of the versions compared in Figure 4. The basic version not utilizing GT4PY's `copy_stencil` runs faster for all the available backends. Maybe this stencil object starts to get beneficial for much larger problems and/or larger parallelizations. Therefore, it would be interesting to extend the range of processes and/or grid sizes to be inspected towards larger numbers. As can be seen in the next section though, this version of the code is better in weak scaling.

Weak Scaling Analysis

In the weak scaling analysis, the time to compute the solution with a varying number of processes for a **fixed problem size per processor** is investigated. The problem size per process was set to 64x64 grid points, which means that the 128x128 problem was solved on 4 processes, 256x256 on 16 and 512x512 on 64 processes, respectively. Weak scaling is usually used for memory-bound tasks when there is not enough memory available from a single process. Also, in practice we are not interested in using more processes for a small problem, but maybe want to know how big we can make the problem for the given amount of available resources. Gustafson's Law [6] states that there is no limit for the scaled speedup and that it increases linearly with the number of processes. The slope dS/dp is less than 1.0, which would correspond to the optimal case. Gustafson's Law can be read as

$$S_{Gustafson} = f_{ser} + (1 - f_{ser})p \ . \tag{6}$$

For the case of the weak scaling, denoted by the subscript $_w$, the speed for the parallel task $c_{p,w}$ is given by $w_1 p/t_p$, which result in the weak scaling speedup

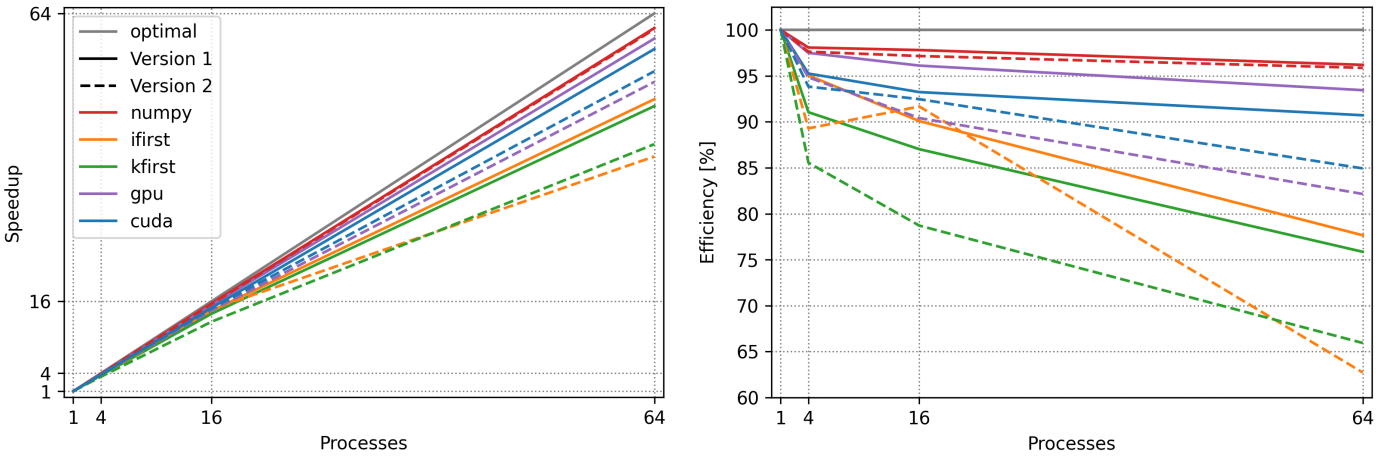
$$S_w = \frac{c_{p,w}}{c_1} = \frac{w_1 p/t_p}{w_1/t_1} = \frac{t_1}{t_p} p \tag{7}$$

and the weak scaling efficiency

$$E_w = \frac{S}{p} = \frac{\frac{c_{p,w}}{c_1}}{p} = \frac{\frac{w_1 p/t_p}{w_1/t_1}}{p} = \frac{t_1}{t_p} \ . \tag{8}$$

Results of Weak Scaling Speedup and Efficiency

The results of the weak scaling analysis are depicted in Figure 5. In Figure 5a the weak scaling speedup according to Equation 7 is shown. We see the approximate linear increase we expect from Gustafson's Law (Equation 6) very well. The slope dS/dp is less than the optimal case of a 100% parallel fraction, i.e. less than 1.0, for all the data. Also we can extrapolate from the picture how the curve continues approximately; it does not seem to have an asymptotic limit of a constant speedup value. In Figure 5b the weak scaling efficiency according to Equation 8 is shown. All the obtained



(a) *Weak scaling Speedup S_w versus number of processes.*

(b) *Weak scaling efficiency E_w versus number of processes.*

Fig. 5: *Weak scaling of the two versions of the GT4PY stencil code for different backends. The legend holds for both subfigures. Version 1 (solid lines) is the performance leveraging and version 2 (dashed lines) the simple GT4PY code. The optimal scaling performance is indicated in gray.*

results for the two versions of the code with the five backends lie underneath the line for optimal parallelization within a reasonable range. Altogether, the results for the weak scaling speedup and efficiency appear to be as expected.

Comparison of Both Versions and the Different Backends

- As already in strong scaling, `numpy` is also the best backend in weak scaling, both in terms of speedup and efficiency. It is again followed by `gt:gpu` and `cuda`, then `gt:cpu_ifirst` and `gt:cpu_kfirst`. Yet again, because of the large runtime of `numpy`, the backend `gt:gpu` is the best candidate for large enough problems, closely followed by `cuda`.
- As already indicated in the last section, the code variant that makes use of GT4PY's built-in storage structures for the halo-updates is the superior choice when comparing the two versions. This applies to both weak scaling speedup and efficiency, as can be found in Figure 5.

Conclusions and Outlook

In this project, we implemented the MPI standard for the halo-updates within the `stencil2d-gt4py` code. A first version uses the `numpy.ndarray` functionality of GT4PY to update the halo points, and a more sophisticated second version leverages GT4PY in updating the halo points by using a stencil object. During the scaling analysis, both Amdahl's and Gustafson's Law could be replicated and explained with the obtained data; In the strong scaling with some irregularities, but in the weak scaling unexceptionally. The basic version not utilizing GT4PY's data storage and management structures when making the halo-updates seems to have better scalability in strong scaling, whereas the version that utilizes those GT4PY modules seems to be superior in weak scaling. Out of all five examined backends, `numpy` has the best speedup and efficiency in both cases of the weak and strong scaling. However it has by far the largest runtime, which makes it infeasible. Out of the other four backends, the choice depends on the problem size and/or level of parallelization. For very large problems and/or number of processes, the GPU-based backends, `gt:gpu` and `cuda`, seem to have an edge over the CPU-based ones, `gt:cpu_ifirst` and `gt:cpu_kfirst`, in terms of runtime, while being more efficient and showing a larger speedup in all regions of problem sizes and degrees of parallelization. Perhaps it would be beneficial to extend the scaling analysis to a larger number of processes than 64, and also to larger grid sizes than studied in this project, in order to see the true power of GT4PY and its different backends.

References

[1] "Gt4py: Gridtools for python, python api to develop performance portable applications for weather and climate." 2022. [Online]. Available: https://github.com/GridTools/gt4py

[2] O. Fuhrer, "Lecture notes of the course high performance computing for weather and climate at eth zürich," 2022. [Online]. Available: https://github.com/ofuhrer/HPC4WC

[3] M. Xue, "High-order monotonic numerical diffusion and smoothing," *Monthly Weather Review*, vol. 128, no. 8, pp. 2853 – 2864, 2000. [Online]. Available: https://journals.ametsoc.org/view/journals/mwre/128/8/1520-0493_2000_128_2853_homnda_2.0.co_2.xml

[4] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, p. 483–485. [Online]. Available: https://doi.org/10.1145/1465482.1465560

[5] I. Tuncer, Gülcat, D. Emerson, and K. Matsuno, *Parallel Computational Fluid Dynamics 2007*. Springer Berlin, Heidelberg, 01 2007. [Online]. Available: http://doi.org/10.1007/978-3-540-92744-0

[6] J. L. Gustafson, "Reevaluating amdahl's law," *Commun. ACM*, vol. 31, no. 5, p. 532–533, may 1988. [Online]. Available: https://doi.org/10.1145/42411.42415