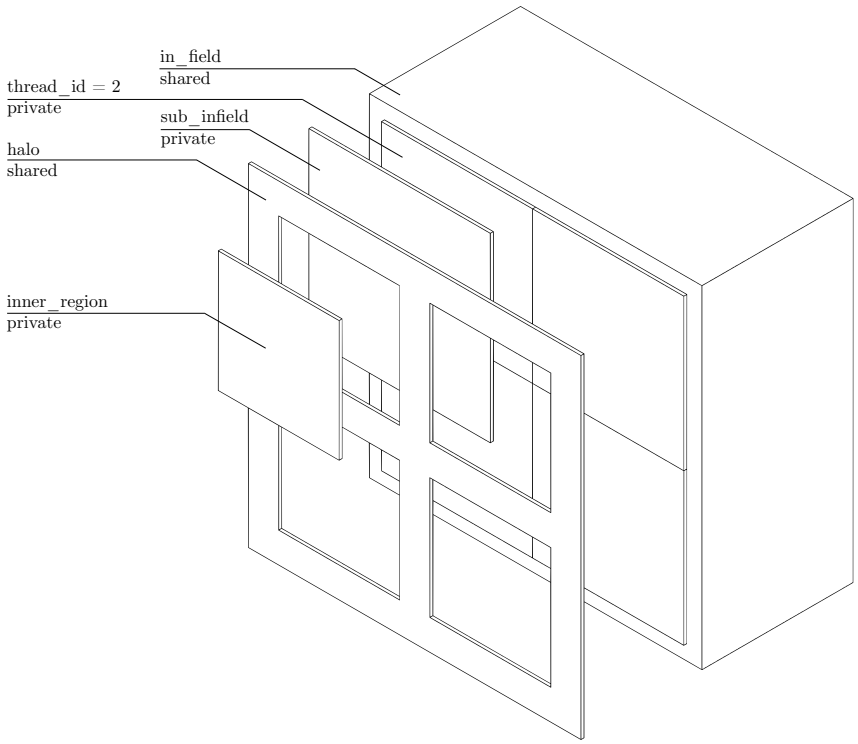
High Performance Computing for
Weather and Climate

Using OpenMP to implement distributed memory



Zoé Jequier, Filippo Ferrazzini

Supervisor: *Tobias Wicky*

ETH Zürich

August 2023

Abstract

With the recent need in research to run models on ever finer scales, various solutions have loomed, most of which related to parallel computation, like OpenMP and MPI. In this report, we inspect the possibility of using the shared-memory parallel paradigm of OpenMP to run code in distributed-memory platforms, which are usually implemented with MPI. We investigated the feasibility and performance of this implementation by applying a 4th-order diffusion model on a 3D grid. Our simulations show results that are better than the ones obtained with MPI and comparable to the ones in standard OpenMP implementations, showing the potential of such implementations on relatively small domains.

Contents

1 Introduction **1**

2 Methods **2**
2.1 Stencil 2D computation 2
2.2 Implementation with OpenMP 3
2.2.1 Allocating the fields 4
2.2.2 Performing the stencil computation 4
2.2.3 Storing the values 4
2.3 Evaluation of performance 4
2.3.1 Scaling 4
2.3.2 Comparison 5

3 Results **5**
3.1 Strong scaling 6
3.2 Weak scaling 7

4 Conclusions and outlooks **7**

A Pseudo-code of apply_diffusion subroutine **9**

1 Introduction

Weather and climate models need to solve at each timestep differential equations over all gridpoints of the domain. The number of operations is high, and increasing the resolution of the horizontal grid by a factor of x increases the computational demand by its third power, x^3. Therefore, in order to run finer gridded atmosphere or climate models, optimization techniques such as parallelisms are needed in order to get results in a reasonable runtime. Various attempts have been considered over the time, with some of the most prominent examples being MPI (Message-Passing Interface) and OpenMP (Open Multi-Processing).

MPI has first been introduced in 1994, and is by far the most popular portable communication library in the paradigm of parallel computing on a distributed address space, and it owes part of its success to its widespread use on clusters and highly-parallel systems. A typical implementation requires a decomposition of the grid into multiple subdomains (e.g. using a partitioner), with each worker (computing unit, or node) performing calculation on the assigned region. To communicate the values across nodes, the MPI interface is used.[1][2]

OpenMP, on the other hand, is a portable standard for the programming of shared memory systems that was originally designed in 1997. OpenMP offers an easier paradigm of parallel computer than MPI thanks to its collection of compiler directives, library routines, and environmental variables.[1][2][3]

As stated, each of these protocols has its own advantages and drawbacks. However, one could go outside the scope and try to extend the ease of OpenMP to a distributed-memory platform as well. This is precisely the goal of this report, where we investigate an implementation of OpenMP in which each thread of a node is assigned to a subdomain and the communication of the values is done through a shared array. We then evaluate our results by running a bi-dimensional stencil computation to compute the diffusion of a variable. This operation is widely used in weather and climate models to control small scale noise that is intrinsic to these models. [4]

2 Methods

2.1 Stencil 2D computation

Stencil computation are at the basis of many algorithms used to numerically solve partial differential equations (PDE). This method relies on grid cells using information stored on neighbouring grid points in order to compute the value of a variable. The simplicity of this method, together with the intrinsic need to solve PDE, is the reason why such algorithms are the classical way of resolving a prognostic variable in weather models.

This report focuses on a 4th-order monotonic diffusion (Equation 1) which aims to control small scale noise in models using a non-dimensional diffusion coefficient α_4:

$$\frac{\partial \Phi}{\partial t} = -\alpha_4 \nabla_h^4 \Phi = -\alpha_4 \Delta_h (\Delta_h \Phi), \tag{1}$$

where ∇_h and $\Delta_h = \nabla_h^2$ designate the nabla and laplacian operator, respectively.

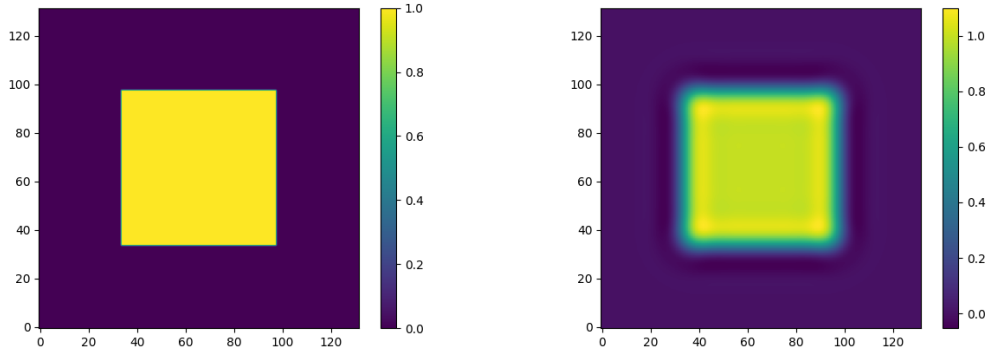


Figure 1: Example of input field (left) and resulting field (right) after applying 1024 steps of the 4-th order monotonic diffusion. The plots were obtained from the OpenMP implementation, based on domain decomposition.

Both sides of the equation have been discretized using the finite difference method. The temporal derivative was computed with a 1st-order forward (Euler) method

$$\frac{\partial \Phi}{\partial t} = \frac{\Phi_{i,j}^{n+1} - \Phi_{i,j}^{n}}{\Delta t}, \tag{2}$$

whereas the spatial discretization was done using a 2nd-order centered method:

$$\Delta_h \Phi_{i,j}^{n} = \frac{-4\Phi_{i,j}^{n} + \Phi_{i-1,j}^{n} + \Phi_{i+1,j}^{n} + \Phi_{i,j-1}^{n} + \Phi_{i,j+1}^{n}}{\Delta x^2}. \tag{3}$$

The numerical implementation of the diffusion equation, $\Delta_h(\Delta_h\Phi)$, is calculated in two steps, by saving the intermediate value in temporary array.

```
tmp1_field(i, j) = -4._wp * sub_infield(i, j, k)                    &
            + sub_infield(i - 1, j, k) + sub_infield(i + 1, j, k)  &
            + sub_infield(i, j - 1, k) + sub_infield(i, j + 1, k)

laplap = -4._wp * tmp1_field(i, j)                                  &
            + tmp1_field(i - 1, j) + tmp1_field(i + 1, j)          &
            + tmp1_field(i, j - 1) + tmp1_field(i, j + 1)

sub_infield(i, j, k) = sub_infield(i, j, k) - alpha * laplap
```

The external halo is managed according to periodic boundary conditions at the beginning of every timestep, and the scalar α (`alpha`) already includes the terms that appear due to discretization

$$\texttt{alpha} = \alpha_4 \frac{\Delta t}{\Delta x^2}$$

2.2 Implementation with OpenMP

To use OpenMP on a distributed memory system we need to overcome some technicalities that are usually not present in systems with MPI, like accessing values that are needed by multiple threads at the same time. For a more detailed information, see Appendix A or the extra material for the entire Fortran code.

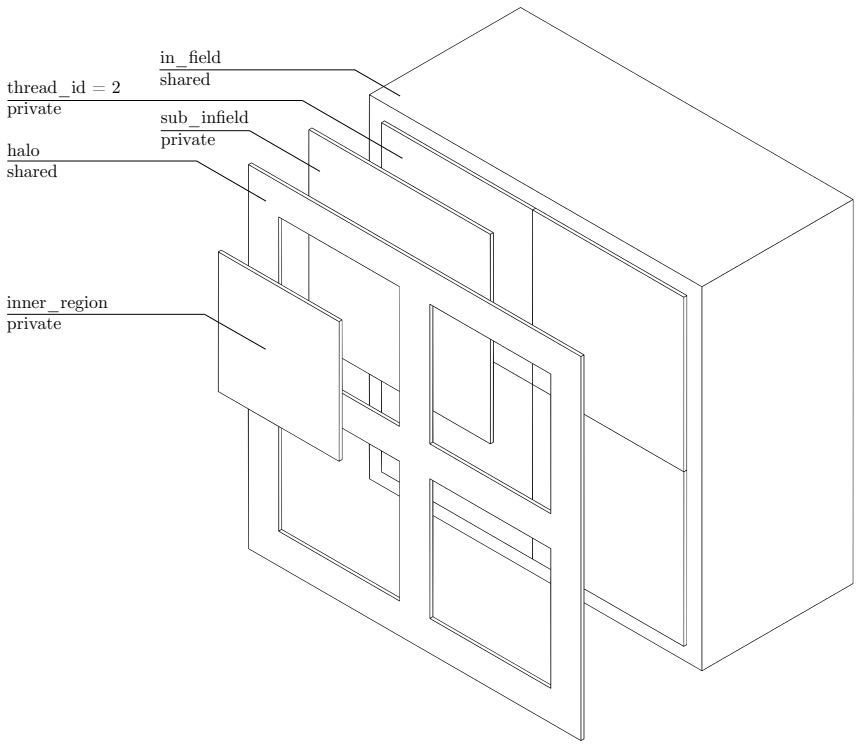


Figure 2: Schematics of the OpenMP implementation. The domain of the `in_field` is divided into subdomains of equal sizes. Thread number three (`thread_id=2` as numbering starts with 0) is first column, second row. The `sub_infield` region includes two grid points on each side as halo. The results are then written either to `halo` or to `inner_region`, depending if the value needs to be shared with other threads or not.

2.2.1 Allocating the fields

The first step is the division of `in_field` into subdomains. For the sake of simplicity, we only divided the region into squared numbers (i.e. 1, 4, 9, 16, ...) and assumed that the size of the whole domain is divisible by the square root of the number of threads used (i.e. 1, 2, 3, 4, ...). This way the region is divided into squares of the same size along the xy-plane (external halo excluded, see Figure 2), and each thread receives two additional cell points of halo per side.

2.2.2 Performing the stencil computation

Each threads can then compute the stencil computation on its entire subdomain according to Equation 3, with the intermediate steps stored privately.

2.2.3 Storing the values

The computed value is then stored either on the private `sub_infield`, or in the shared `halo` array if the values need to be accessed by other threads. The shared region is twice as wide as the halo, i.e. it includes the external halo and the outer part of the inner subdomain (used as halo points by neighbour threads). The location to which the values are stored in `halo` is defined by a mapping function: the indices are shifted across the xy-plane depending on the value of `thread_id`. Assuming `sub_nx` (`sub_ny`) to be the dimension of the subdomain along the x-axis (y-axis respectively), the indices of the threads are then adjusted as following:

```
1 n = sqrt( n_threads )
2
3 row = floor ( thread_id / n )
4 col = modulo( thread_id, n )
5
6 i = i + row * sub_nx
7 j = j + col * sub_ny
```

Example: Assume an `in_field` of size $nx \times ny \times nz = 120 \times 120 \times 64$, plus 2 additional halo points per side (i.e. total array size is $124 \times 124 \times 64$), and `n_threads = 4`, so that in the code presented above `n = 2`. The `sub_infield` will be of size $60 \times 60 \times 64$, plus two additional halo points per side. The region of the `sub_infield` for which the values need to be stored in a shared array consists therefore of the outer 4 cells per each side, and the values in the innermost part of the subdomain (size $56 \times 56 \times 64$) are kept private throughout the entire runtime.

As stated above, each thread has access to data that allow it to calculate values solely within its own subdomain, so there is no race condition occurring within a timestep. However, if a thread was to work faster than its neighbour, neighbour thread could risk reading values of different timesteps when copying the halo. For this reason a barrier is put at the end of each timestep to allow all threads to finish their work and copy the new values to their private arrays.

2.3 Evaluation of performance

2.3.1 Scaling

To evaluate the quality of the results, one calculate how well the problem scales according to Amdahl's law (strong scaling), that defines the speedup $S(s,p)$ (Equation 4) of a parallelized code compared to the time required to run the code with only one thread.

$$S(s,p) = \frac{T(s,1)}{T(s,p)} = \frac{1}{(1-p) + \frac{p}{s}} \tag{4}$$

with s being the number of workers and p the parallel fraction of the code.

The strong scaling is done by running the program while keeping the problem size constant (once at $120 \times 120 \times 64$, once at $180 \times 180 \times 64$) with 1024 timesteps and changing the number of threads, with `num_threads` $\in \{1, 4, 9, 16, 25\}$.

The weak scaling has been evaluated according to Gustafson's law, which describes the workload $W(s, p)$ (Equation 5) that can be done using multiple workers within the same time as if the code was sequential:

$$W(s, p) = (1 - p)w + spw \tag{5}$$

where w is the workload per processor, p the parallel fraction and s the number of workers.

The weak scaling is done by allocating a size of $60 \times 60 \times 64$ with 1024 timesteps for each thread and gradually increasing the number of threads.

Using weak scaling, one can also investigate the parallel efficiency (defined in Equation 6). As the problem size per worker is constant, the additional time that is needed for the code to run in parallel, with multiple workers gives an estimation of the overhead:

$$E(p) = \frac{T(1)}{T(p)}, \quad \text{with } T(p) \text{ the runtime using } p \text{ workers.} \tag{6}$$

A parallel efficiency of 100% would mean that each worker solves the problem as fast as one thread working independently, even when working in parallel. In real case scenarios this is hardly possible since the parallelization is intrinsically linked to a overhead (e.g. communication, synchronization, ...).

2.3.2 Comparison

The performance of the implementation of the diffusion equation using openMP has been compared to an analogue implementation in MPI and a simple OpenMP k-blocking (also called k-parallel) implementation. The running time and respective speedup compared to a baseline has been investigated for each implementation through strong scaling. The baseline chosen in this case, is a sequential code (no parallel region) that applies the 4th-order diffusion equation either on a domain with `nx = 120` or `nx = 180`.

MPI The MPI counterpart of the implementation is structured in an analogous way. The `in_field` is divided into subdomains of equal sizes, with the only difference that the data is not partially stored in a shared array. Instead, the values are communicated among ranks using the messaging protocol.

OpenMP k-parallel The OpenMP alternative version is based on a more intuitive way. As the stencil computation is not depending on other vertical layers, each thread gets to work on the entire layer, simply according to a `!$omp parallel do` statement on the k-loop. Note that to evaluate the weak scaling of this implementation and keep the problem size per worker constant, an additional version has been implemented where the horizontal domain size has been kept constant at 240×240, and 4 vertical layers have been added for each thread.

Each experiment has been carried out 10 times and the average runtime has been taken to ensure the stability of the results.

3 Results

The results of the strong and weak scaling of the different implementation of the fourth order diffusion on a 3D field are presented in this section.

3.1 Strong scaling

Starting from the baseline values of 0.74 s for `nx=120` and 1.69 s for `nx=180`, the tests show promising results, with improvements linked to almost all runs where at least four workers where active. The numerical values of the runtime and the speedup are summarized in Table 1 and Figure 3. As expected, with our implementation of OpenMP, the execution time decreased as the number of threads increased. The only exception to this trend can be found in the run where 25 threads were used, thus exceeding the actual available number of threads and therefore generating additional overhead without benefits in performance.

The trend holds similarly for both the MPI and the k-parallel implementations, although with some interesting twists. On the MPI side, the results are not as impressive, at least for `nx = 180`, with a runtime plateauing just below the baseline. If the domain size is set to `nx = 120`, however, the speedup increases with a peak at nine workers, and diminishes if more workers are used.

In all cases, the runtime is systematically bigger when `nx=180`, as we would expect because the work load by thread is bigger. In contrast, the speedup increases while increasing the domain size for both OMP versions (k-parallel and the shared-memory version), showing that the bigger the domain size, the bigger the benefits from the parallelization. However, in the case of MPI implementation, there is no speedup compared to the baseline when increasing from `nx = 120` to `nx = 180`. This is probably due to the communication overhead that is higher in the MPI case than in the other implementations.

The best results are achieved with the k-parallel version, which is faster both on both domain sizes. The k-parallel speedup is also by far the greatest, with a reduction in computation time of up to 3 or 4 times for `nx = 120` and `nx = 180`, respectively..

		Runtime per number of workers (s)							Speedup per number of workers						
		1	2	4	9	16	24	25	1	2	4	9	16	24	25
nx 120	OpenMP	1.8	-	0.88	0.52	0.46	-	0.53	0.39	-	0.84	1.41	1.62	-	1.39
	MPI	-	0.85	0.49	0.28	0.30	0.34	-	-	0.87	1.51	2.67	2.43	2.17	-
	k-Parallel	0.77	-	0.31	0.23	0.25	0.24	-	0.96	-	2.38	3.20	3.01	3.04	-
nx 180	OpenMP	4.13	-	1.57	1.02	0.84	-	1.11	0.41	-	1.08	1.66	2.01	-	1.53
	MPI	-	2.41	1.50	1.46	1.54	1.60	-	-	0.70	1.13	1.16	1.10	1.06	-
	k-Parallel	1.78	-	0.59	0.39	0.40	0.37	-	0.95	-	2.85	4.37	4.23	4.54	-

Table 1: Runtime and speedup based on the number of worker. As a reminder, the baselines values are 0.74 s for $nx = 120$, and 1.69 s for $nx = 180$.

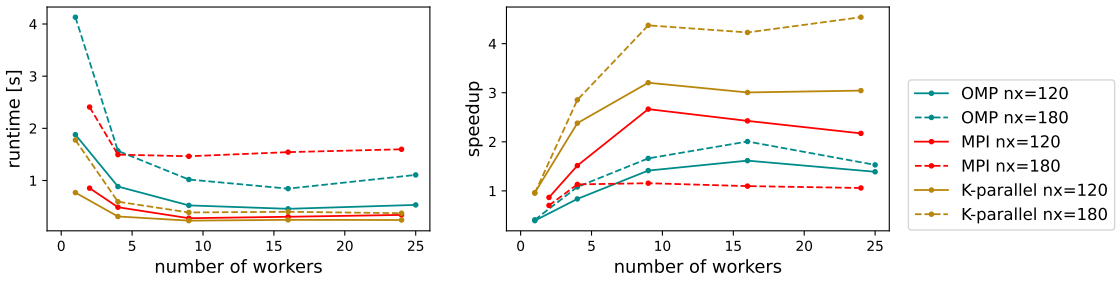


Figure 3: Strong scaling : runtime (in seconds) for each implementation (right hand side) and corresponding speedup compared to a baseline code, that is completely sequential (left hand side).

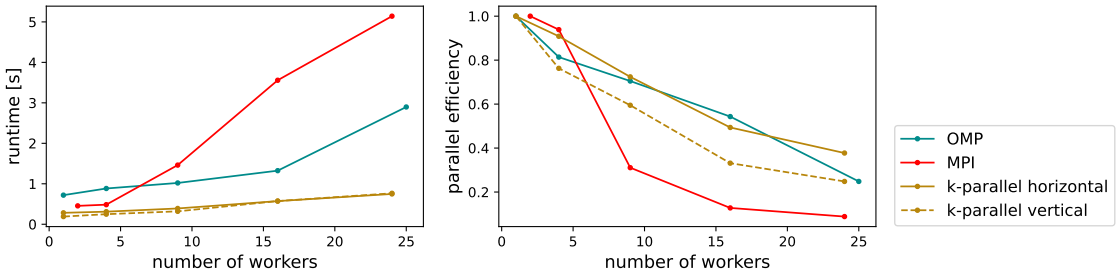


Figure 4: On the left side: time to compute the code with increasing size of the domain (weak scaling). On the right: parallel efficiency of the code with respect to the number of workers.

	Runtime per number of workers (s)							Parallel efficiency per number of workers						
	1	2	4	9	16	24	25	1	2	4	9	16	24	25
OpenMP	0.72	-	0.88	1.02	1.32	-	2.9	1	-	0.81	0.70	0.54	-	0.25
MPI*	-	0.45	0.48	1.46	3.56	5.14	-	-	1	0.94	0.31	0.13	0.09	-
k-Parallel horizontal	0.28	-	0.31	0.39	0.57	0.75	-	1	-	0.91	0.72	0.49	0.38	-
k-Parallel vertical	0.19	-	0.25	0.32	0.57	0.76	-	1	-	0.76	0.59	0.33	0.25	-

Table 2: Parallel efficiency per number of workers. Note that the MPI implementation is of difficult interpretation as it can not be executed with a single worker, thus invalidating the equation in Gustafson's law.

3.2 Weak scaling

The response of the various implementations to Gustafson's law is very different. The results can be found on Figure 4 or Table 2. The most striking result is the low scalability of MPI. With a few workers (2 to 4) the implementation has a runtime comparable to both OpenMP implementations, but already with 9 workers the parallel efficiency drops below 40%, and for 24 workers it even reaches 9%. The reason for that is probably, like in the strong scaling, the overhead of the messaging interface, with many messages sent for a small volume of the computational grid.

On the other hand both OpenMP implementations scale similarly, even though the initial runtime of our implementation is higher. The reason for that is probably due to a bigger initial overhead in our implementation, where many fields are allocated and various functions are called even if not needed in the case of a single thread.

A very interesting result for the k-parallel version is that the way the volume is increased is very relevant for the calculation of the parallel efficiency of the code. The difference between the two methods is explained in more details in subsection 2.3. The "k-parallel horizontal" version, which has fixed amount of vertical layers and variable xy-layer size, has a fixed overhead of distributing 64 layers among the available threads. On the other hand, the "k-parallel vertical" version has a fixed xy-layer size, and each time 4 threads per layer are added, meaning that the overhead increases with increasing volume. This is probably the reason why it starts with a lower runtime but has a worse parallel efficiency than its horizontal counterpart.

4 Conclusions and outlooks

With the growing need for finer resolution models, a single solution is not likely to solve all the answers. A combination of new and existing solutions seems therefore to be one of the few way to reach the goals set for the near future.

OpenMP has proved itself to be a valid and helpful tool for shared-memory parallel programming.

It's easy of use and efficacy allow for rapid parallelization of many tasks, with very little overhead. However, the risks of race conditions are great and, in applications where communication is needed, standard OpenMP implementations seem to be bottle-necked.

On the other hand, MPI allow easy implementations of parallelizations in computations where communication is needed, at the cost of a higher overhead. For small computational tasks, the implementation of this platform seems to be weighting in favour of a simple sequential computations, thanks to easier readability, shorter codes, and comparable runtime. In the case of bigger systems, however, the benefits largely outweigh the disadvantages.

In this report, we have shown that OpenMP can be used for distributed memory systems, with little performance differences to standard OMP implementations, at the cost of simplicity. However, the model presented in this report presents another big advantage compared to a simple k-parallel implementation. Indeed, it can be implemented even when vertical layers are interacting with each others. This is not uncommon in weather and climate models, and can be an interesting use of an implementation that falls between two very different categories.

In the future, one could look into a more general implementation of such a code, where the domain size and partition is not limited to a perfect square numbers, and the divisibility of the domain should not be an issue. This would allow a more accurate investigation of this paradigm, as the simulation could be run for a larger selection of the number of threads.

References

[1] Victor Eijkhout. "Parallel Programming in MPI and OpenMP". In: *The Art of HPC* 2 (2022). URL: `https://github.com/VictorEijkhout/TheArtOfHPC_vol2_parallelprogramming`.

[2] Thomas Rauber and Gundula Rünger. *Parallel Programming for Multicore and Cluster Systems*. 3rd Edition. SpringerLink, 2012.

[3] Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann. "Programming Distributed Memory Sytems Using OpenMP". In: *IEEE Xplore* (2007).

[4] M. Xue. "High-Order Monotonic Numerical Diffusion and Smoothing". In: *Monthly Weather Review* 128 (2022), pp. 2853–2864. URL: `https://doi.org/10.1175/1520-0493(2000)128%3C2853:HOMNDA%3E2.0.CO;2`.

A Pseudo-code of apply_diffusion subroutine

Arguments : *in_field, out_field* : fields ($nx \times ny \times nz$) ▷ *in_field* has additional halo
α, num_iter : diffusion coefficient, number of timesteps

Variables : *sub_infield* : ($sub\_nx + 2 * num\_halo \times sub\_ny + 2 * num\_halo \times nz$)
tmp1_field : ($sub\_nx + 2 * num\_halo \times sub\_ny + 2 * num\_halo$)
halo : same size as in_field
i, j, k / a, b : indices for the subdomain / in_field or halo array

call subdomains() ▷ Allocate *sub_infield* and *tmp1_field*
warmup
$halo \leftarrow in\_field$

Parallel
  Private : *sub_infield, tmp1_field, laplap, thread_id, iter, i, j, k, a, b*
  Shared : *in_field, halo, out_field, nx, ny, nz, sub_nx/ny, num_iter, n_threads, α*
  call adjust_index (i, j, thread_id) ▷ Adjust index based on *thread_id*
  $sub\_infield \leftarrow in\_field(adjusted\_index)$
  for $iter \leftarrow 1$ **to** num_iter **do**
    for $k \leftarrow 1$ **to** nz **do**
      while $(i, j) \in sub\_infield$ **do**
        $tmp1\_field \leftarrow$ Calculate laplacian
      end
      while $(i, j) \in tmp1\_field$ **do**
        $laplap \leftarrow$ Calculate laplacian
        if $iter == num\_iter$ **then**
          $a, b \leftarrow i, j$
          call adjust_index(a, b)
          $out\_field(a, b, k) \leftarrow sub\_infield(i, j, k) - \alpha \cdot laplap$
        else
          if $(i, j) \in inner\_region$ **then**
            $sub\_infield(i, j, k) \leftarrow sub\_infield(i, j, k) - \alpha \cdot laplap$
          else
            $a, b \leftarrow i, j$
            call adjust_index(a, b)
            $halo(a, b, k) \leftarrow halo(i, j, k) - \alpha \cdot laplap$
          end
        end
      end
    end
  end
EndParallel