High Performance Computing for Weather and Climate

Group Project:

UNSTRUCTURED GRID

ALICE MAZZOLENI, HUGO BANDERIER, ANDRIN JÖRIMANN

August 2022

Abstract

The choice of grid for a model depends on the requirements the grid needs to meet. Unstructured grids consist of elements that are similar, but variable in location, orientation and size. They support the use of dynamic resolution and can cover a domain very uniformly. Their implementation, though, requires different approaches than a "regular" grid. We use the `Gmsh` open source mesh generator to tessellate the physical square domain and perform a diffusion test on it. For this, we use a discretized version of the Laplace operator that relies on a lookup table to find neighbouring elements. A rectangular grid serves as a performance baseline to compare against. We find that our unstructured grid models the general characteristics of a diffusion test case satisfactorily, but runs more slowly than the equivalent baseline experiment. Preserving data locality of elements on our unstructured grid by reindexing the elements with a Morton-like curve does not significantly improve the runtime. We attribute the slower performance to higher memory complexity due to more time-consuming accesses of fields that are generally not stored sufficiently locally in memory to avoid reloading caches.

1 Introduction

For the development of any weather and climate model a grid, on which discretization takes place, must be chosen. A multitude of grids can be used to meet different requirements. While most grids traditionally consist mostly of (quasi-)uniform cells, unstructured grids have more recently also been investigated [1–3]. They are created by tessellation of the physical domain, which divides it into geometric elements that are usually similar in size. As a consequence, the location of the grid points is essentially arbitrary and the physical domain cannot readily be represented by the computational domain as in a regular grid, because the order of the cells is not trivial. Further, operations that are based on an increment in the index to access neighbouring cells (e.g. stencils) do not work in irregular grids. Instead each cells is assigned an index and cell neighbours are stored in a lookup table. While the lookup table can be created before running a program and thus does not add to the completion time, this does make optimizing data locality more complex.

Unstructured grids, however, have some unique features that make them promising for certain applications. Most generally, they avoid the pole problem, as no strict regularity must be maintained, which in some cases leads to singularities on a sphere. Another great opportunity arises from the lookup table; unstructured grids lend themselves nicely to dynamic resolution simulations [1, 3]. Whereas e.g. in a rectangular grid a local refinement of the mesh would require laborious special treatment, any new elements can be treated like any other and simply be included into the lookup table of an unstructured grid, requiring only a one-time expensive operation. Similarly, unstructured grids can be viewed as a more general version of grids overall. The methods that are used to run computations on such a grid of irregular nature are often flexible and thus general-purpose by design. The implementation of these methods, however, is not always straightforward.

In this work, we implement a simple laplacian stencil on scalar fields defined on unstructured meshes in `C++`. The code uses the library `gmsh` [4] to create a mesh on a given region of 2D space, optimizes its indexing, creates a lookup table and other utilities to speed up the computation and finally performs one or several stencil steps. The program also allows to create a mesh out of a regularly-spaced square grid in order to benchmark the implementation against the standard finite-difference laplacian. We show that despite the various optimizations performed on the mesh, the lookup table implementation is always slower than the finite-difference version, and explore a few leads to try and explain this slowup.

2 Discrete laplacian on unstructured grids

Our implementation of the discrete Laplacian on unstructured triangular or quadrilateral meshes is loosely based on the so called C-grid implementation used in the ICON shallow water model [5]. In our implementation, the scalar fields (including the divergences of vector fields) are defined at the barycenter of mesh elements (triangles or quadrilaterals) and their finite difference equivalent need one index corresponding to the mesh element. They correspond to space-averaged values of this field over the mesh element. The only vector fields to consider are the normal gradients of scalar fields, defined on the elements' edges. They therefore need two indices : one ranging over the elements and one over its edges. We call N the number of sides of an element (3 or 4) and $\omega_i(j)$ the function that returns the jth neighbour of element i, where the jth neighbour of i shares element i's jth side. The length of element i's jth side is noted $l_{i,j}$ and the distance between the center of i and it's jth

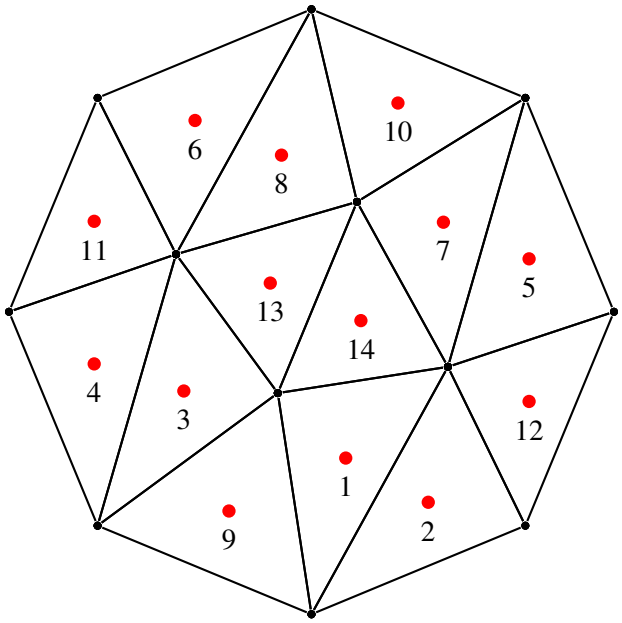


Figure 1: Example of unstructured grid.

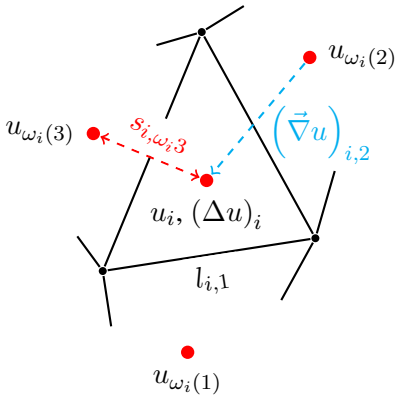


Figure 2: Useful quantities to define the laplacian on an unstructured grid.

neighbour is $s_{i,\omega_i(j)} = s_{\omega_i(j),i}$

2.1 Discrete normal gradient.

On each edge j of the mesh element i, we define the gradient of a scalar field u $\left(\vec{\nabla}u\right)_{i,j}$. As the mesh is irregular, the vector from the center of an element to the center of it's jth neighbour is not necessarily colinear to the outwards normal vector $\vec{n}_{i,j}$ of the corresponding edge, nor does it necessarily intersect that edge at its midpoint. Nonetheless, we use the following naïve definition of the gradient on an element's edge, keeping in mind that it will most probably cause truncation errors [6] :

$$\left(\vec{\nabla}u\right)_{i,j} \cdot \vec{n}_{i,j} \approx \left|\left(\vec{\nabla}u\right)_{i,j}\right| \approx \frac{u_{\omega_i(j)} - u_i}{s_{i,\omega_i(j)}} \tag{1}$$

2.2 Discrete divergence using Gauss's theorem.

For a smooth enough 2D vector field $\vec{f}$, on a surface Ω of boundary $\partial\Omega$:

$$\int_\Omega \vec{\nabla} \cdot \vec{f} \mathrm{d}\sigma = \int_{\partial\Omega} \vec{f} \cdot \vec{\mathrm{d}}n \tag{2}$$

where $\vec{\mathrm{d}}n$ is the boundary's outward normal vector. For the discrete case we can rewrite :

$$|\Omega_i| \left(\vec{\nabla} \cdot \vec{f}\right)_i = \sum_{j=1}^{N} l_j \vec{f}_{ij} \cdot \vec{n}_j \tag{3}$$

2.3 Discrete laplacian.

Combining the two previously defined operators, we can rewrite :

$$(\Delta u)_i := \left(\vec{\nabla} \cdot \vec{\nabla} u\right)_i \approx \frac{1}{|\Omega_i|} \sum_{j=1}^{N} l_j \left(\vec{\nabla} u\right)_{i,j} \cdot \vec{n}_j \approx \frac{1}{|\Omega_i|} \sum_{j=1}^{N} l_j \frac{u_{\omega_i(j)} - u_i}{s_{i,\omega_i(j)}} \tag{4}$$

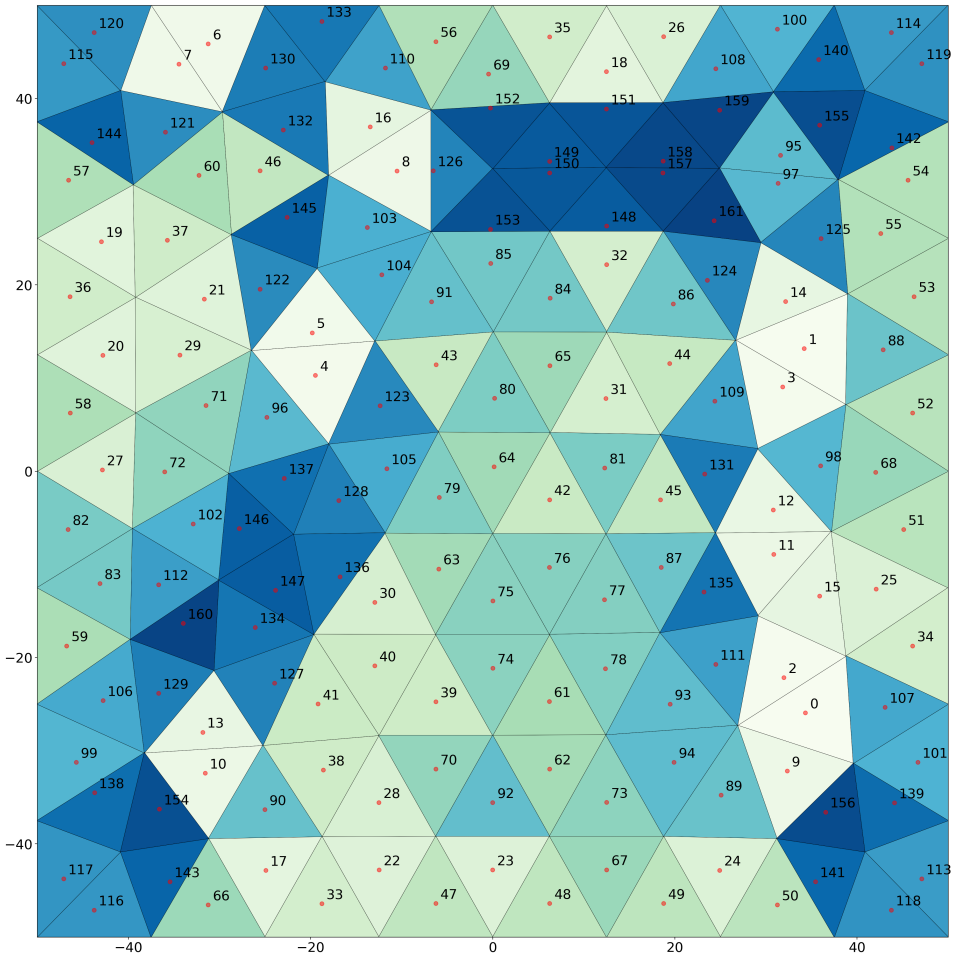


Figure 3: Representation of the triangular mesh as created by `gmsh`. The element indices are given at the center of each cell. The color of each cell changes according to its index. The lack of extensive smooth color gradients indicates how the indices are assigned seemingly randomly.

3 Locality-preserving indexing of mesh elements

3.1 Morton-like curve for unstructured meshes

The computational efficiency of a stencil operation can be improved by increasing data reuse while it is still in the cache, instead of having to reload it from memory which is much slower. This is typically achieved by increasing the ”locality” of the data that the stencil uses. Fields on unstructured grids are stored in 1D arrays whose order in memory does not necessarily represent the 2D positions of the element. In this context, locality of data means making sure that elements close in 2D space are also close in the 1D array. In this work, we

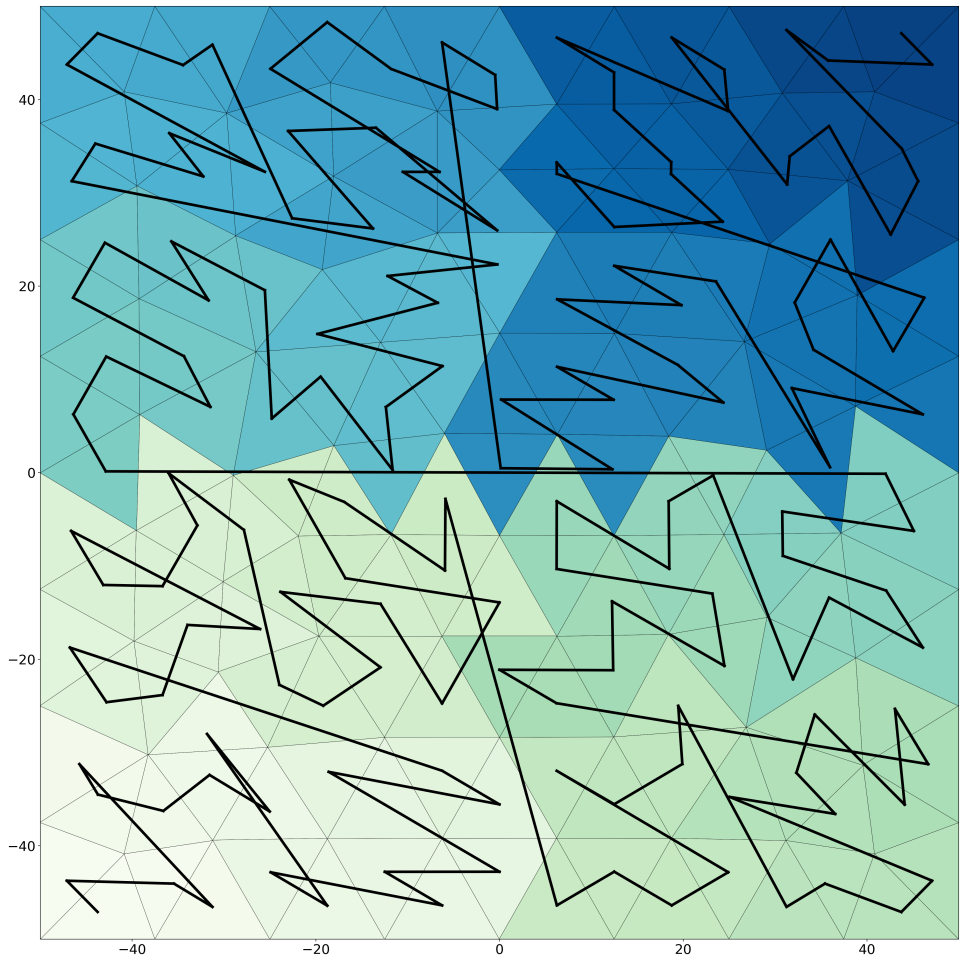


Figure 4: Representation of the localized triangular mesh reindexed using the GeoHash algorithm. The elements are indexed more neatly in sectors as the Z-like curve shows, but some jumps are still present.

choose to use an algorithm similar to the GeoHash algorithm, but in flat 2D space. The algorithm allows one to use a construction identical to the Morton space-filling curve without the need for an underlying square grid. The procedure goes as follows [7]:

1. Convert the coordinates of each element's center to binary using bissection. Our grid is defined on $[-50, 50] \times [-50, 50]$ and we take an element whose center is at $(35, -28)$ as an example. The first bit of the binary string encoding the x-coordinate will be 1, as $x > 0$, the second one will be 1 as $x > 25$ etc... until we convert these coordinates to (1101100, 0011110).

2. Lump the two binary numbers together, alternating with one from y and one from x, to get 01011011111000.

3. Convert to integer and `argsort` to get the mapping from any indexing to the "optimal" indexing.

When the mesh is created by `gmsh`, the elements are seemingly randomly ordered, as can be seen on Figure 3. After applying the GeoHash algorithm, the mesh elements are ordered such that neighbouring triangles have a close index, see Figure 4. While less optimal than Hilbert curves [8] in terms of locality preservation, the choice was made to use a Z-like curve thanks to its simple implementation on arbitrary grid sizes, and flexibility towards missing hashes and arbitrary hash sizes.

3.2 Generalized Hilbert curve for structured meshes

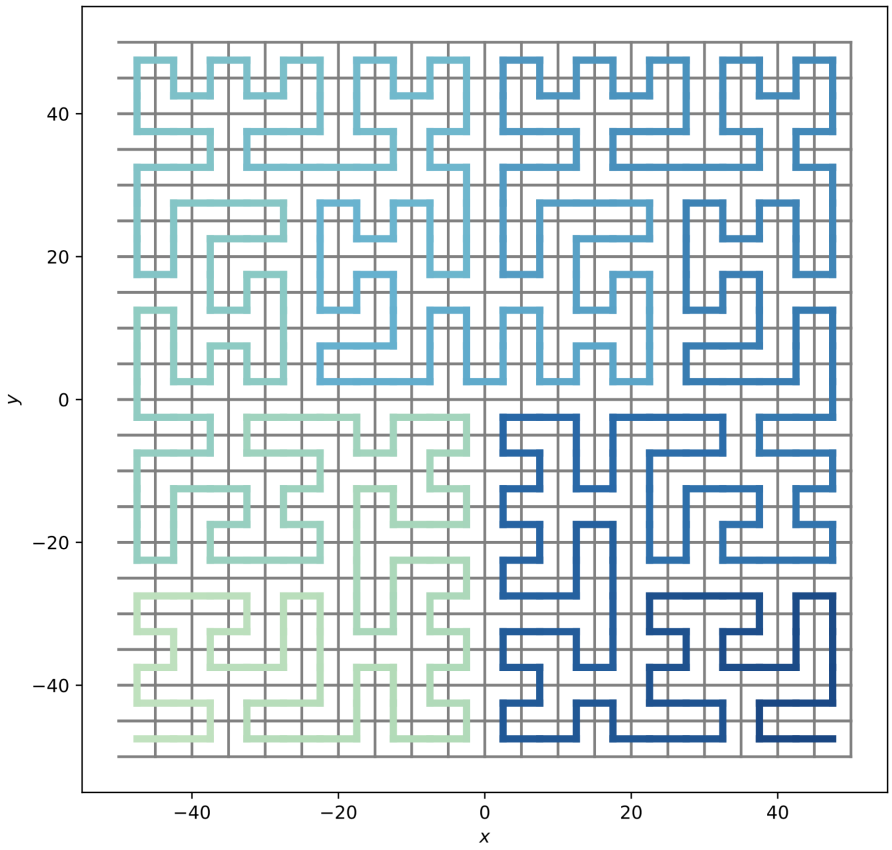


Figure 5: Generalized Hilbert curve on a 20×20 grid. The color from light green to blue represents increasing index.

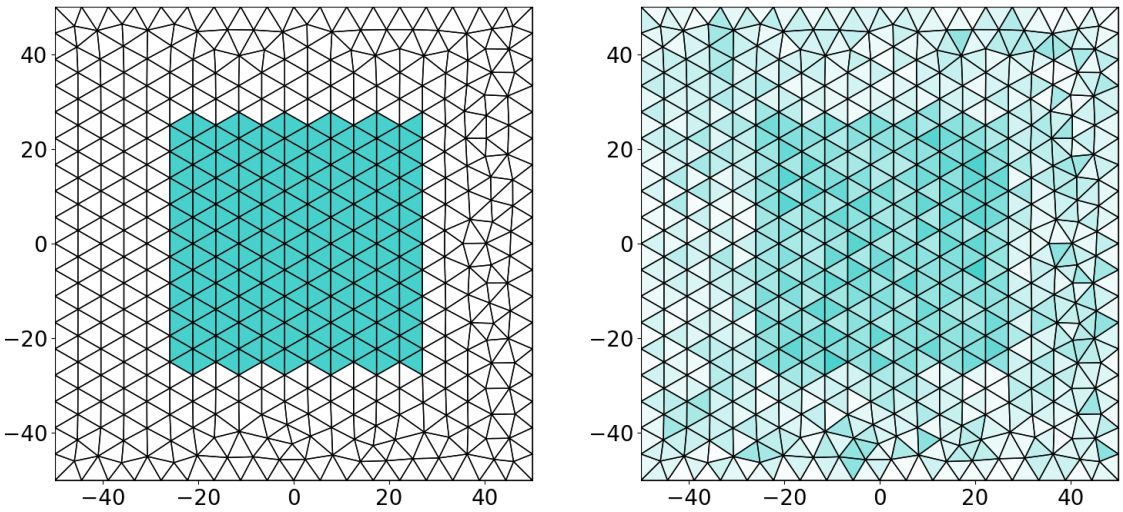
For meshes explicitly constructed from a 2D rectangular grid, a solution to the locality problem exists : Hilbert curves. These space filling curves go once through every element of a grid and preserve locality very well. Normally defined on square grids of sizes $2^n \times 2^n$, they can be generalized to arbitrary rectangular grids [9]. The result of this particular algorithm can be seen on Figure 5. This mapping is very useful to compare our implementation of the laplacian against the finite-difference one on identical grids.

4 Results

In Figure 6 one can see the visualisation of the diffusion operator with constant diffusion coefficient ($c = 1$) for an unstructured triangular mesh with a total of 774 elements defined on the domain $[-50, 50] \times [-50, 50]$ and created by `gmsh`. We used the discrete Laplacian introduced in Section 2.3 to discretize the Laplacian and used Forward Euler with a timestep of 0.1 to discretize the time derivative.

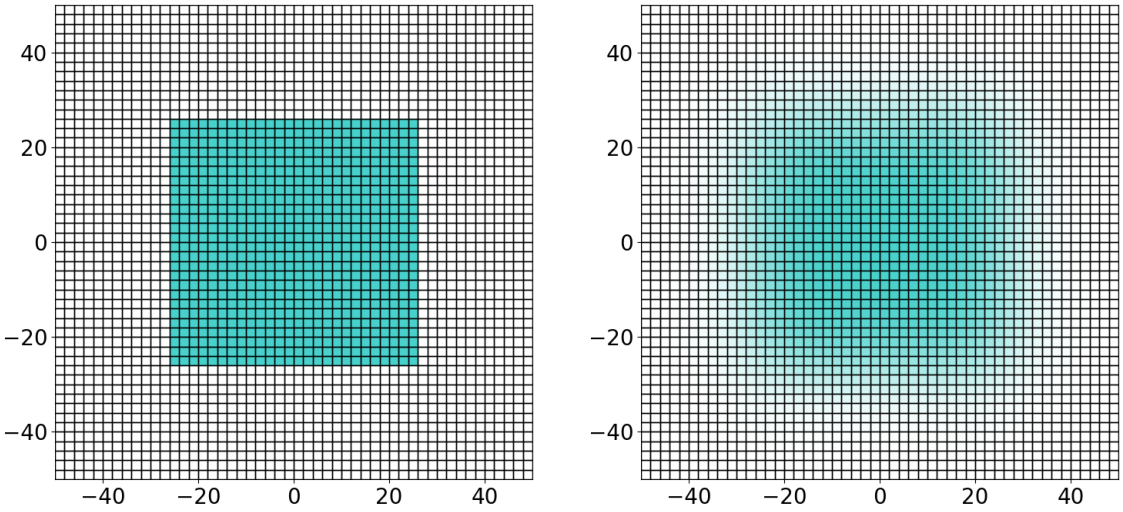
$$\begin{aligned} \frac{\partial u(x,t)}{\partial t} &= c\Delta u(x,t) \\ \frac{u_i^{n+1} - u_i^n}{\Delta t} &\approx \frac{1}{|\Omega_i|} \sum_{j=1}^{N} l_j \frac{u_{\omega_i(j)} - u_i}{s_{i,\omega_i(j)}} \end{aligned} \tag{5}$$

As an input to the scheme, we use a field initially defined on a square grid, that is full of zeros except for a central square of ones. We then use bilinear interpolation to map this input onto our mesh. Alternatively for the benchmark on a regular square grid, we can directly use the known gilbert curve to map the field onto the mesh. This is done on Figure 7, where this time the mesh was created explicitly from a 50×50 square grid.



(a) t = 0 (b) t=100

Figure 6: Diffusion operator for triangular mesh after 100 time steps.



(a) t = 0 (b) t=100

Figure 7: Diffusion Operator for an explicitly constructed rectangular mesh after 100 time steps.

4.1 Performance

All performance measurements were done on a MacBook Air with a 1.3 GHz Dual-Core Intel Core of type i5. The machine has a L2 cache size of 256 KB per core and a L3 cache size of 4 MB. The size of a cache line is 64 bits or 8 doubles. We start by comparing the time it takes to build the mesh using `gmsh`, create the lookup table and reindex the mesh used to create Figure 6 which has 772 elements, and show the result on Table 1. The various initialization times can easily be ignored at least for such small grids.

Action	Building mesh	Creating lookup table	Reindexing
Time [s]	0.0795335	0.0530452	0.0140329

Table 1: Initialization times.

4.1.1 Stencil computation benchmark on square grids.

We then move on to the benchmarking of the stencil computation. On Figure 8 one can see the required time to compute the discrete Laplacian from Section 2.3 compared to the Laplacian computed using centered finite differences, on identical square grids. While both methods use roughly the same amount of time on floating point operations their major difference is in their memory complexity. The discrete Laplacian from Section 2.3 has a total of 7 memory accesses times the number of nodes per cell depending on our mesh. In the Code Snippet 1, we can count the number of accesses and also notice that the mesh coefficients $|\Omega_i|$, $s_{i,\omega_i(j)}$ and l_j have been lumped together into a single number $A_{i,j} = \dfrac{l_j}{|\Omega_i| \times s_{i,\omega_i(j)}}$ to minimize both the number of floating-point operations and the number of memory accesses. The discrete Laplacian induced by using finite differences has a total of 9 memory accesses. This is generally less than for the Laplacian in Section 2.3 as the number of nodes per cell is 4 for rectangles and 3 for triangles. However when using sophisticated space filling curves as the Hilbert curve from Section 3.1 we can drastically improve cache locality. Figure 9 shows the histogram of the distances between two neighbours in memory. As one can retrieve from the histogram most neighbours have a distance of less than 5 between each other. As a cache line consists of 8 doubles most neighbours share a cache line. In the finite difference approach on the other hand one has to load a new cache line half of the time when accessing a new neighbour, given that the cache line has not already been loaded.

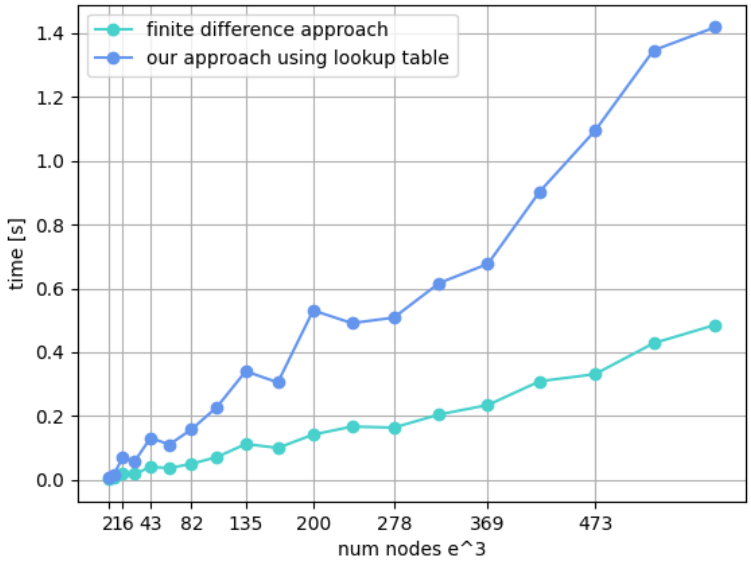


Figure 8: Comparison of average runtime to compute the discrete Laplacian using method 2.3 and a finite difference scheme.

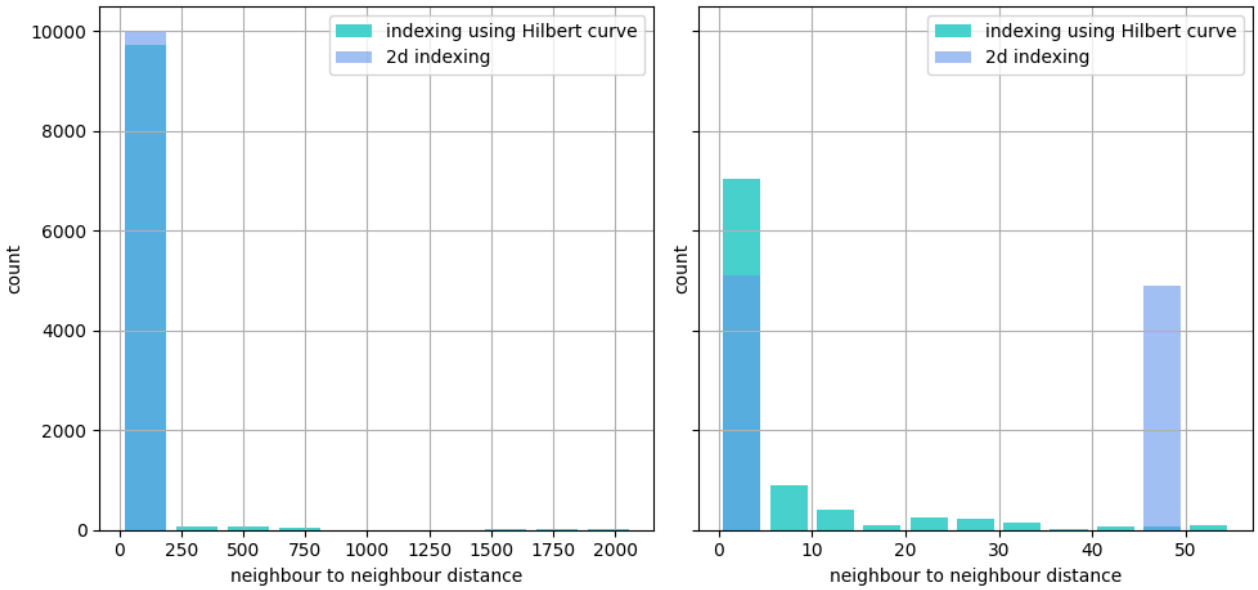


Figure 9: Most neighbours are on the same cache line when using indexing induced by the Hilbert curve.

As one can see in Figure 8 our discrete Laplacian takes drastically longer to compute compared to the finite difference version. Although neighbouring elements are much closer in memory we need to access a lot more fields which are not necessarily anywhere near each other in memory - except by change.

4.1.2 Runtime on unstructured triangular grids.

Table 2 also makes clear that the performance is probably bounded by the accessing of several fields and not by the indexing of the neighbours, since the reindexing has no significant effect on the performance.

```cpp
void Mesh::laplacian(const Eigen::VectorXd& field_prev, Eigen::VectorXd& out_field) const
{
    int neighbour;
    for(std::size_t i = 0; i < numElements; ++i){
        out_field(i) = 0;
        for(std::size_t j = 0; j < numNodePerElement; ++j){
            neighbour =  lookup_table(i,j);
            if (neighbour == -1)
                out_field(i) += lookup_table_coeff(i, j)
                                * (boundary_value - field_prev(i));
            else
                out_field(i) += lookup_table_coeff(i, j)
                                * (field_prev(neighbour) - field_prev(i));
        }
    }
}
```

Listing 1: C++ implementation of the discrete Laplacian as introduced in section 2.3

```cpp

void compute_laplacian(MatrixXd& in_field, MatrixXd& out_field,
                       const double step_size_x, const double step_size_y)
{
    //based on second order centered finite difference
    for(std::size_t i = 1; i < in_field.rows() - 1; ++i){
        for(std::size_t j = 1; j < in_field.cols() - 1; ++j){
            out_field(i, j) = (in_field(i, j + 1) - 2 * in_field(i, j)
                               + in_field(i, j - 1)) / step_size_y
                            + (in_field(i + 1, j) - 2 * in_field(i, j)
                               + in_field(i - 1, j)) / step_size_x;
        }
    }
}
```

Listing 2: C++ implementation of the discrete Laplacian using centered finite differences.

Number of elements	Time without reindexing [ms]	Time using reindexing [ms]
11682	13.6975	14.1343
14570	17.5595	16.9562
18414	21.592	20.8967
24124	25.6255	25.2951
32162	38.3143	38.756

Table 2: Computation times with and without GeoHash reindexing.

5 Conclusion

In this work, we have demonstrated the capabilities of our unstructured mesh program for stencil operations on scalar fields. The program builds a mesh and creates its lookup table in a short time. It then interpolates an

arbitrary input field and performs stencil operations, outputting reasonable results but with a slowup of almost 3x when compared against a finite differences implementation. This slowup can come from several places but the memory access is the most likely cause. This is despite the efforts in reindexing the various arrays to increase locality. The Hilbert curve indexing for square grids, and the Z-like curve indexing for unstructured grids, while noticeably increasing locality, end up having disappointing effects on the runtime.

References

1. Bacon, D. P. *et al.* A Dynamically Adapting Weather and Dispersion Model: The Operational Multiscale Environment Model with Grid Adaptivity (OMEGA). *Monthly Weather Review* **128,** 2044–2076. ISSN: 0027-0644. doi:10.1175/1520-0493(2000)128<2044:ADAWAD>2.0.CO;2 (7 July 2000).
2. NCAR. *Model for Prediction Across Scales* https://mpas-dev.github.io/ (2010).
3. Linardakis, L., Reinert, D. & Gassmann, A. *Icon grid documentation* tech. rep. (Tech. rep., DKRZ, Hamburg, available at: https://www. dkrz. de/SciVis/hd-cp ..., 2011).
4. Geuzaine, C. & Remacle, J.-F. *Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities* 2009. https://gmsh.info/.
5. Wan, H. *et al.* The ICON-1.2 hydrostatic atmospheric dynamical core on triangular grids – Part 1: Formulation and performance of the baseline version. *Geoscientific Model Development* **6,** 735–763. https://gmd.copernicus.org/articles/6/735/2013/ (2013).
6. P. A. Jayantha, I. W. T. A comparison of gradient approximations for use in finite-volume computational models for two-dimensional diffusion equations. *Numerical Heat Transfer, Part B: Fundamentals* **40,** 367–390 (2001).
7. Liu, Z. *et al.* Development and performance optimization of a parallel computing infrastructure for an unstructured-mesh modelling framework. *Geoscientific Model Development Discussions* **2020,** 1–32. https://gmd.copernicus.org/preprints/gmd-2020-158/ (2020).
8. Moon, B., Jagadish, H., Faloutsos, C. & Saltz, J. Analysis of the clustering properties of the Hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering* **13,** 124–141. doi:10.1109/69.908985 (2001).
9. Červený, J. *gilbert2d* Aug. 29, 2022. https://github.com/jakubcerveny/gilbert.