We are grateful to the referee for the very helpful comments and given suggestions.

1. **The main subject of the paper is the MPI implementation and load balancing, but I suspect some aspects of the model are limiting on-node performance. For example, they describe their choice of conditional masking vs multiplicative masking for land points (pg 2 line 42), their non-optimal combination of loop and index ordering (pg 2 L90), and the potential advantages of unstructured meshes (p2, L41). They have included at least some discussion of these in the paper so I'm not suggesting any changes now, but may have some implications on later comments below and would encourage them to explore these as they continue their optimizations in the future.**

    In this model, boundary conditions are included into matrix elements, which are stored as an array KT(6,13), where the first dimension corresponds to 6 triangles composing Finite Element, and the second dimension corresponds to 13 types of "wet" points: 1 inside the domain and 12 types of boundary points. This approach is similar to multiplicative masking, as B.C.s are applied by the product to KT, and to unstructured mesh models, as matrix elements are precomputed. The difference from the unstructured mesh models is that only unique elements of the matrix are stored and neighboring points are referenced directly. So, the mask of wet points serves only to restrict the number of computations. We thank the referee and will think in the future how to organize calculations more efficiently.

    In our opinion, for the model we have for now, it is not reasonable to change loops' order, as it harms model infrastructure and our parallelization approach, but array indices may be chosen more optimally, setting "depth" index as the first. Nevertheless, this interchange is not crucial for the goals we address in the paper.

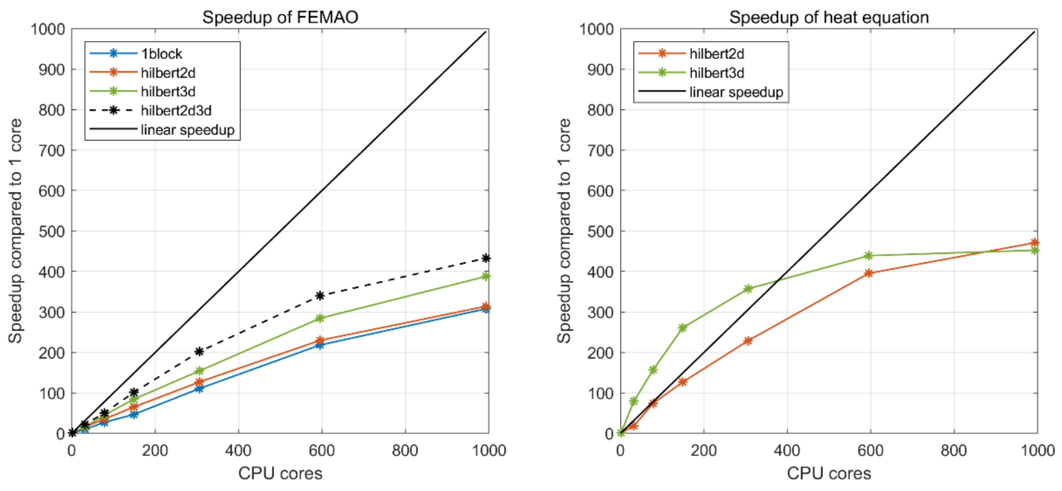
2. **First, the authors show speedups in figure 4 with significantly super-linear speedups in the 2-d case. They attribute this to cache performance without additional evidence (eg from hardware counters or other performance tools). That may be the case, but I think this super-linearity is large enough to warrant further exploration into the cause.**

    This super-linear speedup is measured for the code section of approximate length 1000 code lines which consist of 6 loops like in algorithm 3. We stress that these loops have slightly different organization of the calculation and may accelerate in slightly different rates. Some of the loops work with 4D arrays, where the first additional dimension corresponds to "antidiffusive fluxes". Some loops have additional if-conditions, which are needed to perform flux correction in quasi-monotone scheme. Superlinearity occurs at the low-to-middle number of cores, and these cases are usually omitted when scaling up to many cores is shown. Moreover, usually speedups are shown including exchanges, and for this option our speedups are not superlinear.

    Finally, from the practical viewpoint, the presented parallelization approach together with the chosen loops/indices ordering may lead to superlinear acceleration. As an example, consider very simple "heat equation" loop:

```
do j = js, je
    do i = is, ie
        do k = 1,depth(i,j)
            Tn(i,j,k) = 0.25_8 * (T(i+1,j,k) + T(i-1,j,k) + T(i,j+1,k) + T(i,j-1,k))
        end do
    end do
end do
```

It accelerates superlinearly at the low-to-middle number of cores for appropriate weights even when MPI exchanges are taken into account (green line in the right subfigure):



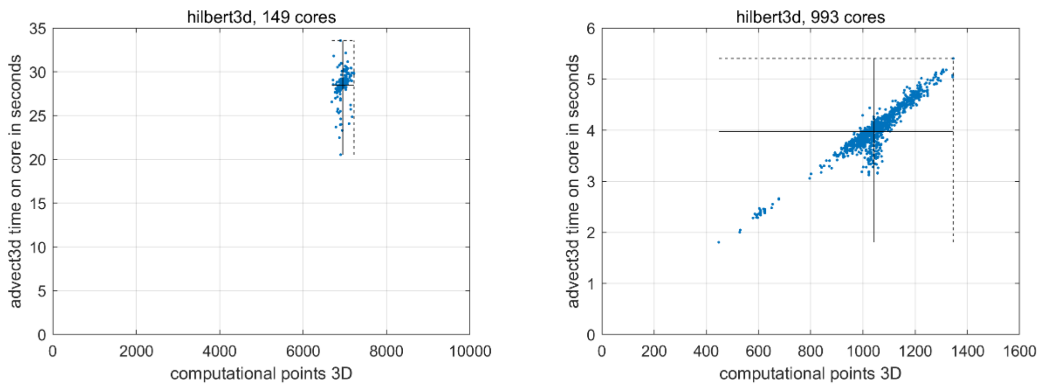
Regardless of the actual reason it happens (decrease in the number of cache misses, non-optimal organization of the calculations, or something else), this "heat equation" loop constitutes what we actually intended to do, and there is nothing to optimize here.

3. **Third, the large variation in work load at high core counts (fig 5,6) also seems higher than one might expect. As you get fewer points/blocks per core, there will naturally be a little higher variability, but this seems larger than expected and might point to additional problems.**

   As the referee pointed out, balancing could be better. In experiment presented in the manuscript, balancing is limited by the outlier point (figs. 5 and 6), corresponding to the CPU core with 5 blocks and maximum load. This outlier point limits LI to 46%. We have checked balancing optimization procedure (algorithm 4) and found that it doesn't guarantee monotone decrease of LI, as subroutine "remove_not_connected_subdomain" can increase LI. After choosing the best iteration, LI was decreased to 29%. As this behavior is crucial only for "hilbert3d 993 cores" experiment, in the revised manuscript we will update it. Additionally, we have tested METIS multilevel k-way contiguous partitioning algorithm and found that it doesn't give better balancing (LI=39%).

4. **Second, the computational time as a function of wet points seems a bit counter-intuitive (Fig. 5). The authors have shown percentage of wet points rather than total wet points to emphasize their diagnosis again of memory access. But without also seeing the total number of points (computational load), it's a little hard to get a more complete picture. Again, this effect seems too big to attribute solely to cache effects and it seems like more might be going on here.**

   Figures 5 and 6 are provided to assess separately data structure efficiency and load balancing efficiency and clearly show limitations of the described model. As the referee pointed out, the lack of point-to-point correspondence between 5 and 6 figures lead to incomplete picture of what is going on. Here we provide scatter plot (6 figure y axis – 5 figure y axis):



Scatterplots are provided with mean values (solid lines) and maximum values (dashed lines). These values completely define Load Imbalance (LI) in partition and advect3d runtime. As follows from the left figure, spread in runtime is more then spread in the number of computational points. This means that computations are limited by the organization of the calculations, but not by the accuracy of the partitioning algorithm. As advect3d is a function with approximate length of 2500 code lines, which consists of 6 loops like in algorithm 2, and each loop has slightly different organization of the calculations, we claim that overestimation of runtime LI only by 15% in comparison to partition LI is a very good result. In the right figure, there is strict correlation between the number of computational points and advect3d runtime, and computations are limited by the balancing procedure. As this figure is more informative than figure 6, in the revised manuscript we will attach the new figure.

5. **Fig 2 has cropped the bottom of figures**

   Fig 2 is shown as we expected. We do not provide axis labels as they correspond to the mesh points, but not to geographical coordinates.

6. Minor edits will be taken into account in the revised version of the manuscript.