Halo Update Performance Model

Philip Ploner, Xuying Zeng

August 30, 2024

Abstract

Halo updates are performed as part of a diffusion stencil calculation in C++, using distributed memory parallelism via Message Passing Interface (MPI). The performances of these halo updates are measured with respect to node number $n_{node} \in \{1, ..., 9\}$ and halo size $n_{halo} \in \{2, ..., 10\}$. The time taken for a halo update is found to be linearily increasing with a higher halo number. Increasing the node number shows decreasing performance time with some exceptions. Both behaviours can be explained by the data size needed to be transmitted per node.

1 Introduction

Distributed memory parallelisation is a concept widely used in high performance computing applications such as weather and climate modeling. In those applications, distretized iterators of differential equations, such as stencil operations, are solved not just on one big 3D field, but on several smaller 3D fields in parallel. In order for such a parallelization to work, it is essential that before each iteration of, for example, a stencil step, the boundary conditions of the corresponding computational node are updated. Such a boundary condition update is achieved via Message Passing Interface (MPI) communication with the nodes containing neighbouring field data, and is called halo update.

This project investigates the performance of these halo updates by performing such a distributed memory parallel computation for an example 3D grid. It uses the C++ programming language, thereby improving computational efficiency and speed in comparison to heigher level programming languages such as Python. In particular, the project assesses the performance of the halo update process with respect to the number of nodes used for the computation, n_{node}, and the size of the transmitted halos, n_{halo}.

To this end, it implements the division and distribution of 3D data grids of arbitrary size, passes the necessary boundary conditions between the processes, measures the times taken for each of these halo updates, and locally performs the stencil like diffusion operations on each node. In this way, the developed code enables the parallel solution of large-scale stencil like 3D problems with corresponding quantification of the halo update performance.

2 Method

This section gives an overview over the developed code and explains the most important methods used during the scope of this project in detail. The code runs on the Piz Daint supercomputer, which is accessed through jupyter.cscs.ch.

2.1 Code outline

To perform the halo update performance measurements, a framework of several code files has been built. Its most important components are stated here. More in-depth explanations of individual steps and functions are provided in the comments of the code files themselves as well as the corresponding readme file.

- utils.h contains the storage3D class, which allows to store 3D fields together with metadata corresponding to the dimensions of there various axes and the sizes of their halos.
- partitioner.cpp implements the distribution of a given storage3D element over a given number of nodes. It calculates the dimensions of the subfields and implements scattering and gathering methods based on MPI.
- stencil_nnodes.cpp implements a parallelized version of stenci type diffusion operator and in particular performs the necessary halo updates and their individual performance measurements.

2.2 Implementation of distributed memory parallel computing

In the most general sense, the core of parallel computing lies in the decomposition of a large-scale computational task into multiple smaller scale tasks. Those smaller scale tasks are themselves then assigned to indivdual ranks for simultaneous execution. In the case of distributed memory parallelism this different ranks correspond to different nodes of a super computer. This code implements the parallel solution of a 3D diffusion problem through the following steps:

1. Initialization of the MPI environment. This allows the use of functions from the MPI.h library, and in particular the controlled sending of data between the n_{node} ranks.
2. Initialization of a model 3D field, using the custom storage3D class. The model field used for this project has xyz-dimensions 256x256x64. The inner part, between 1/4 and 3/4 of a given axis, is filled with 1, where as the outer part of the field is filled with 0. A visualization of this model field allows for an easy check of a successful application of a diffusion process.
3. Initialization of an instance of the partitioner class. This decomposes the original 3D domain into n_{node} smaller 3D domains using two dimensional domain decomposition. It also provides a scatter and gather function adapted to the calculated decomposition. A visualization of such a two dimensional domain decomposition is depicted in figure 1.
4. Distribution of the data to the different ranks with the scatter function of the partitioner class.
5. Perform halo update on each node, sending own data corresponding to boundaries of neighbouring nodes to those neighbouring nodes, and receiving the needed boundary data from neigbouring nodes. For the here employed stencil computation, only boundary conditions in the x and y directions are required. A halo size of n_{halo} therefore corresponds to n_{halo} lines each sent to and received from the bottom, left, right and top of each plane on a node. A visualization of such a parallel halo update is depicted in figure 2, and more details about the implementation of these updates are given in section 2.3.
6. Each process independently performs one step of the diffusion stencil on its local subgrid.
7. Steps 5 and 6 are repeated for a number of iterations. For the purpose of this analysis, 1024 iterations have been chosen.
8. One final halo update is performed after the last iteration to synchronize the field data on each rank.
9. The distributed fields are gathered again on the root node using the gather funtion of the partitioner class.
10. The final assembled global field is saved in an output file and compared with a reference diffusion performed without parallelization on single node in order to validate correct diffusion.

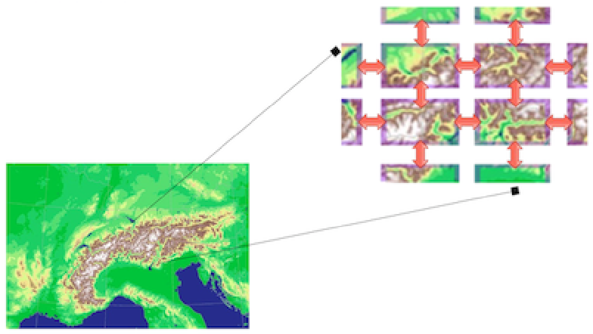

Figure 1: Example visualization of a 2D domain decomposition as performed in weather and climate models[1].

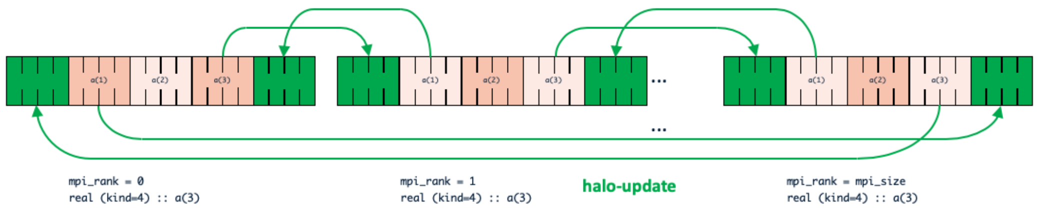

Figure 2: Example visualization of a parallel halo update in memory space[1].

2.3 Implementation of halo updates

There are several steps necessary to perform a correct halo update. First of all, it is necessary to identify the four neighbouring ranks, with which the data exchange will be performed. This is done via corresponding member functions of the partioner class. Then the data a given rank wants to send has to be collected and packaged in a sendbuffer. The data to be sent corresponds to the n_{halo} outermost lines in x and y direction of each plane, that does not correspond to the halo (see figure 2 for a visualization in memory space, and figure 3 for a visualization on a 2D plane). The data to be transmitted has to be package into a one dimensional std::vector in order to ensure it occupying a contiguous memory space. MPI will only send data of a specified size of contiguous memory space starting from the given adress of a start position. As a consequence, sending the storage3D class element directly would result in a shifted field obtained on each node, because there are other spaces inside the memory associated memory block that can contain for example metadata.

After the the data to send per node is packaged inside a std::vector, one then allocates receive buffers of the corresponding sizes on each node and then gives the MPI send and receive commands. The given commands have to be non-blocking, because blocking commands will result in a deadlock situation, where every send command will first wait to get communicated a corresponding receive command on another node, while that node itself will still be waiting for its needed receveive command from yet again another node and so on. Then the system will wait until all the transmission processes are executed, before unpacking its received data again into a storage3D element and proceeding with its corresponding stencil operations.

2.4 Performance measurements

The performance of the halo update is first measured on each node separately. For each call of the halo update function, the time before and after the measurement is taken and the time difference in seconds saved in an array. Therefore, each node will after all 1024 iterations be left with 1025 halo update performance measurements, where the 1025th measurement corresponds to the additional halo

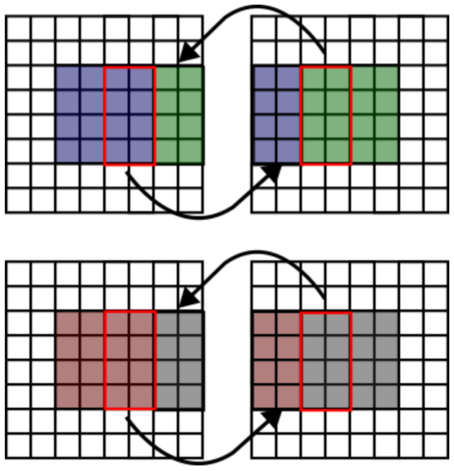

Figure 3: Example visualization of a parallel halo update in 2D space. Note that in our code the left and right halo updates include the corners, unlike in this picture[2].

update after the final iteration.

After all ranks have filled their array of the halo update performances, they will send their vector to the root rank via MPI communication. As the total performance of the calculation is bounded by the node where the halo update takes the longest, for each 1025 halo updates the root rank will only save the time of the lowest performance. Therfore, one run of the stencil_nnodes.cpp script will produce one halo update performance array with 1025 entries, where each entry is the one from the node which took the longest. The arrays are then saved in .csv files for further analysis.

2.5 Probed parameters

The performance measurement is carried out for n_{node} ranging from 1 to 9, because 9 is the highest node number available for access on Piz Daint. The halo number n_{halo} is varied from 2 to 10, where 2 is the minimal halo number needed for the given stencil algorithm to work. Grid size is fixed at 256x256x64 and the number of iterations is also fixed at 1024.

2.6 Data verification

The correctness of the parallelized diffusion code is verified by comparison with a reference field. The reference field is created using the non-parallelized stencil code from the course exercises. Visual comparison, using the validate_results function as also given in the exercises shows identical looking diffused fields.

However, for $n_{node} \in \{3, 5, 6, 7, 8, 9\}$ not all entries of the parallelized field is within the given relative toleranze of 1e-8 and absolute tolerance of 1e-5 to the reference field. In those cases, there is usually a small percentage of field entries having a difference higher than 1e-2. These are most likely caused by the way the partitioner is designed: The fields gathered from the workers can sometimes include halo points of the smaller fields. Most of these halo points are taken from non-halo points of their neighbours during the final halo update. But the outside borders of those halos, i.e. the corners of the fields after the halo update, will not have been diffused in the same way the corresponding non-parallelized entries have in the reference field, because the stencil could not have been applied to them. This can in the end lead to this small number of entries having discrepancies from the reference field.

A potential further investigation would therefore be to look at the way corners are handled in detail

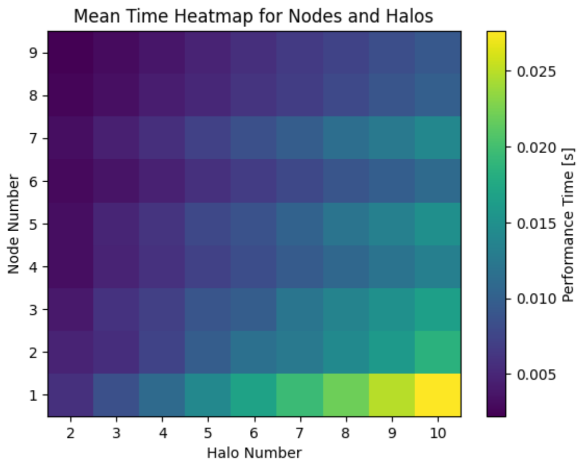

Figure 4: Mean time heatmap for nodes and halos

and try to figure out a solution for those small discrepancies. This will require thorough testing with smaller fields but more than 4 nodes and high numbers of iterations, as the discrepancies are not visible enough at low iteration counts. This was beyond the scope of this project.

3 Performance evaluation

Performance evaluation is critical when performing High Performance Computing application development, especially when dealing with tasks involoving large amounts of data and complex computations. In order to analyse the efficiency of the code and verify its accuracy, the performance measurements outlined above are performed.

In our project, we used series of measurements to evaluate the performance of the code. The data then includes the performance times for different numbers of nodes(n_{node}) and different halo sizes(n_{halo}). Performance time is the time in seconds required to complete the call of the halo update function at the fixed number of nodes. The data under each setting contains time measurements for 1025 halo updates, including 1024 updates before each diffusion operation and the last one after all interations are completed. We can then takes this data and analyse how the code performs under different configurations.

The data files containing the 1025 time measurements for each configuration are stored in CSV format, where the file name contains information about the corresponding number of halos and number of nodes. We used Jupyter Notebook as our main tool during data analysis. We first imported the CSV file into a NumPy array. For each setting, we calculated the mean and standard deviation of the performance times. We plot the mean data for each configuration and use the standard deviation as error bars.

First, we plotted heatmaps based on the performance mean time at different settings(see Figure 4). Two main patterns are clearly observed from the heatmaps: when the halo size n_{halo} increases, the performance mean time shows an upward trend(see Figure 5); while the number of nodes n_{node} increases, the performance time then shows a decreasing trend(see figure 6). Among all the number of nodes, when $n_{node} = 1$, the performance mean time for the growth of n_{halo} is the most responsive, and accordingly, the longest run time occurs when the $n_{node} = 1$ and $n_{halo} = 10$.

In the graphs we can see that the error bars, which come from the standard deviation of the arrays, are small in relation to the plotted mean values. For the further arguments we consider the data therefore as precise. Looking at the data arrays, we see that high standard deviations are usually

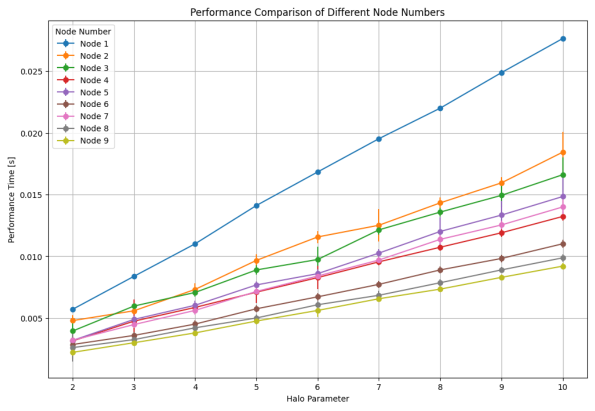

Figure 5: Performance comparison of different halo numbers under corresponding node numbers

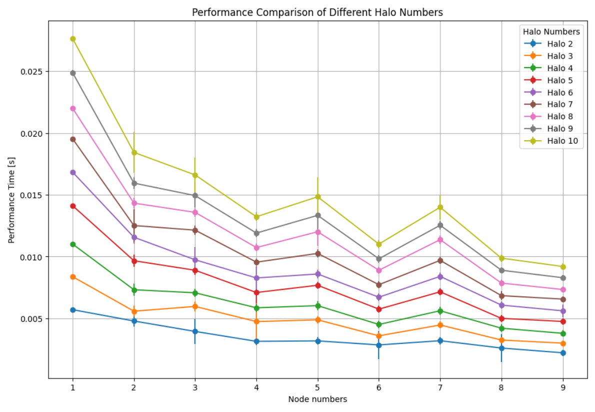

Figure 6: Performance comparison of different node numbers under corresponding halo numbers

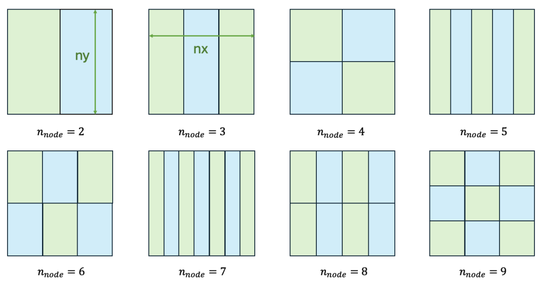

Figure 7: Distribution of one xy plane for different node Numbers

connected to a small number of iterations taking unusually long. It can also be observed from Figure 5 that certain data points with large standard deviation are offset. For example, when the number of nodes $n_{node} = 2$ and $n_{halo} = 3$, the running time is higher than the case when the number of nodes $n_{node} = 3$ and $n_{halo} = 3$. This behaviour may be related to some volatility of the high performance computer performance and might be reduced by running the measurement series multiple times. As the performed operations are the same in each iteration, there is no other obvious reason for these outliers.

These behaviours can all be explained by the size of the data transferred between the different nodes: in parallel computing, when the number of nodes increases, the amount of data to be processed by each node decreases, and therefore the computation time per node is shortened. The change in processing time with n_{node} is closely related to the division of the halo region(see Figure 7). As n_{node} increases, the size of the region that each node needs to process decreases, but not linearly. This means that in general less boundary data needs to be transferred between nodes, leading to a decreae in data transfer time, which makes the overall processing time shorter. However, there are exceptions: when n_{node} is equal to 5 or 7, according to the way the region is divided, there are still as many values in the y direction of the graph as in the original array despite the division in the x direction, so the data that needs to be transmitted will still produce small peaks on Figure 6, although the parallelism still reduces the total computational time. However, despite this, we can see an overall decreasing trend.

4 Conclusion

The behaviour of the performance time measurements can be completely explained by the number of data points that needs to be sent. Therefore we can conclude that to optimize a MPI parallelized computation one should always try to distribute the data in such a way that as little information has to be sent as possible. There may be cases where a higher node number actually increases the size of the data to be exchanged in a halo update, in such a case this would then need to be weighted carefully with the performance benefit of the rest of the calculations that one gets with the further parallelization. Furthermore, this investigation shows that there is a performance loss associated with high n_{halo}, so the halo size should always only be choosen as small as necessary.

Possible further investigations could strengthen the precision of the acquired data by performing multiple runs of the measurement series. This would account for irregularities from occasional super computer performance drops. Additionally, one could scale up the parameters more and see if the observed behaviour continues in the same way. It could be possible to see a limiting behaviour especially with high n_{node}, when there is so little data to be sent that the overhead of the MPI communication protocols will dominate.

In general, the same methods that decrease total computation time, i.e. increase parallelism to reduce computation to be done per node, also decrease the time needed for the halo update. Therefore, within the scope of the parameters probed in this investigation, the performance of the halo update is not a limiting factor for increasing distributed memory parallelization.

5 Use of artificial intelligence

During coding ChatGPT, Phind and Github Copilot (running on the ETHZ student licence) have been used as a help for syntax, structuring, understanding and debugging. In the case of C++, a programming language with which both authors did not have a great deal of experience with, AI was particularily helpful in debugging problems corresponding to variable scoping and memory management. Every code snippet received by prompting an AI has been checked for correctness line by line. In the vast majority of cases, further manual adaptations were necessary for the code to work as intended for the purpose of this project.

References

[1] Oliver Fuhrer. Lecture notes for high perfmormance computing for weather and climate models, 2024.

[2] Sorush Khajepor. A c++ mpi code for 2d arbitrary-thickness halo exchange with sparse blocks, 2022. Accessed: 2024-08-28.