A fast nan-sensitive mean filter function – Report for HPC4WC Course Project, Project Group 9

Verena Bessenbacher, Ulrike Proske

August 7, 2020

1 Motivation

In Weather and Climate applications, one is often confronted with observations containing missing values. Many analyses, however, depend on the observations to be gapfree. For example, satellite remote sensing data of the land surface has missing values due to a variety of factors, including satellite overpass geometry, dense vegetation, cloud or ice cover and other obstacles. When combining different satellite datasets, only taking the spatial or temporal points where all variables are observed, leads however to large data losses. Therefore, gapfilling is necessary. One approach to gapfilling is the interpolation of a univariate field in space and time. A very simple form of spatiotemporal interpolation is taking the mean of the surrounding spatial and temporal points, if imagining a dataset as a three-dimensional cube (latitude, longitude, time).

2 Dataset Description

We are confronted with a global, 10-year daily satellite remote sensing dataset of 2-meter temperature, precipitation and surface layer soil moisture in 0.25 degree resolution. Each of the variables has a different pattern of missing values, and the pattern is dependent on time, latitude, longitude and several other factors. We would like to fill all gaps with a spatiotemporal interpolation of all their neighbors that are two pixel in each direction. More specifically, imagine the pixel to gapfill in the middle of a three dimensional cube with a sidelength of 5 pixels. The value with which to fill the gap is the nanmean of all the points in the cube. With nanmean, we mean a nan-ignoring mean as for example `numpy.nanmean` is (where nanmean(x,y,nan) = mean(x,y); nanmean(nan,nan,nan) = nan).

Our dataset is four dimensional. The four dimensions and coordinates are:

- 3 variables: 2-meter temperature, precipitation, surface layer soil moisture
- 3653 timesteps, corresponding to daily resolution in the 10 year window 2002-2012
- 1440 latitudinal points and
- 720 longitudinal points, which corresponds to global coverage at a 0.25 degree resolution

We acknowledge that this is a large dataset, taking up around 8.5 GB of diskspace. Next to the filter algorithm, another important bottleneck is loading this data into RAM. Since we are lucky enough to have a large enough RAM at our disposal to load the whole dataset, we exclude for this project the time it takes to load it into RAM and focus on speeding up the filtering function only.

3 Problem Definition

Practically, performing a gapfilling procedure as described above can be done with convoluting filters that are also common in image processing. The `scipy.ndimage` library in python is often used for such purposes. However, since the gapfilling filter needs to be nan-aware, no off-the-shelf filter is available. The gapfilling function needs to be passed to the `scipy.ndimage.generic_filter` function.

This function is already written using `cython` to enhance performance, however, on our dataset it still takes several days to finish (see Table 1).

The goal of our project is to rewrite the filter function using tools and concepts introduced in the HPC4WC course. In particular, by viewing the convolution filter as a stencil function, we can employ a larger toolbox available for fast stencil computation in python, C and FORTRAN. We would like to obtain a speedup close to only 1-digit times slower than the theoretical limit (see Section 4).

4 Setting the Benchmark

In python, we use the standard `numpy.float32` data type, using 32 bit per datapoint. The dataset therefore consumes

$$4\,\mathrm{B} \cdot 3 \cdot 3653 \cdot 720 \cdot 1440 \approx 45\,\mathrm{GB} \tag{1}$$

The difference between this data size and the size of 8.5 GB mentioned in Section 2 is that in the file on disk, all ocean points are omitted for efficient saving. However, those points are needed in the dataset to retrieve the spatiotemporal relationships again. Once the dataset is loaded into RAM, calculating the spatiotemporal mean for each datapoint should in the theoretical minimum be only one *read* and one *write* process per datapoint. If we assume a theoretical bandwidth of 40 GB/s, which is roughly the one of daint, we should be able to perform the calculation in

$$\frac{45\,\mathrm{GB}}{40\,\mathrm{GB\,s^{-1}}} \cdot 2[\mathrm{READ/WRITE}] \approx 2\,\mathrm{s} \tag{2}$$

seconds. However, hardware architecture also needs to be considered. For this project, we decide to work on the IAC internal *ch4* server, since it is able to accommodate the whole dataset in RAM, having 1500 GB RAM. *ch4* has 64K L1 Cache, 1024K L2 Cache and 25344K L3 Cache. Its bandwidth and therefore the theoretical minimum time needed for the calculations is simliar to the one on daint[1]. However, the 48 CPUs of *ch4* are shared. If we are not parallelizing and only using one CPU, we will not be able to utilize the full bandwidth, which represents only a theoretical peak limit.

As a goal for this project, we are satisfied once we are in the one-digit times slower than the theoretical benchmark.

5 Software Development

Over the course of this project, we developed multiple implementations of the nan-sensitive mean filter function. Starting out with the naive implementation in *baseline_explicit.py* and with the ready-made `generic_filter`, the other versions in Section 5.2 aim to address particular problems of these base

[1] Processor: Intel(R) Xeon(R) Gold 6246 CPU @ 3.30GHz, 3300 MHz

versions in order to speed up the program. Where not specified otherwise, the programs load the dataset described in Section 2, and then filter the data in the ways described below. This filtering is timed. All scripts can be found at https://github.com/ofuhrer/HPC4WC/tree/master/projects2020/group09.

5.1 Naive implementations

As a first approach, we naively implement the filter function (1) explicitly using python loops and (2) applying `np.nanmean` to the `scipy.ndimage.generic_filter` function. This serves as a baseline for later improvement steps.

5.1.1 baseline_explicit.py

In this implementation, the python program loops over all data points, copies all data in the surrounding 5x5x5 box, loops through this data and sums up the individual elements. At the same time, it checks whether each element in the box is nan, and if it is not, increases a counter. In the end, the result is the sum of the elements divided by the counter, which is equal to the result of `np.nanmean`. This result dataset is then used to fill the nans in the original dataset.

The implementation can be found in *baseline_explicit.py*. We have no explicit timings for this version, since it took several days to finish on the large dataset and was eventually aborted.
In the following we refer to the script *compare_versions.py*, except for the FORTRAN code, which is in separate files.

5.1.2 generic_filter

Another naive implementation is to use the ready-made function
`scipy.ndimage.generic_filter(data, np.nanmean, footprint=np.ones((1,5,5,5)), ...)`

Both of the naive implementations are slow compared to the benchmark (see Table 1). In the following, possible reasons as well as resulting speed up strategies are described:

1. The filtered data is used to fill gaps (nans) in the original data. Therefore, a check whether the original data is nan while looping through all data points as in *baseline_explicit.py*, could avoid the computation of unnecessary filtered data at points where it will not be used. This is implemented in the cython programs. However, as such a check is only possible where the loop through data points is explicit (e.g. not in *generic_filter*), we did not always implement the check to keep comparability between programs.

2. Python is a high-level language. Its functions get recompiled every time they are used. In the case of the `generic_filter`-function, this means that `np.nanmean` gets recompiled for every data point. This can be avoided by **precompiling** parts of the code (using `numba` or `cython`, as described in Section 5.2.1 and 5.2.2) or by using a **compiled language** like FORTRAN (see Section 5.2.3).

3. Reads and writes from RAM into Cache and back are expensive. As stated above, the filter function needs to read and write every data point at least once. Remember that the L3 Cache on *ch4* is $25\,344\,\mathrm{kB}$ in size. This means that $(\frac{25344 \cdot 1024\mathrm{B}}{4\,\mathrm{B/float32}}) = 6\,488\,064\,\mathrm{float32}$ can be stored in L3 Cache. This is less than the number of floats stored in our dataset. Therefore, when a data point is needed

another time, it might not be in Cache anymore and needs to be read out of memory again. A smarter way to manage Cache storage is to use **Blocking**. With Blocking, we divide the dataset into cubical subsets that entirely fit into Cache. The advantage is that within one of those cubes, almost all values that need to be accessed are already in Cache, as compared to normal loop execution, where we go along one axis of the dataset entirely, making it necessary to load lots of datapoints into Cache that are deleted before they are used again. Without Blocking, in *baseline_explicit.py*, likely every data point needs to be read roughly on the order of 25 times. We do not know whether `scipy.ndimage.generic_filter` already optimizes for this. As our program is only concerned with one data variable at a time, it is mostly working with 3-dimensional cubes (time, lat, lon) of data. The maximal side length of such a cube that fits completely into the L3 Cache is $6\,488\,064\,\mathrm{float32}^{\frac{1}{3}} = 186\,\mathrm{float32}$. Blocking is implemented in both FORTRAN versions (see Section 5.2.3) and in `cython_loop` (see Section 5.2.2).

4. Data copies are expensive. In *baseline_explicity.py*, the data from the surrounding box is copied into *values* explicitly. This is more expensive than **only using pointers**. This problem is addressed in the version using *numba.stencil* (see Section 5.2.1).

5. There are several ways to implement code that effectively produces a nan-sensitive mean of the provided values. The first way is to use off-the-shelf available functions as `np.nanmean`. In a second way, as already outlined in the introduction, we can view the nan-sensitive mean as a stencil function, utilising fast stencil implementations available in python. The third option can be a simple loop architecture that loops through the provided values, sums them up while ignoring nans and counts the number of non-nan values. Finally it divides the sum by the count. It is non-trivial to see which of those implementations is the fastest one, as shown below.

6. In both naive implementations, every data point is addressed sequentially. On *ch4*, we have the option to use multiple CPUs at once, which allows us to parallelise the process. The simplest **parallelisation** is the one over the three variables, as the corresponding data points in time and space can be addressed independently of the other variables. A parrallelisation over variables is implemented in *compare_versions.py*, see Section 5.2.5.

5.2 Advanced implementations

Here, we describe the scripts in which the solutions mentioned above are implemented, and evaluate their performance. Their runtimes are summarized in Table 1. Since *ch4* is a shared server, accurate timing proved difficult. We addressed this problem by running and timing multiple times and taking the mean in Table 1.

5.2.1 Numba

`Numba` is a python package that can help us to circumvent the fact that python compiles every function at runtime, which means that `np.nanmean` is compiled for each pixel where it is been applied. The decorator `numba.njit` is used for the purpose of compiling the function only once and afterwards using the compiled function for each function call.

`Numba` also offers a stencil decorator for efficiently applying stencils to large datasets. We can replace the `np.nanmean` with an appropriate stencil function. We have found that the fastest stencil solution uses a stencil that sums over the (5,5,5) cube. This sum function is used once over the values, where it

Table 1: Program timings, array-size: 3x3653x1440x720, mean of three runs (two for cython)

Solution	Time
baseline_explicit.py	several days
generic filter with np.nanmean	$>$ 16h
numba.njit	2h 4min 12s
numba.stencil	1h 28min 14s
numba.njit + numba.stencil	51min 52s)
cython with stencil	04min 37s
cython with stencil, blocked	12min 44s
cython with loop	23min 47s
cython with loop, blocked	20min 02s
cython with stencil, with variable parallelisation	02min 46s
baseline_fortran.f90, not blocked	2h 34min 12s
baseline_fortran.f90	2h 34min 44s
fortran_nocopy.f90, not blocked	31mins 45s
fortran_nocopy.f90, 100x100x100 blocked	30mins 43s

Table 2: Fortran blocking timings, smaller array-size: 3x365x1440x720.

Block size	Time (s), mean of three runs	
	baseline_fortran.f90	*fortran_nocopy.f90*
No blocking	853	146
100x100x100	872	181
150x150x150	886	187
180x180x180	876	174
200x200x200	879	175

treats every nan as zero and once over a mask array, which indicates nans with a zero (otherwise one). We then divide the first result by the second to obtain the nan-sensitive mean. Other solutions, including a stencil getting both the mask and the values array and effectively computing the nanmean in one stencil call have shown to be slower (not shown).

With `numba`, we can only speed up by saving the compilation time. Other issues of the slow code are not adressed. However, the `scipy.ndimage.generic_filter` function is already `cython` optimised under the hood, so we do not know whether some of the speed up we see takes place because of blocking or very efficient read and write statements.

A third option with `numba` is to use both decorators to speed up the process. We found such a combination of the `stencil` and the `njit` decorator to be the fastest, and the pure `stencil` decorator the slowest. We assume this is because the stencil implementation additionally needs to run the stencil on the mask array, and needs to be executed consecutively for all three variables, making it six executions of the stencil in total. This might not be an optimal implementation, however, we did not find any faster stencil implementation that would correctly produce the nanmean results. Combining both decorators is fast since the stencil function is precompiled with `njit`, giving additional speed-up.

5.2.2 Cython

We also translated the filter function into C with `cython`. This is done to eliminate the compilation time and, in a second step, we are also able to introduce blocking into the function, trying to maximise usage of the data loaded into Cache. We tried both writing the nanmean as a stencil and a simple loop structure in C and compared them with each other.

`Cython` allows running part of the code completely in C, and with the `nogil` statement we ensure that the C function does not jump back into the python interpreter. The functions with `cython` are overall the fastest throughout this evaluation. As compared to `numba`, `cython` allows not only to skip the compilation step for every function call, but runs entirely in C. In both implementations, the stencil and the loop nanmean function, we check before computation whether the value at hand is nan, and only compute the nanmean if so. This gives a strong speedup, since points and computations can be skipped.

The loop is slower than the stencil. We hypothesise that this could be because of two reasons: First, for the stencil function, we need to create a mask array and replace all nans with zeros in the values array. These are additionally two read and writes, but they happen outside the timings for consistency with the numba solution. Secondly, the loop function contains a lot of if-statements, which prevent loop unrolling from the compiler. Additionally, the loop function does not validate with the original results. We currently do not know why this is the case, since the results look reasonable at first glance.

Additionally, we tried both implementations of the function within a Blocking structure. For the stencil function, the blocking sizes are adjusted to a smaller cube size, since two arrays (mask and values) need to be in Cache at the same time. We calculated from the known size of the Cache that a cube of the side length $\left(\frac{6\,488\,064\,\text{float32}}{2\,\text{arrays}}\right)^{1/3} > 148\,\text{float32}$ still fits into Cache. However, the blocking code does not run faster than the not blocked code. For a discussion on possible problems, see Section 5.3.

5.2.3 FORTRAN

In order to test the performance of a precompiled language, we also wrote a FORTRAN script for the filter function (*baseline_fortran.f90*). Since the script is to be used for testing only, we chose to construct the input field within the script to keep it simple, rather than read in the field. The input field is filled with random numbers between 0 and 1. In order to stay consistent with the `numba` implementation, we also chose to circumvent the treatment of nans with a mask. In the Fortran implementation, this mask is arbitrarily created with the condition of an input value smaller or equal 0.5 giving a mask value of 1. Of course, the two arrays, one for the values and one for the mask, need more disk and Cache space, but this was the only way we saw to imitate the handling of nans as necessary for the cython implementation.

The program loops over all variables, calls two functions that loop over the other dimensions and call a stencil for the computation of the weights (sum of non-nan value appearances in the surrounding box) and the result values. The stencils are written out explicitly. For the loops over the dimension, there is the option to turn on blocking with the `lblocking` switch, and to set the block sizes explicitly with `blockx`, `blocky`, `blockz` in the beginning of the program (strategy 2). The results of this basic FORTRAN script can be validated with *baseline_fortran_check.py*. Interestingly, the basic version validates only with a set absolute tolerance of 0.05.

The performance of this basic FORTRAN version is surprisingly slow considering that it is a compiled language. We would have expected FORTRAN to be at least as fast as `cython`, but it is only about as fast as `numba`.

Again, we identified several reasons for the slow performance, which a second version of the fortran script, *fortran_nocopy.f90*, improves upon:

1. Data copies are expensive and unnecessary (same as 4. in Section 5.1.2). In *fortran_nocopy.f90*, we avoid the creation of copies by passing the whole array (e.g. `mask(:,:,:,:)`) to the subroutine, as well as the indices `i`, `j`, `k`. The array can then be indexed inside the subroutine, and the creation of unnecessary copies is avoided.

2. Subroutine calls are expensive. In *baseline_fortran.f90*, the subroutines `weights_stencil` and `nanmean_stencil` get called once for every data point. This was avoided in *fortran_nocopy.f90* by inlining the content of the subroutines.

3. Reads and writes from RAM into Cache are expensive. The computation of weights and results was separated into two different subroutines in *baseline_fortran.f90*. In both subroutines, the mask array had to be read at least once. One of these two reads was saved in *fortran_nocopy.f90* by combining the weights and values computations in the inner most loop within the same subroutine.

4. An additional speed up could be achieved by using fast compiler options (`-O3 -ftree-vectorize -funroll-loops`).

5. The compiler can do the unrolling of small and simple loops by itself. This allowed to write the stencils themselves as loops over the 5x5x5 box in *fortran_nocopy.f90*, and not as explicit sums. This improves the readability of the program.

fortran_nocopy.f90 achieves a speed up of more than 5 compared to *baseline_fortran.f90* (see Table 1). The script itself is also much shorter and simpler to look at. In addition, probably due to its simplicity and avoided copying of data, it validates more precisely against `numpy` (it validates even without a specified absolute tolerance).

5.2.4 Halo updates

For most of the development, we focused on the filter function itself, ignoring the points at the boundaries. However, for a completely filled dataset, these halo points need to be treated. Specifically at the longitudinal boundaries, the boundary points should be computed using points from the other end of the array, since these points will be neighbors on a sphere. For upper and lower boundaries, no such solution exists since at the pole it is not clear which points will be the neighbouring ones on a sphere. Therefore we chose not to extend the box for the stencil at the northern and southern boundaries but rather to use a smaller box for these boundary points. The box ends at the boundary, so that it is only 3x5x5 or 4x5x5 in size (for the outermost and outermost-1 points). Similarly, for time, an extension of the stencil box e.g. from the past to the future at the time boundaries does not make sense. Consequently, also for the time boundaries, smaller boxes were used.
A separate function for the halo updates can be found in *halo_bc_update.py*. It loops over all halo points. For each edge or corner point, the corresponding box indices were specified manually. The function `numba_nanmean` is then called for each box. This program validates against the `generic_filter` function with `mode='wrap'` at the longitudinal boundaries. However, it is quite slow. The possible reasons are similar to the ones mentioned before, and could be adressed in the future:

- Data copies are expensive (same as 4. in Section 5.1.2 and 1. in Section 5.2.3).
- Subroutine calls are expensive. `numba_nanmean` gets called for every point in the boundaries. A speed up could possibly be achieved by inlining the content of the function.
- Reads and writes are expensive. Since the edges are addressed consecutively, the loops over the edges are somehow naturally blocked. As arrays of size 8x1440x720 still do not fit into the L3 Cache, the outer loops in *halo_update* force more than one read of every addressed data point (even without created copies as pointed out above). Therefore, also in this function more extensive blocking could be beneficial (see 3. in Section 5.1.2).

5.2.5 Parallel var

Since no interpolation happens between the first dimension of the dataset, the variables, it is straight forward to parallelise their processing over 3 CPUs to obtain a performance boost. We use `concurrent.futures` for this purpose.
We used the fastest implementation thus far, i.e. *cython with stencil, blocked*. Parallelisation over the variables leads to an approximate speed-up of one third of the original code. This is expected since the calculation over the variables are are completely independent from each other.

5.3 Blocking

Contrary to what we expected, the tests with cython and fortran show no speed up in the runs that use blocking, independently of dataset and block size (see Tables 1 and 2). There are several possible reasons for this:

- We introduced a safeguard for block sizes that do not fit exactly into the dataset size: we are looping over more than the strictly necessary points. However, there is an if-statement checking whether the point is within the dataset. If not, no computations or reads are executed, so the additional time consumed should be small.

- Blocking over 3 dimensions is slower than blocking in 1 dimensions, because the values in a block are not next to each other in memory.
- In the versions that use a sum-stencil for the computation of the nanmean, two arrays are coexisting in Cache at the same time: the mask array (indicating whether a value is nanor not) and the values array (with the original values of the array). Therefore, blocking sizes need to be small such that the blocks of both arrays can fit into Cache for efficient blocking.
- The compiler possibly executes some sort of blocking by itself, which could be more efficient than the one we force to be executed manually.

6 Conclusion

In this project, we implemented different versions of a nan-sensitive mean filter function. By using techniques such as precompiling, parallelisation, and pointer passing, we were able to vastly improve the function's performance. In fact, the fastest version is the `cython` version that uses a stencil, without blocking.
On the other hand, our attempts at the implementation of blocking did not yield a speed up in performance. Ultimately, we are still two orders of magnitude slower than the theoretical benchmark. Thus, we learned that writing performance optimal code is difficult, even for a problem that presents itself as simple as the filter function. During the course of the project we also learned the following take home messages:

- Analyse what the biggest performance limiter is first, then start to optimize your code.

  > Premature optimization is the root of all evil.
  > *– Computer Programming as an Art, Knuth's Turing Award lecture (1974), printed in Communications of the ACM, Volume 17, Issue 12, Dec. 1974 (see p.671)*

- For optimal code, we need to understand what the compiler is doing (by itself), and what is hindering it to compile optimally (see e.g. point 1, 4, and 5 in Section 5.2.3).
- "Clean" code is not equal to fast code (see e.g. the split up into numerous subroutines in baseline_fortran.f90 versus the inlining in fortran_nocopy.f90).

Environment description

A list of the packages in the environment used within this project can be found at `HPC4WC/projects2020/group09/environment.yml`.