Master thesis

Automatic domain decomposition for HPC stencil codes in heterogeneous systems

Lukas Strebel

lstrebel@ethz.ch

Supervisor

Prof. Thomas Schulthess

Dr. Lucas Benedicic

Swiss National Supercomputing Centre (CSCS)

Swiss Federal Institute of Technology Zürich (ETH)

October 1, 2018

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Acknowledgements

I would like to thank CSCS and ETH for providing the opportunity to take on this thesis project. I would also like to thank Prof. Thomas Schulthess for supervising my Master thesis.

Also thanks to my other supervisor Lucas Benedicic for guiding me throughout the whole thesis.

Additionally, thanks to Felipe Cruz for helping with various problems.

As well as a special thanks to Stefano Ubbiali for not only providing the reference codes used throughout the thesis, but also for helping with any questions I had.

Contents

1 Introduction

1.1 Problem statement

The overarching problem addressed in this thesis is the decomposition of the computational domain for stencil based simulations. Domain decomposition is necessary to distribute and balance the computational load among distributed processing units. Additionally, including heterogeneous processing units adds further complications to the domain decomposition problem.

Solutions to the domain decomposition problem should be automatic, i.e. require a minimal amount of manual input. Automating domain decomposition helps separating concerns between domain scientists and computer scientists.

Therefore, the thesis focuses on the creation of an automatic domain decomposition library addressing these problems.

1.2 Background

Most scientific or engineering codes that model and simulate natural phenomena require more computational resources than a single processing unit can provide. Distributed systems, such as high performance clusters, are a common way to provide the necessary resources for complex simulations. However, porting such codes from their usually serial form to codes that run in parallel on distributed systems creates several challenges. These challenges are generally outside the scope of the domain specific developers.

Decomposing the computational domain and distributing it to the processing units is one of these challenges.

1.2.1 Distributed heterogeneous systems

Distributed systems have been used for decades in high performance computing (HPC) to compute simulations for various scientific fields. A more recent development is the introduction of accelerator devices e.g. graphics processing units (GPU) to the nodes in HPC clusters. These accelerator devices bring enormous computational power with their large number of cores for throughput-critical computations. But their architecture is different compared to CPUs for latency-critical applications.

Therefore, some HPC systems have a hybrid configuration of CPUs and GPUs to combine the advantages of both architectures. However, this also means that nodes in HPC clusters no longer have homogeneous processing units, which makes the distribution of the computational domain ever more complex. Mittal and Vetter [2015] provide a more detailed description of the evolution and motivation behind heterogeneous computing.

1.2.2 Stencil codes

Domain decomposition is used in various type of simulations. This thesis focuses on domain decomposition for stencil codes.

Stencil codes are a family of codes commonly used in various scientific computing applications. The name "stencil codes" refers to computations using a fixed pattern of values from points on a mesh.

These fixed patterns often originate from explicit finite difference approximations on structured meshes. Structured meshes guarantee that every point in the mesh, except points on the boundary, has the same number of neighboring points. Explicit finite difference approximations use the values on the mesh at a given point in time to compute the values for the next time step. Finite difference approximations are one common way to solve partial differential equations (PDEs). Numerous scientific fields - such as fluid dynamics or climate dynamics - model the natural phenomena of their studies as PDEs.

A single time step in a stencil code consists of iterating through every point on the mesh according to the fixed pattern, gathering values from other points, and using these values to compute the value for the next time step. Simulations of complex natural phenomena usually require very fine spatial and temporal resolutions as well as computations involving multiple different quantities on every grid point. Therefore, even theoretically simple stencil codes are in practice frequently more intricate.

Complex stencil codes usually require their large mesh to be decomposed into computational sub-domains for each processing unit to be solved in a reasonable amount of time. Therefore, domain decomposition is an important part of stencil codes.

1.2.3 Domain decomposition / graph partitioning

Domain decomposition can be seen as a specific form of load balancing for codes in which the load is roughly equal for each point in a domain. Stencil codes fulfill this characteristic.

The sub-domains that result from domain decomposition for stencil codes need to synchronize the values across their boundaries to be consistent. This synchronization requires communication between the different processing units.

The quality of a given domain decomposition is determined by the balance of computational load among all processors and the reduction of communication among different processors.

A common way to model and solve domain decomposition is through graph partitioning. When using this strategy, the elements of the discrete domain are represented by the vertices of a graph, with edges representing the connections between neighboring elements. The sum of all values on the edges between two sub-domains is called the edge cut. The edge cut in this model represents the total amount of communication between the two sub-domains.

Therefore, graph partitioning algorithms that use a cost function to minimize such an edge cut, while maintaining an approximate uniformity of sub-domain size, provide domain decompositions that minimize the communication cost.

1.3 Related works

In literature, the process of creating sub-domains from a larger domain is referred to by different names: domain decomposition, static mapping, (graph) partitioning, topology mapping, or sometimes more general as a form of load balancing.

Domain decomposition is used to account for different computational load imbalances in various simulations. Common sources for such imbalances are irregular grids (e.g. finite element method), task-based applications, or heterogeneous communication costs. In Saxena et al. [2015] it is stated that heterogeneous architectures have not received as much attention as a source of imbalance. Nevertheless, heterogeneous architectures are a source of imbalance even for simulations that do not have any other source of imbalance.

However, even for different sources of imbalance, most often the problem is modeled as a graph, and a graph partitioning algorithm is used to solve for a domain decomposition.

In the general case, graph partitioning has been proven to be NP-complete (Feldmann and Foschini [2015]). However, relaxing the constraint to roughly equal instead of exactly equal partitions allows various different heuristics to generate a sub-optimal solution in reasonable time.

A general overview and comparison between different graph partitioning algorithms can be found in Karypis and Kumar [1998a]. Specifically, Table 9 illustrates different criteria for useful graph partitioning and how each algorithm ranks in them. Even though this paper is a few years old, both of the most common graph partitioning libraries, METIS [1] and SCOTCH [2], are based on the basic algorithms described in the paper.

While conceptually very similar the algorithms in these libraries differ in their mathematical description and their modeling approach.

1.3.1 SCOTCH library

The SCOTCH library models the problem as a set of two graphs: a valuated, undirected source graph $S(V, E)$ representing the parallel processes (vertices) and communication channels (edges). And a not-valuated target graph $T(V, E)$ representing the topology of the target machine. The target graph is not valuated because it assumes an homogeneous architecture.

To solve the graph partitioning on these two graphs, the SCOTCH library employs a dual recursive bipartitioning algorithm. This algorithm is formulated by Pellegrini [1994] in the following way:

[1] http://glaros.dtc.umn.edu/gkhome/views/metis Accessed: 25.9.18
[2] https://gforge.inria.fr/projects/scotch/ Accessed: 25.9.18

The goal of a mapping is then defined as: $\varphi : V(S) \to V(T)$ such that $\varphi(v_s) = v_t$ if v_s is mapped onto processor v_t.
And $\psi : E(S) \to E(T)$ such that $\psi(e_S) = \{e_T^1, e_T^2, \ldots, e_T^n\}$ with $|\psi(e_S)|$ as the number of edges in the target graph that are used.
With these definitions, the cost function to be minimized is defined as the edge cut function f_c:

$$f_c(\varphi, \psi) = \sum_{e_s \in \text{cut}(E(S))} (c(e_S) \cdot |\psi(e_S)|) \tag{1}$$

where $c(e_S)$ is the value of the edge (i.e. the communication cost) and cut $(E(S))$ contains all edges belonging to the two subsets of the bipartition for which the cost function is computed.

1.3.2 METIS library

The older METIS library was developed on the basis of a multilevel bipartitioning algorithm. In their model, a single valuated, undirected source graph is partitioned, assuming that the target architecture has uniform cost for any communication between two processes. However, the graph partitioning algorithm in the METIS library has been expanded, as explained in Karypis and Kumar [1998b], to allow multiple constraints to be balanced.

1.4 Source graph generation for stencil codes

For graph partitioning to be used for domain decomposition, a source graph representing the domain is required.

Simple one and two dimensional stencils are often either visualized as points connected with lines or written in matrix form. For example the well known stencil used in the computation of the Laplace equation can be expressed as $\begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix}$, which is equivalent to writing the formula as $u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}$.

The matrix form also illustrates the communication pattern needed by the stencil. The position in the middle of the matrix represents the grid point itself. The other non-zero entries in the matrix represent the other grid points needed in the computation.

The computational cost can be approximated by the number of non-zero entries in the matrix e.g. the Laplace stencil requires 4 additions and one multiplication with a scalar constant.

A source graph to partition a given domain could be built by creating a vertex in the source graph for every grid point in the domain. In such a source graph, each vertex would have edges based on the communication patterns and a vertex weight based computational cost approximations of a stencil as previously described. However, even for simple stencils, like the one used in the Laplace equation, it is intuitive but unfeasible to have a vertex for each mesh vertex for meshes with several millions of elements due to the size of such a source graph.

Thus we will investigate models and methods to achieve the generation of a practical source graph to partition the domain.

1.5 Context

Domain decomposition is not the only concern that separates scientific models from well performing HPC code.

Complex scientific codes, like numerical weather prediction and climate simulation models, have grown over years to multiple hundred thousands lines of code written by a large community of scientific developers. Maintaining such large codes for the ever-changing HPC environment is challenging.

The GridTools library provides a domain specific language (DSL) designed to reduce this effort for stencil codes. The main method of the GridTools DSL is to separate the concerns of the scientific model developers (domain experts) from the concerns of the computer scientists (HPC experts). Separation is accomplished by letting the domain experts use the DSL in their model development process without the need to think about performance. At the same time, computer scientist no longer need to understand the complete details of the model but instead can focus on optimizing the core computations used in stencil codes for different programming models and architectures as separate back-end of the GridTools DSL.

The GridTools DSL is written in C++, while domain scientist are often more familiar with higher level languages or simple Fortran as a language close to the mathematical formulations of their models. An additional DSL for Python (GT4Py) aims to overcome this programming language barrier for domain scientist. GT4Py adds an additional abstraction layer on top of the GridTools DSL so that domain scientists can use the simple and expressive Python syntax for model development without losing the advantages of the low-level optimizations of the GridTools back-end.

Within this context, the focus of this thesis is on the automation of the domain decomposition. The resulting automatic domain decomposition library will be part of the larger effort to separate HPC concerns from scientific modeling concerns.

1.6 Implementation and libraries

The focus of this thesis is the implementation of an automatic domain decomposition library to solve the problems described in the previous sections. The next few sections will describe some implementation aspects and existing libraries that will be used.

1.6.1 Implementation details

The implementation language will be Python. Python provides simple and expressive syntax and is increasingly used in the scientific community. Also, in the context of GT4Py, Python is the natural choice for the automatic domain decomposition library.

Standard practices for software development and testing will be employed. This includes unit tests, continuous integration (CI), version control system, code documentation, and adhering to standard style guide conventions.

1.6.2 Communication library

The communication between processors is a significant part of computations involving domain decomposition. Using a high quality communication library is therefore important. Message Passing Interface (MPI) is the leading standard for distributed systems. The communication model of MPI matches very well with finite difference computations and has been used in various finite difference implementations.

In the context of Python, this thesis will make use of the MPI for Python (MPI4Py) library, which provides bindings for MPI in Python. MPI4Py is open source, and as described in Dalcín et al. [2005] supports the most important MPI communications, namely point-to-point, as well as collective communications of general Python objects.

Furthermore MPI4Py supports one-sided MPI. One-sided MPI is the remote memory access (RMA) communication model of MPI. One-sided MPI allows processes to expose part of their memory to other processes, in order for them to directly write or read from this memory. In contrast to the two-sided communication model, this allows the decoupling of data movement and process synchronization.

Since one-sided communication models are a new trend in HPC, this thesis will explore the use of one-sided MPI in the context of domain decomposition and stencil computations.

1.6.3 Graph partitioning library

Solving the graph partitioning problem to determine a domain decomposition is a fundamental part of the proposed work. However, the focus of this thesis is automating the process and modeling heterogeneous systems. Therefore the automatic domain decomposition library will use another library to solve the underlying graph partitioning problem.

The reason for building on top of existing libraries is two-fold. Firstly, it saves time from implementing methods to solve the underlying basic graph partitioning problem. Secondly, building on top of popular libraries can make it easier in the future to keep up to date with the latest graph partitioning methods.

As mentioned in the state of the art review section 1.3 the two most popular graph partitioning libraries at the moment are METIS and SCOTCH.

1.7 Structure of the thesis

The remainder of this thesis is split into three additional sections. In the next section the domain decomposition approach is described and basic model for the source graph generation is explained in detail.

The section after that documents the design, implementation, and user facing functionality of the domain decomposition library developed during this thesis.

The last section showcases the domain decomposition in two case studies used to validate the implemenetation and experiment with the performance and scalability of the domain decomposition library.

2 Domain decomposition

Domain decomposition in the context of this thesis refers to the process of dividing a computational domain into smaller parts in order to run on multiple processing units. Importantly, domain decomposition in this context is not be confused with the *domain decomposition method* in mathematics and numerical analysis. The *domain decomposition method* is used to solve boundary value problems by splitting them into smaller parts. Instead, domain decomposition in high performance computing and this thesis is the name of the process used to prepare a domain for distribution to multiple processing units. Specifically, domain decomposition in this thesis is used for solving partial differential equations using finite difference methods on multiple processing units.

The next sections outline the general approach to domain decomposition, the specifics of the model developed and used in this thesis, an overview of the graph partitioning method, and an example for the subdivision model and graph partitioning for domain decomposition.

2.1 General approach

Some form of domain decomposition is necessary for any distributed computation. The process of decomposing a domain varies widely depending on the problem that is distributed i.e. the domain.

Domain decomposition approaches overlap in some aspects with load balancing methods. For example, many approaches for both use over-decomposition.

Over-decomposition is a two step process. First, it means splitting the problem into more parts than needed at the end. Then in a second step combining these smaller parts into the final decomposition.

For domain decomposition this means that there needs to be a model to describe how the domain should be split into small parts. As well as a model for distributing the smaller parts to the final domain decomposition. The distribution of the smaller parts to the processing units handles the load balancing aspect of domain decomposition by minimizing a set of cost factors.

For domain decomposition the two main cost factors are distributing the same number of grid points to each processing i.e. **computational cost** and minimizing the amount of communication between different processing units i.e. **communication cost**.

Given these over-decomposed domain parts and their corresponding cost factors a graph partitioning algorithm can be applied to get a balanced distribution for each processing unit.

This two step process describes the general approach to domain decomposition and is described in more detail as pseudo-code in Algorithm 1.

To note in Algorithm 1 is that the over-decomposition method introduces one new parameter: the number of subdivisions per dimension. This parameter is further explained

in Section 2.2.

Input: Computational Domain, Number of Processing Units, Stencil Information
Output: Decomposed Domain, Partitioning for each Processing Unit

```
1  Datastructure DomainSubdivision
2  |  Stores (id, size, boundaries, gridpoints, neighbors)
3  Function DomainDecomposition(Domain, NumProcs, StencilExtent)
4  |  /* Step 1: Over-decompose domain and store subdivisions in
   |     adjacency list.                                          */
5  |  SubdivisionWeightedAdjacencyList, ListOfDomainSubdivisions ←
   |   SubdivideDomain(Domain(SizePerDim, PeriodicityPerDim),
   |   NumSubdivPerDim, StencilExtent)
6  |  /* Step 2: Partition subdivisions based on adjacency list.  */
7  |  PartitionList ← GraphPartitioning(SubdivisionWeightedAdjacencyList,
   |   NumProcs)
8  Function SubdivideDomain(SizePerDim, PeriodicityPerDim, NumSubdivPerDim,
   StencilExtent)
9  |  /* Calculate derived information:                           */
10 |  TotalNumberOfSubdivisions ← NumSubdivPerDim
11 |  SizeOfSubdivision ← Domain, NumSubdivPerDim
12 |  SizeOfBoundaries ← SizeOfSubdivision, StencilExtent
13 |  /* Create adjacency list:                                   */
14 |  foreach Subdivision in TotalNumberOfSubdivisions do
15 |  |  foreach Direction do
16 |  |  |  Neighbors ←
   |  |  |   HandleDomainBoundaries(NumSubdivPerDim, PeriodicityPerDim)
17 |  |  end
18 |  |  AppendToWeightedAdjacencyList ← Neighbors, SizeOfSubdivision,
   |  |   SizeOfBoundaries
19 |  |  StoreSubdivisionInformation(id, size, boundaries, gridpoints,
   |  |   neighbors)
20 |  end
21 |  return SubdivisionWeightedAdjacencyList, ListOfDomainSubdivisions
22 Function GraphPartitioning(SubdivisionWeightedAdjacencyList, NumProcs)
23 |  if FileOutputIsEnabled then
24 |  |  WriteAdjacancyListToFile(SubdivisionWeightedAdjacencyList,
   |  |   FormatFlag)
25 |  end
26 |  CallGraphPartitioningLibrary(SubdivisionWeightedAdjacencyList, NumProcs)
27 |  return Partitioning
```

Algorithm 1: Pseudo-code description of the two-step process to decompose a domain.

2.2 Subdivision model for source graph generation

As mentioned before, source graphs where every source graph node represents a single physical domain grid point would be impractical for large domains. Therefore, any alternative has to combine some of these domain grid points into separate subdivisions even before the domain decomposition occurs.

Any model for over-decomposition of domains should be guided by a few principles described in detail in Table 1.

Criteria	Explanation
Uniformity and regularity.	One of the advantages of regular grids for finite difference is simplicity of used data structures and the performance benefits these include. Modeled domain subdivisions should keep this simplicity as well.
Clearly defined and easy to determine boundaries.	The boundaries between these separate subdivisions determine what and how the partitions will communicate with each other. Creating clear and simple boundaries is necessary to make the communication process simple and as fast as possible. The partitioning process will add complexity to the boundary communication, therefore any complexity from the over-decomposition process should be avoided.
Uniform number and direction of neighbors.	Even though graph partitioning can decompose graphs with vertices of various degrees, for the simplicity and performance of the communication aspect introduced by a domain decomposition the number of neighboring subdivisions should be uniform. Subdivisions at non-periodic boundaries of the physical domain are an unavoidable exception to this criteria.
Simple calculations of cost functions.	The overall goal of providing good domain partitions can only be achieved if the graph partitioning algorithm receives good estimates for the communication and computational cost for each subdivision. Therefore, any subdivision model needs simple but accurate estimates for both of these cost functions.

Table 1: Criteria for over-decomposition models used to create subdivisions of a regular domain.

The simplest method of subdividing a regular grid is by splitting it across each dimension separately, i.e. creating a number of uniform subdivisions. This guarantees that every subdivision has the same amount of neighbors, with the exception of sections at the boundary of the physical domain if the boundary is not periodic. Also every

subdivision has the same size and therefore distributing them equally during the graph partitioning also means distributing the grid points equally.

For this kind of subdivisions it is simple to define the interior and boundary between subdivisions, as visualized for a two-dimensional and three-dimensional example in Fig. 1 and Fig. 2 respectively. This distinction is important for the calculations of the communication and computational cost as described in Sections 2.2.1 and 2.2.2.

One limit to the minimal size of a subdivision in any direction is given by the corresponding stencil extent. Meaning a stencil can not extend into multiple subdivisions, but only the direct neighbor, otherwise subdivision would have multiple neighbors in the same direction. This should not be a problem, since usually stencils are not as large as any reasonable sized subdivision.

An additional benefit from such a simple subdivision approach is that only one more parameter has to be determined i.e. the number of subdivision per dimension.

Having more subdivisions per dimension gives the graph partitioning finer control over edge cut minimization and vertex weight balance. However, this finer control comes at the cost of more complex boundaries. Complex boundaries meaning boundaries where more subdivisions have to communicate with each other than for more straightforward boundaries, since more communication does not necessarily mean higher communication cost for these simple estimates.

Some of these effects this parameter choice can have on the domain decomposition are mentioned in Section 2.4 and shown in Fig. 5.

2.2.1 Communication cost

The communication cost between subdivisions is the main criteria being minimized in the graph partitioning algorithm. Splitting the domain into uniform subdivisions keeps the communication cost computation simple.

The communication cost for each neighbor of the subdivision is determined by the size of the boundary between the two and the extent of the stencil in the corresponding direction.

For three dimensional domains the communication cost can therefore be computed from the following formula:

$$cc(i) = sds\left(\left(\left\lfloor\frac{i}{2}\right\rfloor - 1\right)\%3\right) \cdot sds\left(\left(\left\lfloor\frac{i}{2}\right\rfloor + 1\right)\%3\right) \cdot se(i), \tag{2}$$

where $cc(i)$ is the communication cost for side i. The index $i \in [0, 1, 2, 3, 4, 5]$ is combination of side and dimension, i.e. 0 is negative x-direction, 1 is positive x-direction, 2 is negative y-direction, and so on. $sds(j)$ is the subdivision size of dimension $j \in [0, 1, 2]$. Lastly $se(i)$ is the stencil extend of side and dimension i. The division, rounding and modulo in the formula is simply a short version of selecting the index of the other two

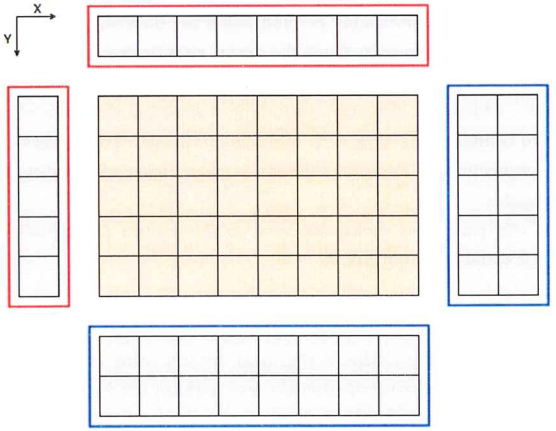
dimensions for the computation of the area between two subdivisions.

For example, the communication cost for the subdivision shown in Fig. 1a is $cc(0) = 5, cc(1) = 10, cc(2) = 8, cc(3) = 16, cc(4) = cc(5) = 0$ i.e. the same as the number of squares shown in the corresponding halo regions.

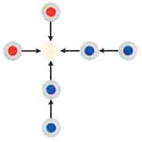
2.2.2 Computational cost

The computational cost, corresponding also to the vertex weight in the source graph, is even easier to compute, since it is proportional to the number of grid points inside the subdivision.

For example, the computational cost for the subdivision shown in fig. 1a is 40 i.e. the same as the number of squares shown as the grid points inside the subdivision.



(a) In the middle and in light red the grid points belonging to this subdivision. Outside in gray the four halo regions. The red and blue lines surrounding the halo region indicates negative and positive directions respectively.



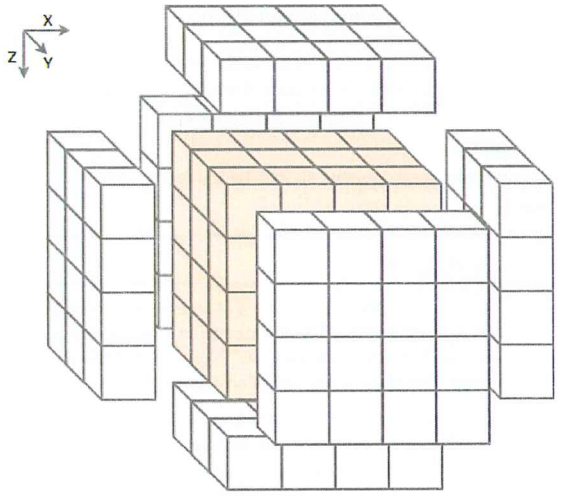
(b) Stencil representation corresponding to the subdivision on the left panel.

Figure 1: Schematic view of a single two-dimensional domain subdivision.

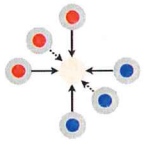
2.2.3 Source graph generation

With the subdivision model definition and the two cost functions defined the source graph can be easily generated. As shown in lines 14 to 20 of Algorithm 1 the source graph is generated as a weighted adjacency list by iterating over all subdivisions to determine their neighbors and handle the physical domain boundaries.

Determining the neighbors is an uncomplicated index calculation, as shown in Eq.(3) and handling the physical domain boundaries only requires catching the corresponding cases during the index calculation.



(a) In the middle and in light red the grid points belonging to this subdivision. Outside in gray the six halo regions.



(b) Stencil representation corresponding to the subdivision in figure 2a.

Figure 2: Schematic view of a single three-dimensional domain subdivision.

$$neighborindex = ((i + x) \cdot ndy + (j + y)) \cdot ndz + (k + z) \ , \tag{3}$$

where i, j, k are the index number of the current subdivision in each dimension, and x, y, z are either -1 for the negative neighbor in the corresponding direction, $+1$ for the positive neighbor in the direction, and 0 otherwise. Also, ndy, ndz are the number of subdivisions in y-direction and z-direction respectively.

The weighted adjacency list consists of four arrays. The array *vweights* contains the vertex weight for each subdivision i.e. vertex, thus its length is equal to the total number of subdivisions. The array *eweights* similarly contains the edge weight for each edge, and is therefore twice as long as the number of edges. The array *adjncy* contains the index of the end vertex for each edge, and is therefore twice as long as the number of edges. The order in *adjcny* determines the starting vertex for each edge. It starts with the edges of the vertex with index 0, after all edges of vertex 0 it continues with edges of vertex 1 and so on. And lastly the array *xadj* contains for each vertex the *adjncy*-index for the first edge that does not belong to the vertex. The array *xadj* has the length of the total number of subdivisions. i.e., VERTICES

The two arrays containing information about the edges are twice as long as the number of edges since graph partitioning methods do not require undirected graphs, even though

for domain decomposition only undirected graphs are used.

An example adjacency list in this format can be seen in fig. 3, this adjacency list is the one corresponding to the source graph used in fig. 4.

vweights

200	200	200	200	200	200

eweights

20	10	20	20	10	20	10	10	20	10	20	20	10	20

adjncy

1	3	0	2	4	1	5	0	4	1	3	5	2	4

xadj

2	5	7	9	12	14

Figure 3: Weighted adjacency list arrays for source graph in the left panel.

2.3 Load balancing - graph partitioning method

Once a domain is over-decomposed and the model has used its communication and computational cost estimates to generate a source graph, the graph partitioning algorithm can be used to distribute the subdivisions into one partition for each processing unit.

Graph partitioning on such a weighted source graph means cutting the graph into partitions such that the edge cut between all partitions is minimized (i.e. minimal communication cost) while also keeping the vertex weights balanced (e.g. balanced computational cost).

More details about the graph partitioning method can be found in Karypis and Kumar [1998b] and are not the focus of this thesis.

The output of the graph partitioning methods is a list of indices for each vertex i.e. subdivision the index corresponds to a partition i.e. the index of a processing unit.

A small example to demonstrate a few of the key characteristics of graph partitioning is shown in Fig. 4. Specifically, Fig. 4 shows three possible partitions for an example source graph. For visual simplicity a two-dimensional domain is used, however a three-dimensional domain would work in the same way. The source graph corresponds to a domain of size 30×40 with three subdivisions in x-direction and two subdivisions in y-direction. In this example these six subdivisions are partitioned into two partitions, represented by the color of the circle in the Fig. 4a, Fig. 4b, and Fig. 4c.

Figure 4a shows a case where a graph partitioning cut causes a computational imbalance between the partitions. Partitioning one subdivision more to one partition in such

a small example causes a very large imbalance. Usually the graph partitioning method would not allow a result with this much imbalance. However, in larger graphs some computational imbalance up to a threshold is very common. Most graph partitioning methods allow manipulating the imbalance threshold.

The mixed direction cut shown in Fig. 4b has the largest communication cost of the three partitions shown here. However, it is a fully valid partition and partition boundaries that look similar are common in partitioning of larger graphs. See the boundary between the brown and red partition in Fig. 5a for example. This should highlight the importance of having defined clear and simple defined boundaries in the subdivision model, since the graph partitioning model already adds complexity to the partition boundaries.

Finally, Fig. 4c shows the ideal case for this small example. Ideal cuts like this are only likely in such small and simple cases.

2.4 Example subdivision and domain decomposition

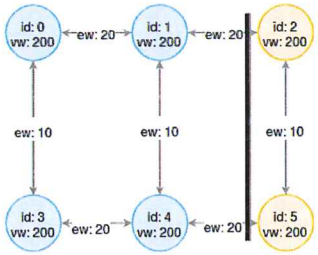
The following example should highlight some aspects of the discussed subdivision and domain decomposition.

All the following figures in this section use the same domain and stencil. The domain has size $2048 \times 1024 \times 40$ and the stencil extents are given as $[1, 1, 1, 1, 0, 0]$. This means the stencil uses 1 neighboring grid point in negative and positive x- and y-direction. The boundary of the domain is also periodic in x-direction and non-periodic in y- and z-direction.

Figures 5a, 5c, and 5e show the results if the domain is split into $16 \times 8 \times 1$ subdivisions. This means every box in the figures represents $128 \times 128 \times 40$ grid points.

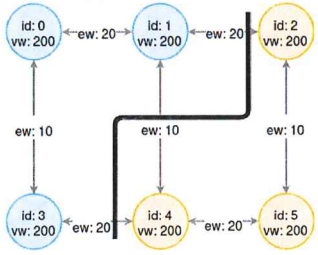
For comparison Figs. 5b, 5d, and 5f show the results if the domain is split into $32 \times 16 \times 1$ subdivisions. This means every box in the figures represents $32 \times 32 \times 40$ grid points and 4 of these boxes correspond to one box in the figures on the left side.

This comparison should highlight the difference the number of subdivisions and their size can make for the overall domain decomposition. Also the difference the graph partitioning can make is shown in Fig. 5.



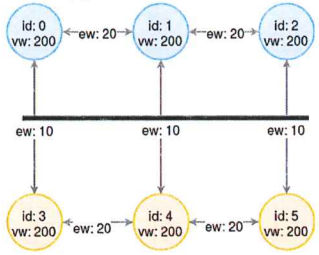
Communication Cost: 40
Computational Imbalance: 400

(a) X-direction cut.



Communication Cost: 50
Computational Imbalance: 0

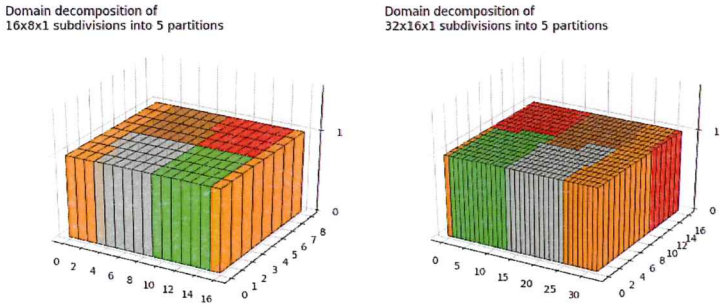
(b) Mixed direction cut.



Communication Cost: 30
Computational Imbalance: 0

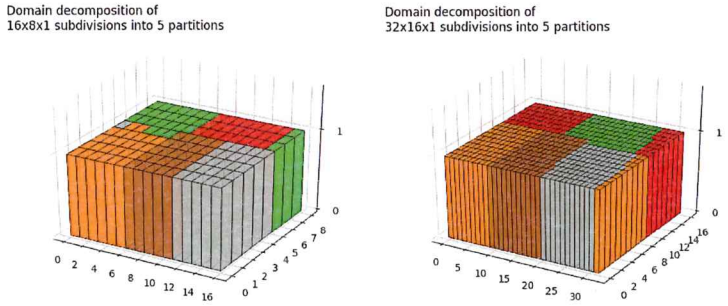
(c) Y-direction cut.

Figure 4: Visual representation for a small graph partitioning example. Each circle represents one subdivision of size 10×20. The label "id" is the subdivision identification number. The label "vw" describes the vertex weight and is equal to the size of the subdivision. The label "ew" describes the edge weight and is equal to the size of the boundary between the corresponding subdivisions. The color of the circle depicts the corresponding partition. The thick, black line shows the boundary between the two partitions i.e. the cut chosen by the graph partitioning.



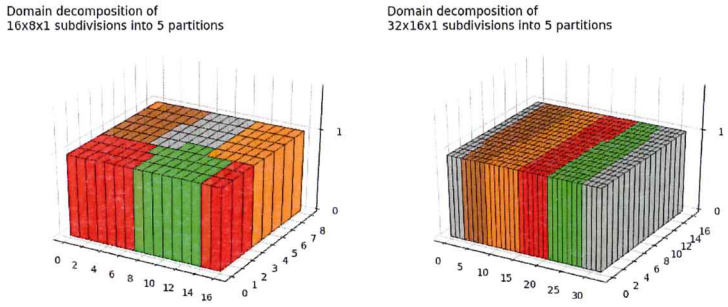
(a) Partitioned by Metis.

(b) Partitioned by Metis.



(c) Partitioned by PyMetis

(d) Partitioned by PyMetis



(e) Partitioned by Scotch

(f) Partitioned by Scotch

Figure 5: Domain subdivision and domain decomposition into 5 partitions. Each box represents one subdivision of grid points. Each color represents one partition. The domain has size 2048 x 1024 x 40 and a periodic boundary in x-direction. Even though the domain is fully three dimensional, the domain is only decomposed in the x-y plane i.e. the horizontal plane. Such a decomposition represents an often encountered use case for domains where the size of the two horizontal dimensions are much larger than the size of the vertical dimension e.g. numerical weather prediction models. Not decomposing in the vertical dimension is important for locality and performance since the vertical dimension is usually the innermost loop dimension. This also fits with many parameterizations that use the single column model assumption.

3 Library design and implementation

The previous chapter focused on the domain decomposition method and how to model the underlying source graph for stencil codes. However, domain decomposition is only the initial part of this automated domain decomposition library for stencil codes. Once the domain is decomposed additional functionalities need to be provided to make a stencil code run on a decomposed domain.

Conceptually the library can be separated into three distinct parts: Pre-process, runtime, and post-process. The flowchart in Fig. 6 gives an abstract overview of the whole automatic domain decomposition process.

The next sections will first outline the design and functionality of each of these three parts and then show and explain the user facing functions in more detail with examples.

3.1 Design

3.1.1 Pre-process

The pre-process contains functionalities needed before the main, distributed stencil computations are performed. Specifically, the actual domain decomposition happens during the pre-process. Therefore the pre-process contains functions to prepare the graph partitioning files, perform the actual graph partitioning, and store the results.

The design and implementation of the pre-process functions follows the descriptions of the source graph generation and graph partitioning from the previous chapter in the Sections 2.2.1, 2.2.2, and 2.2.3 and the pseudo-code of Algorithm 1 very closely and is therefore not repeated here. The user-facing part of the pre-process implementation is outlined with examples in Section 3.2.1.

3.1.2 Runtime process

During the runtime of a stencil computation the domain decomposition library has to manage a few tasks. Most of them arise because once the domain is decomposed into smaller parts each of these parts needs to be managed separately. At the same time the goal for the user facing parts of the library is to avoid adding complexity. To achieve both these goals the runtime functions of the domain decomposition library are organized into two main classes: The user facing `DomainDecomposition` class and the internal `DomainSubdivision` class.

The `DomainDecomposition` class holds a list of all subdivisions i.e. `DomainSubdivision` objects belonging to the partition of the corresponding MPI process. Specifically, in the initialization of the `DomainDecomposition` class each MPI process loads the full serialized list of subdivisions from the pre-process and removes all subdivisions given to a different partition by the graph partitioning of the pre-process.

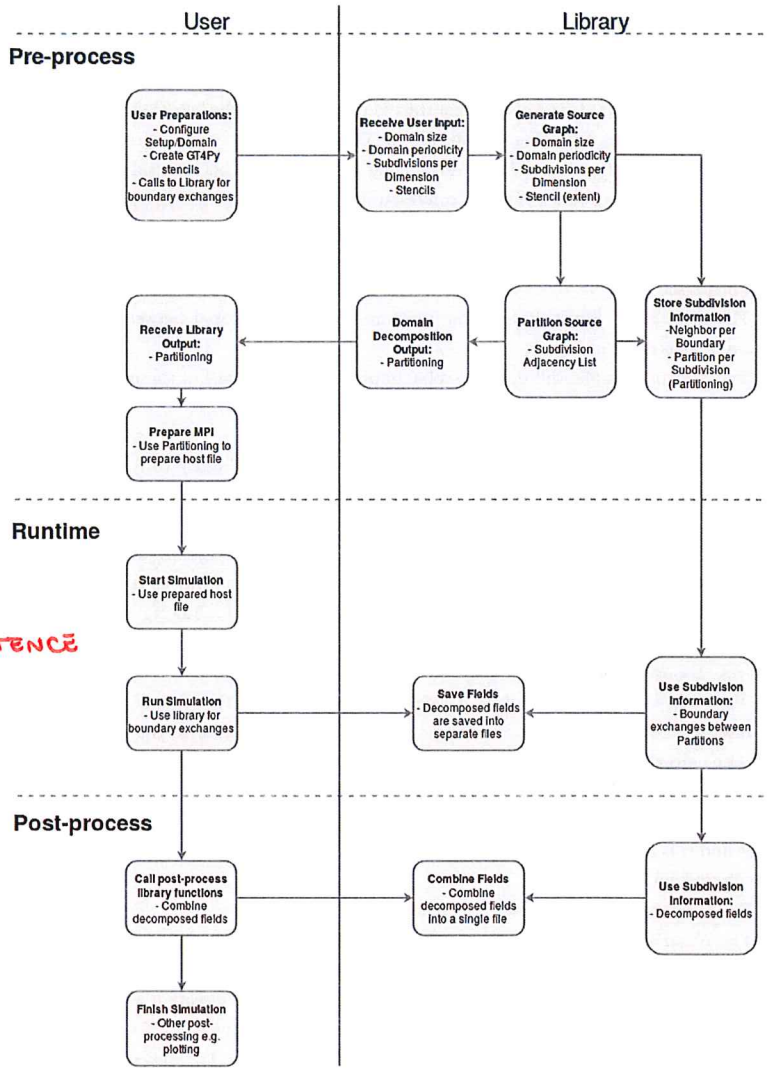


Figure 6: Schematic description of the flow for the automatic domain decomposition process.

In general the purpose of the `DomainDecomposition` is mostly to hide the separate subdivisions from the user and give the user a single point of contact with the library functions. Internally the `DomainDecomposition` contains functions that mostly just delegate function calls to all subdivisions.

The `DomainSubdivision` class in contrast is responsible for the actual stencil computation. This includes managing and storing the fields in NumPy arrays, carrying out the GridTools4Py stencils computations, exchanging the boundaries between subdivisions, as well as applying the global boundary condition in the appropriate subdivisions.

The fields and stencils need to be managed in the `DomainSubdivision` class because each subdivision needs its own fields and each GridTools4Py stencil has to be instantiated with a specific field.

The hierarchical structure and the functions that are delegated between these two classes is illustrated in Fig. 7.

The next paragraphs will describe some implementation details of the main runtime functionalities.

Field registration replaces for the user the usual call to NumPy to instantiate an array for a field. The subdivisions store fields as a dictionary containing the name of the field as the key and the array itself as the value. The name of the field needs to be stored alongside the array because of the indirection introduced by the domain decomposition. The indirection here is that the user wants to operate on a single array as the field while the decomposed domain consists of many arrays for each field. Storing the name of the field in addition to the NumPy arrays is a necessary compromise between these two aspects. Basically, this addition allows the user to refer to a field by its name in many functions where in the original code it would return to the NumPy array of the field.

Internally the field registration delegates the array instantiation to the subdivisions and also calls the internal functions to setup the NumPy views to define the halo region used in the communication function. The creation of these boundary views is done during field registration because this is the first time the subdivisions get the name of a field as input and it is assumed that any field might need to communicate its boundary with the neighboring subdivisions.

The following boundary views are created:

The NumPy view called `recv_slices` stores the indices of the halo region in each direction. The indices of the halo region in a given direction are $0 : hs$ for negative directions where hs is the size of the halo in the corresponding direction and $-hs :$ for positive directions. Note that the various hs in these descriptions can refer to different values if the halo size varies between dimensions and directions. The indices in the two dimensions not corresponding to the direction are both $-hs : hs$. The order of directions for `recv_slices` is negative x-direction, positive x-direction, negative y-direction, positive y-direction, negative z-direction, and positive z-direction.

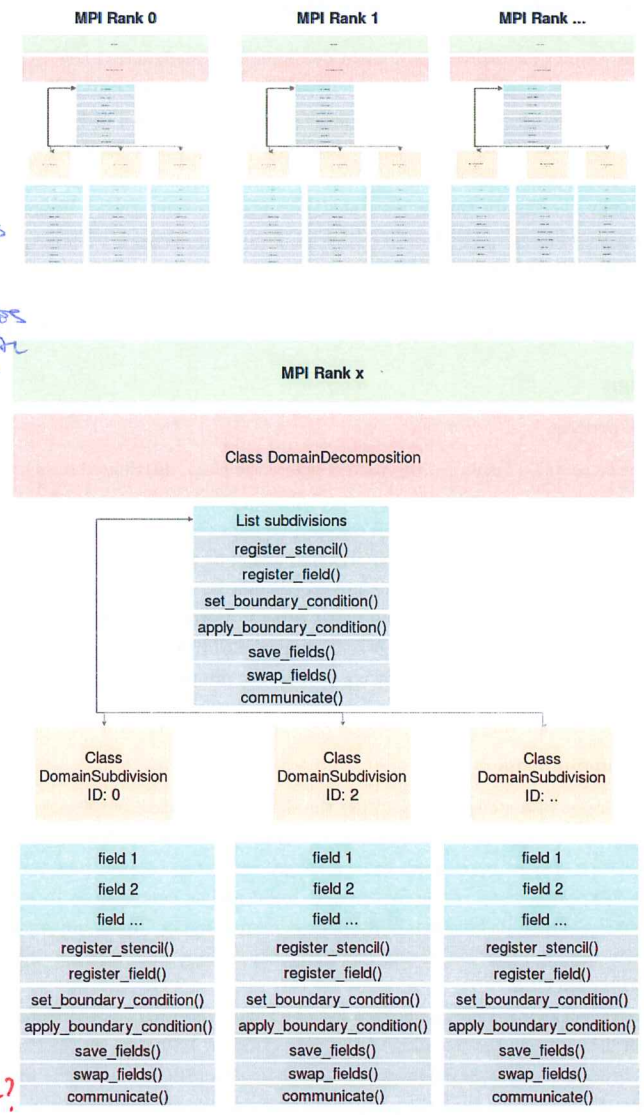


Figure 7: Hierarchical structure of the main components of the runtime classes of the domain decomposition library.

The next NumPy view called `send_slices` stores the indices of the outermost regions neighboring the halo regions. These are the boundary values that are the halo region of the neighboring subdivision. For these send regions the indices in a given direction are $hs : hs + hs$ for negative directions and $-hs - hs : -hs$ for positive directions. Indices in the two non-direction dimensions are again $-hs : hs$. The order of directions for `send_slices` is negative x-direction, positive x-direction, negative y-direction, positive y-direction, negative z-direction, and positive z-direction.

The third NumPy view is called `get_local` and stores the the same indices as `send_slices` in a different order. The order is positive x-direction, negative x-direction, positive y-direction, negative y-direction, positive z-direction, and negative z-direction. This NumPy view is used for the local communication between subdivisions in the same partition. The reason for the difference in order is described in more details in the communication paragraph.

The last NumPy view is called `get_global` and combines the same indices as the last two NumPy views with the global coordinates of the subdivision. This NumPy view is only used if the global boundary condition is provided in a single file and therefore is not feasible for large computational domains.

Stencil registration can be done once all the fields used in the stencil have been registered. Stencil registration involves the same keywords that instantiation of a GridTools4Py stencil would. With the exception of the *domain* keyword which is ignored since the stencil domain depends on the subdivision size and the stencils halo and not user input. However an additional keyword *reductions* allows the user to limit the stencil domain if only a smaller part of the domain should be used.

Otherwise the most obvious difference is that instead of NumPy arrays directly the *inputs* and *outputs* keywords need to be provided with the field names instead since every subdivision has their own NumPy array to give to the GridTools4Py stencil.

To keep the handling of GridTools4Py stencils similar to non-decomposed code the `register_field()` function returns a single object that has a `compute()` function that can be called after instantiation. For the domain decomposition library this object is not the GridTool4Py stencil directly but instead an object of the small internal helper class `DomainDecompositionStencil` which does nothing more than store the subdivisions stencils in a list and provide a `compute()` function to delegate to the subdivisions `compute()` functions.

Communicate functions are needed because the subdivisions of the domain need to exchange their boundaries with each other in between computations. For the user this function needs to be called every time the values of a field need to be up to date i.e. in most cases after or just before the stencil computation. Internally this function encompasses all functions updating the halo region of a field or a list of fields. Specifically,

the `DomainDecomposition.communicate()` function delegates to each `DomainSubdivision.communicate()` function which in turn calls either two-way or one-way communication functions depending on the optional flag. Additionally, both the two-way and one-way communication have to handle neighboring subdivisions being in the same partition by calling a function to handle local communication.

Both the two-way and the one-way communication function are designed to follow the same procedure. Section 3.3 gives more details on the differences for the MPI one-sided approach. The communication procedure iterates over each direction sequentially, but for the two-way communication non-blocking send and receive functions are used so that the communication of the second direction can start before the communication of the first direction is finished.

For each direction the communication function first checks if it is at the global boundary, because the global boundary is handled in its own function. Afterwards, the communication function checks if the neighbor in the given direction belongs to the same partition and if yes calls the local communication function. Otherwise the external communication functions are called. For non-blocking two-way communication the processors with an even MPI rank send first and receive second, while processors with an odd MPI rank receive first and send second. This is done to avoid the situation where two processors both wait to receive from each other.

Local communication is just a copy between `DomainSubdivision` fields. To copy the boundary a NumPy view of the halo region to receive into and a NumPy view to access the boundary are used. The halo region NumPy view is the same for local communication as for external communication. The NumPy view to access on the neighbor differs from the send NumPy view of the external communication in the order of directions.

For example, if the negative x-direction has to be updated from a local neighbor, it is necessary to access the outermost region that is not the halo region in positive x-direction of the neighboring subdivision in negative x-direction. However, when sending the negative x-direction to our neighbor the outermost region that is not the halo region in negative x-direction has to be sent to the neighbor in negative x-direction.

This difference is the result of the external communication function combining the sending and receiving of the boundary in a given direction, while the local communication only needs to handle the receiving.

3.2 User interface

The following subsections go through the additional code and code modifications required to transform a non-distributed GridTools4Py code into a domain decomposed and distributed GridTools4Py code.

3.2.1 Pre-process: Additional setup information.

To start the domain decomposition pre-process the `DomainPreprocess` class needs to be initialized with the size of the computational domain, the periodicity of the global boundary, and the number of subdivisions per dimension. Optionally, the path and prefix arguments can be provided to store all the files under the given path and using the given prefix.

Additionally the domain decomposition method needs as user input the access patterns of the stencils used. The stencil patterns are added by calling the `add_stencil()` function of the `DomainPreprocess` class. The `add_stencil()` needs as input a Python dict with the name of the field as the key and a list of size 6 as the item. The list of size 6 contains the access pattern list for each direction.

The first entry in the list corresponds to the access pattern in the negative x-direction. The second entry corresponds to the positive x-direction. The third entry corresponds to the negative y-direction, then the 4th is the positive y-direction, followed by the negative z-direction and lastly the positive z-direction.

The access patterns themselves are lists of indices that the stencil accesses in that direction. For example the simple two dimensional Laplace stencil would have a list that looks like this: `[[1], [1], [1], [1], [0], [0]]`

After all stencils are added this way the actual domain decomposition can be done by calling the `preprocess()` function of the `DomainPreprocess` class. This function generates the graph to be partitioned and saves the subdivision information in a pickle file.

Lastly, the `pymetis_partitioning(nparts)` function of the `DomainPreprocess` class can be called to generate the partitioning. The input argument `nparts` is the number of partitions that should be generated.

Listing 1 shows a full example for this pre-process user code.

3.2.2 Pre-process: Initialization of fields.

Before the any stencil computation can start the initial values of the fields need to be provided. In simple, serial codes this can be done just before the time stepping. However, for the distributed parallel code it is simpler and better to store the initial values of the fields in files during the pre-process. This is simpler, because all distributed nodes will need access to their parts of the fields. Therefore, if the initial conditions were set during the runtime, either each subdivision would need to initialize its field or a centralized


```
ddc = DomainPreprocess(domain=cdomain, periodic=periodic,
    subdivs_per_dim=slices, path=path, prefix=prefix)

# Add Use case specific stencils:
if method == 'forward_backward':
    ddc.add_stencil({"unow": [[1], [1], [1], [1], [0], [0]]})
    ddc.add_stencil({"vnow": [[1], [1], [1], [1], [0], [0]]})
elif method == 'upwind':
    ddc.add_stencil({"unow": [[1], [1], [1], [1], [0], [0]]})
    ddc.add_stencil({"vnow": [[1], [1], [1], [1], [0], [0]]})
elif method == 'upwind_third_order':
    ddc.add_stencil({"unow": [[2], [2], [2], [2], [0], [0]]})
    ddc.add_stencil({"vnow": [[2], [2], [2], [2], [0], [0]]})

# Once all stencils are added call preprocess and partitioning
ddc.preprocess()
ddc.pymetis_partitioning(nparts)
```

Listing 1: Example code of the domain pre-process function calls and additional information needed to decompose a domain.

initialization would need to be sent to each subdivision. Distributing the initialization to each subdivision can become complicated for initial conditions that depend on their position in the global domain. Initializing centrally and sending the initial values out can become a bottleneck for large domains, if the global domain needs more memory than a single node can provide.

Therefore, at least at the moment the domain decomposition library uses NumPy files generated in the pre-process for initialization of the fields.

A simple example for this field initialization can be viewed in Listing 2.

However, for large grids this will run into the size limitation of NumPy arrays and instead the fields need to be initialized into separate files per subdivision. This can be done by iterating over subdivisions and saving the appropriate portion of the fields into files with a postfix identifying the corresponding subdivision identification number. A detailed example of this is shown in 3.

3.2.3 Main: Loading domain decomposition information.

The main or runtime code needs to be modified in a few places to change a serial code to use the domain decomposition library.

The first change is initializing the `DomainDecomposition` class and loading the subdivision information and domain partitioning generated in the pre-process.

The `DomainDecomposition` class needs the name of the partitioning file and the format of the partitioning (i.e. either "metis" or "scotch") as input arguments. Optionally, the same path and prefix arguments as in the pre-process have to be provided if the files are stored under the given path and using the given prefix.

```
def prepare_initial_condition(nx, ny, nz, dxs, dxe, dys, dye, nb):
        domain = [(dxs, dys), (dxe, dye)]
        datatype = np.float64

        # Create the grid
        x = np.linspace(domain[0][0], domain[1][0], nx - 2 * nb)
        y = np.linspace(domain[0][1], domain[1][1], ny - 2 * nb)

        # Instatiate the arrays representing the solution
        unew = np.zeros((nx, ny, 1), dtype=datatype)
        vnew = np.zeros((nx, ny, 1), dtype=datatype)

        # Set the initial conditions
        for i in range(nb, nx - nb):
            for j in range(nb, ny - nb):
                if (0.5 <= x[i - nb] and x[i - nb] <= 1.0) and (0.5 <=
                    y[j - nb] and y[j - nb] <= 1.0):
                    unew[i, j, 0], vnew[i, j, 0] = 0.0, 1.0
                else:
                    unew[i, j, 0], vnew[i, j, 0] = 1.0, 0.0

        # Apply the boundary conditions
        unew[0, :, 0], vnew[0, :, 0] = 0., 0.
        unew[-1, :, 0], vnew[-1, :, 0] = 0., 0.
        unew[:, 0, 0], vnew[:, 0, 0] = 0., 0.
        unew[:, -1, 0], vnew[:, -1, 0] = 0., 0.

        np.save(path + prefix + "shankar_initial_conditions_unew.npy",
            unew)
        np.save(path + prefix + "shankar_initial_conditions_vnew.npy",
            vnew)
```

Listing 2: Example code of the domain pre-process function to initialize the fields "unew" and "vnew" for the Shankar use case.

BUT IN THIS WAY ALL FILES ARE STORED IN THE SAME NODE → STILL MEMORY LIMITATIONS?


```
def prepare_initial_condition(nx, ny, nz, sx, sy, sz, dxs, dxe, dys,
    dye, nb, eps):
        domain = [(dxs, dys), (dxe, dye)]
        datatype = np.float64

        snx = nx // sx
        sny = ny // sy
        snz = nz // sz

        x_domain = domain[1][0] - domain[0][0]
        dx = float(x_domain) / (nx - 1)
        y_domain = domain[1][1] - domain[0][1]
        dy = float(y_domain) / (ny - 1)

        for isx in range(sx):
            for isy in range(sy):
                for isz in range(sz):
                    ind = (isx * sy + isy) * sz + isz
                    # Create the grid
                    x = np.linspace(isx * snx * dx,
                                    (isx + 1) * snx * dx,
                                    snx)
                    xv = np.repeat(x[:, np.newaxis], sny, axis=1)
                    y = np.linspace(isy * sny * dy,
                                    (isy + 1) * sny * dy,
                                    sny)
                    yv = np.repeat(y[np.newaxis, :], snx, axis=0)

                    # Instatiate the arrays representing the initial
                        conditions
                    unew = np.zeros((snx + 2 * nb,
                                     sny + 2 * nb,
                                     snz), dtype=datatype)
                    vnew = np.zeros((snx + 2 * nb,
                                     sny + 2 * nb,
                                     snz), dtype=datatype)
                    # Set the initial conditions
                    unew[nb:-nb, nb:-nb, 0] = (
                    - 4. * eps * np.pi
                    * np.cos(2 * np.pi * xv) * np.sin(np.pi * yv)
                    / (2. + np.sin(2. * np.pi * xv)
                    * np.sin(np.pi * yv))
                    )
                    vnew[nb:-nb, nb:-nb, 0] = (
                    - 2. * eps * np.pi * np.sin(2 * np.pi * xv)
                    * np.cos(np.pi * yv)
                    / (2. + np.sin(2. * np.pi * xv)
                    * np.sin(np.pi * yv))
                    )

                    np.save(path + prefix +
                    "zhao_initial_conditions_unew_"+str(ind)+".npy",
                        unew)
                    np.save(path + prefix +
                    "zhao_initial_conditions_vnew_"+str(ind)+".npy",
                        vnew)
```

Listing 3: Example code of the domain pre-process function to initialize to fields "unew" and "vnew" into separate files for each subdivision for the Zhao use case.

```
prep_domain = DomainDecomposition("subdomains_pymetis.dat.part.4", "
    metis", path=path, prefix=prefix)
```

Listing 4: Example code for the loading of the domain decomposition information.

The subdivision information file uses the name generated in the pre-process and therefore does not need to be provided by the user.

Listing 4 shows such an initialization call.

3.2.4 Main: Instantiation of fields.

The instantiation of the fields used by the stencil computations needs to be modified for the use with the domain decomposition library.

Usually in the serial code field instantiation corresponds to a single call to `numpy.zeros` before the initial conditions are applied.

For the domain decomposition library the fields need to be registered with call to the `register_field()` function of the `DomainDecomposition` class.

The fields need to be registered in this way because the separate subdivision need to instantiate their own NumPy array of the field instead of a global array for each field.

Registering the fields to the `DomainDecomposition` class requires the name of the field, the extent of the halo of the field, and the initial value file generated in the pre-process.

The extent of the halo is a list of size 6 were each entry corresponds to one direction and the value is the largest offset in that direction. The order is the same as for the stencil access pattern in the pre-process i.e. negative x-direction, positive x-direction, negative y-direction, positive y-direction, negative z-direction, and positive z-direction. But in contrast to the stencil access pattern the field halos only need the maximum of the stencil offsets in each direction and not every access.

The field halo extent here are needed to instantiate fields in the subdivisions in the correct size since they have to contain the interior cells corresponding to the solution as well as the halos to each of their neighbors used in the boundary exchanges.

Listing 5 and Listing 7 show the direct comparison between the original instantiation of four fields and the corresponding registration process needed in the domain decomposition library. This should demonstrate that code needed to be added by the user is kept to a minimum.

In the case where the initial conditions were written in a separate file for each subdivision, an additional, optional keyword `singlefile` has to be set to `False`. Listing 6 shows such a case.

3.2.5 Main: Instantiation of stencils.

A small code modification is also necessary for the instantiation of the stencils. While in serial code the stencils can be directly instantiated by a call to gridtools python module,


```
# Instantiate the arrays representing the solution
unow = np.zeros((nx, ny, 1), dtype = datatype)
unew = np.zeros((nx, ny, 1), dtype = datatype)
vnow = np.zeros((nx, ny, 1), dtype = datatype)
vnew = np.zeros((nx, ny, 1), dtype = datatype)
```

Listing 5: Example code for the original user field instantiation.

```
halo = [1, 1, 1, 1, 0, 0]
prep_domain.register_field(fieldname="unow", halo=halo, field_ic_file=
    path + prefix +"shankar_initial_conditions_unew.npy")

prep_domain.register_field(fieldname="vnow", halo=halo, field_ic_file=
    path + prefix +"shankar_initial_conditions_vnew.npy")

prep_domain.register_field(fieldname="unew", halo=halo, field_ic_file=
    path + prefix +"shankar_initial_conditions_unew.npy")

prep_domain.register_field(fieldname="vnew", halo=halo, field_ic_file=
    path + prefix +"shankar_initial_conditions_vnew.npy")
```

Listing 6: Example code for the same field instantiation using the domain decomposition library.

```
halo = [1, 1, 1, 1, 0, 0]
prep_domain.register_field(fieldname="unow",
                           halo=halo,
                           field_ic_file=path + prefix
                           + "zhao_initial_conditions_unew",
                           singlefile=False)
```

Listing 7: Example code for field instantiation using the domain decomposition library in the case where the initial ~~condtions~~ conditions where written to a file per subdivision.

```
# Instantiate stencil object
stencil = gt.NGStencil(
    definitions_func = definitions_func_,
    inputs = {'in_u': unow, 'in_v': vnow},
    global_inputs = {'dt': dt_, 'dx': dx_, 'dy': dy_, 'eps': eps_},
    outputs = {'out_u': unew, 'out_v': vnew},
    domain = domain_,
    mode = gt.mode.NUMPY
    )
```

Listing 8: Example code for the original user stencil instantiation.

```
# Instantiate stencil object
stencil = prep_domain.register_stencil(
    definitions_func=definitions_func_,
    inputs={'in_u': "unow", 'in_v': "vnow"},
    global_inputs={'dt': dt_, 'dx': dx_, 'dy': dy_, 'eps': eps_},
    outputs={'out_u': "unew", 'out_v': "vnew"},
    mode=gt.mode.NUMPY
    )
```

Listing 9: Example code for the same stencil instantiation using the domain decomposition library.

in the domain decomposition libary version the stencil needs to be registered to the `DomainDecomposition` class instead.

As with the field registration this is due to distributed nature of the domain decomposition library. Each subdivision needs its own gridtools stencil to use for their computations.

However, for stencil instantiation the code modifications needed are very minimal. Listing 8 compared to Listing 9 show that the only adjustments needed are the function call from `gt.NGStencil()` to `"DomainDecompositionInstance".register_stencil()`, and for each field in the inputs and outputs instead of directly providing the numpy arrays registering their names.

Also, note that the domain keyword can be dropped, since the subdivisions need to define their domain based on the halos provided in field registration.

3.2.6 Main: Accessing field values

In some cases user created code may need to access the values of a field directly. Since the domain decomposition library distributes the field values across all subdivisions directly accessing field values becomes more difficult. However, in many cases the field values needed in one subdivision are only the field values of that subdivision. For this specific case, the `DomainSubdivision` class provides a `get_interior_field()` function.

Note that accessing the fields directly also has to involve a manual iteration over all


```
for sd in prep_domain.subdivisions:
    internal_field[:, :, :] = sd.get_interior_field("yv")[:, :, :])
```

Listing 10: Example code for accessing the interior field of a subdivision directly.

local subdivisions. Listing 10 shows an example of direct field access.

Accessing the values of a field at a location that is not in the local subdivision is not supported. In the distributed nature of domain decomposition direct field access would have to involve additional communication calls outside of simple boundary exchanges.

3.2.7 Main: Applying global boundary conditions.

Applying the global boundary condition when using the domain decomposition library needs some code modifications to account for the distributed subdivisions. Since not all subdivisions contain the global boundary, the `DomainDecomposition` class provides the function `set_boundary_condition()` to handle the transfer of the boundary condition in a given direction to the responsible subdivisions.

The `set_boundary_condition()` function needs as input the name of the field, the direction of the global boundary, the size of the global halo for that boundary, and a numpy array of the correct size of the global boundary in that direction containing the boundary values.

The direction of the boundary is encoded in a number ranging from 0 to 5 and corresponds to the same order as used in the halo definition for the field registration i.e. 0=negative x-direction, 1=positive x-direction, 2=negative y-direction, 3=positive y-direction, 4=negative z-direction, and 5=positive z-direction.

After the boundaries are set this way, calling the `apply_boundary_condition()` function of the `DomainDecomposition` class will actually apply them.

The set and apply function are separated so that in cases in which the boundary condition does not change during the runtime it only needs to be set once.

Listing 11 shows an example of setting and applying a boundary condition during time stepping. Note that in this example the code has to manually iterate over the local subdivisions to make sure that any subdivision gets the correct boundary condition applied.

3.2.8 Main: Swapping and communication of fields.

An often used technique in stencil codes is to swap between two arrays during time stepping where one array contains the current solution and the other array is used to store the next solution. Fields are not stored globally when using the domain decomposition library therefore swapping has to happen indirectly. The `DomainDecomposition` class provides a function called `swap_fields(field1, field2)` that delegates the swapping of the two fields to the underlying subdivisions. Listing 12 shows an example.

```
# Set the boundaries
t = (n + 1) * float(dt)
unew_west = {}
unew_east = {}
unew_north = {}
unew_south = {}
for sd in prepared_domain.subdivisions:
    unew_west[sd] = (- 2. * eps * np.pi
                     * np.exp(- 5. * np.pi * np.pi * eps * t)
                     * np.sin(np.pi
                     * sd.get_interior_field("yv")[:nb, :, 0]))
    unew_east[sd] = (- 2. * eps * np.pi
                     * np.exp(- 5. * np.pi * np.pi * eps * t)
                     * np.sin(np.pi
                     * sd.get_interior_field("yv")[-nb:, :, 0]))
    unew_north[sd] = np.zeros((sd.size[0], nb))
    unew_south[sd] = np.zeros((sd.size[0], nb))

    unew_west[sd] = unew_west[sd].reshape((halo[0],
                                           sd.size[1],
                                           sd.size[2]))
    unew_east[sd] = unew_east[sd].reshape((halo[1],
                                           sd.size[1],
                                           sd.size[2]))
    unew_north[sd] = unew_north[sd].reshape((sd.size[0],
                                             halo[2],
                                             sd.size[2]))
    unew_south[sd] = unew_south[sd].reshape((sd.size[0],
                                             halo[3],
                                             sd.size[2]))

    sd.set_boundary_condition("unow", 0, unew_west[sd])
    sd.set_boundary_condition("unow", 1, unew_east[sd])
    sd.set_boundary_condition("unow", 2, unew_north[sd])
    sd.set_boundary_condition("unow", 3, unew_south[sd])

# Apply boundary conditions for all subdivisions:
prepared_domain.apply_boundary_condition("unow")
```

Listing 11: Example code of handling the global boundary condition.

```
# Advance the time levels
prep_domain.swap_fields("unow", "unew")
prep_domain.swap_fields("vnow", "vnew")
```

Listing 12: Example swapping two arrays in the domain decomposition library.


```
# Communicate partition boundaries
prep_domain.communicate("unew")
prep_domain.communicate("vnew")
```

Listing 13: Example calling for internal boundary exchange of two fields in the domain decomposition library.

```
# Save
if (save_freq > 0) and (n % save_freq == 0):
    prep_domain.save_fields(["unew", "vnew"], postfix="t_"+str(n + 1))
```

Listing 14: Example calling to save two fields during time stepping using the domain decomposition library.

Because the solution arrays are distributed when using the domain decomposition library in between time steps the internal boundaries between subdivisions have to be exchanged. Exchanging the internal boundaries is the main overhead added by decomposing and distributing the domain of a stencil computation.

However, for the user this exchange is as simple as calling the `communicate(fieldname)` function of the `DomainDecomposition` class for each field that needs its boundaries exchanged. Listing 13 shows an example of this.

3.2.9 Main: Saving fields.

Because the fields are stored in the distributed subdivisions when using the domain decomposition library, saving them to file during or after time stepping is slightly more complicated than doing so in the serial case.

For the user the `DomainDecomposition` class provides a simple `save_fields()` function to save any field at any time. The `save_fields()` function takes a list of field names as well as optionally the path, a prefix, and a postfix for the files as input. When `save_fields()` is called, distributed subdivisions will save their respective parts of the listed fields to their own file. This means that each saved field will produce a number of separate files. The next subsection will outline how these separate files can then be combined after the distributed computations are done. Listing 14 shows a small example of saving fields during runtime.

3.2.10 Post-process: Combining fields.

The output of the distributed computations with the domain decomposition library are one file for each subdivision, each field and each save point. These many files can be combined back into one file for each field and each save point. To do this the `DomainPostprocess` class needs to be initialized and then provides the `combine_output_files()` function.

```
postproc = DomainPostprocess(path=path, prefix=prefix)
postproc.combine_output_files([100, 100, 1], "unew", path=path, prefix
    =prefix, postfix="t_0", save=True, cleanup=True)
postproc.combine_output_files([100, 100, 1], "vnew", path=path, prefix
    =prefix, postfix="t_0", save=True, cleanup=True)
```

Listing 15: Example post-processing.

The initialization is required to load the subdivision information that was generated in the pre-process. The subdivision information is important for the post-process because it stores the global coordinates for each subdivision. Once the `DomainPostprocess` class is initialized the `combine_output_files()` function can be called.. The `combine_output_files()` function needs the size of the domain, the field name as its main input. Also if defined during the saving of the fields the same path, prefix, and postfix needs to be provided. Optionally the binary flags `save` and `cleanup` can respectively be used to control if a new file with the combined file should be saved and if the old separate files should be deleted.

Listing 15 shows an example initialization and call to combine the subdivision files for time step 0 that deletes the subdivision files after they are combined.

3.3 MPI One-Sided

The library includes an option to switch the communication strategy from the common two-way send/receive model to instead use MPI one-sided one-way communication model. The option to use MPI one-sided is enabled by initializing the `DomainDecomposition` (shown in Listing 4) with the optional keyword `comm_onesided` set to `True`.

3.3.1 Communication procedure

MPI4Py supports three remote operations, namely `Put`, `Get`, and `Accumulate`. These operations can transfer data between specific memory buffers called windows. The library uses the `Put` one-sided operation to keep the communication functionalities similar between the two-way and one-way models. In keeping the similarity between both communication models each subdivision has to have a window for each field and each direction, i.e. each halo region is in separate window. With this setup the communication procedure follows these steps: The one-sided communication iterates over the directions, for each direction it uses `Put` to transfer the outermost region of that direction into the window of the neighboring subdivision. MPI one-sided `Put` can only transfer contiguous data similar to the MPI `Isend` operation. This is done with the help of NumPys `ascontiguousarray` function and the indices stored in the `send_slices` NumPy views as described in the communication paragraph of Section 3.1.2. For synchronization the MPI one-sided `Fence()` method is called before and after the `Put` operation.

Once all the halo regions are transferred into their respective window, a local copy operation is necessary to transfer the halo region from the window into the actual NumPy array. It has to be placed into the halo region of the opposite direction. Copying into to the opposite direction is needed because this corresponds to the boundary received from the neighbor in that opposite direction i.e. the neighbor located in the negative x-direction provides the halo region in positive x-direction. Figure 8 illustrates these two steps and should highlight why the data in the window has to be copied to the opposite side.

3.3.2 Windows as buffers

The communication procedure described in the section before relies on the MPI one-sided windows as buffers for the transfer and as a result requires an additional copy operation. An idea to avoid this would be to declare the whole field array as a window. However, for all but one dimension the halo regions are not contiguous and it does not seem to be supported to `Put` data into the window in a non-contiguous way. The MPICH documentation of the `Put` operation mentions displacements from the start of the window can be provided as an argument, but not a stride for non-contiguous data. Additionally, the MPI4Py `Put` function does not even have the displacement as an argument.

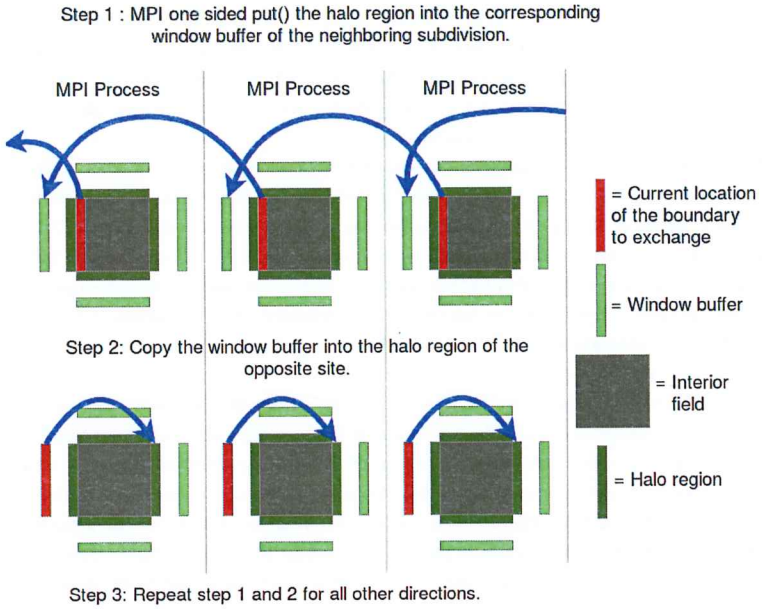


Figure 8: Schematic representation of the MPI one-sided communication procedure.

Therefore, at least at the moment it seems necessary to use the MPI one-sided windows as a buffer that allows the NumPy `ascontiguousarray` function and copying using NumPy views to handle the transfer of the halo regions that are non-contiguous in the NumPy array.

3.3.3 Potential limitations

Keeping the implementation of the one-sided communication similar to the two-way communication procedure has some limiting effect on the potential for one-way communication.

Specifically, a limitation arises because of limitations on the number of MPI communicators. Each MPI one-sided window is connected to the communicator given at its construction. However, each window also internally creates a separate MPI communicator.

As mentioned before, the current implementation requires a MPI one-sided window per direction, per field, and per subdivision. Most MPI implementations limit the number of MPI communicators to 2048. [3] With this setup the 2048 limitation can be reached very easily for large domains. For example, a stencil running on 8 nodes with 6 MPI processors per node and 1 subdivision per processor already creates $8 \cdot 6 \cdot 1 \cdot 6 = 288$ windows per field. In that case, if more than 7 fields are used the MPI communicator limit is reached.

This limitation is accounted for by only creating a MPI one-sided window when it is actually needed e.g. excluding directions with a halo size of 0.

Nevertheless, this limitation can still make it not feasible to use the one-way communication for larger stencil codes. By designing the one-sided communication procedure to be less similar to the two-way communication and be more optimized for one-sided communication this limitation could be avoided.

[3] See for example the following mailing list thread discussing this limit: https://lists.mpich.org/pipermail/discuss/2016-September/004908.html Accessed 25.9.18

4 Case study

To show the functionality and applicability of the automatic domain decomposition library a few case studies were carried out. This chapter and the following sections will outline each case study and the corresponding experimental results.

The base implementation of each test case was provided by Stefano Ubbiali written in Python and using GT4Py stencils.

4.1 Case 1: Burger's equation

Burger's equation is a well-known nonlinear partial differential equation. Burger's equation can be used to model various physical phenomena. Most commonly it is used as a model for traffic flow or shock waves in a fluid. Additionally it is widely used to test numerical schemes since it has analytical solutions for a set of initial conditions.

4.1.1 Case description

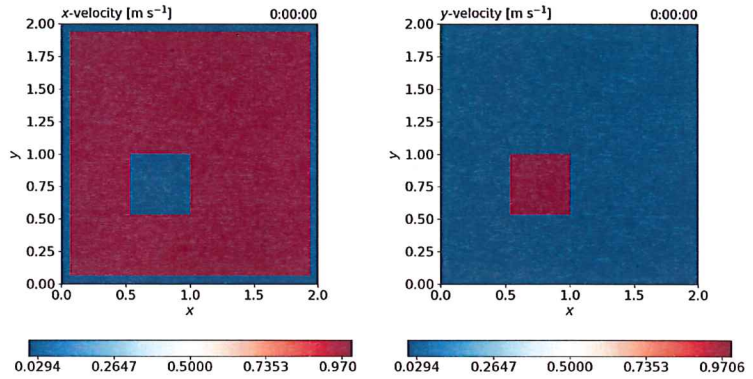
The two-dimensional, viscid Burger's equation is given by the following system of scalar equations as described in Zhao et al. [2011]:

$$\begin{aligned} \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} &= \varepsilon\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \\ \frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} &= \varepsilon\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \\ \text{with } (x, y, t) &\in D \times (0, T] \end{aligned} \tag{4}$$

These two equations characterize the Burger equation as a set of equations for the velocity in x and y direction. Both equations consist of two parts. On the left hand side, the advection of the velocity itself. On the right hand side, the diffusion caused by viscosity.

To complete the full description of the Burger equation, the initial and boundary conditions are generally given in the following form:

$$\begin{aligned} &\text{Initial conditions:} \\ u(x, y, 0) &= u_0(x, y), \ (x, y) \in D \\ v(x, y, 0) &= v_0(x, y), \ (x, y) \in D \\ &\text{Boundary conditions:} \\ u(x, y, t) &= f(x, y, t), \ (x, y, t) \in \partial D \times (0, T] \\ v(x, y, t) &= g(x, y, t), \ (x, y, t) \in \partial D \times (0, T] \end{aligned} \tag{5}$$



(a) X-velocity initial condition. (b) Y-Velocity initial condition.

Figure 9: Initial condition for the Shankar test case.

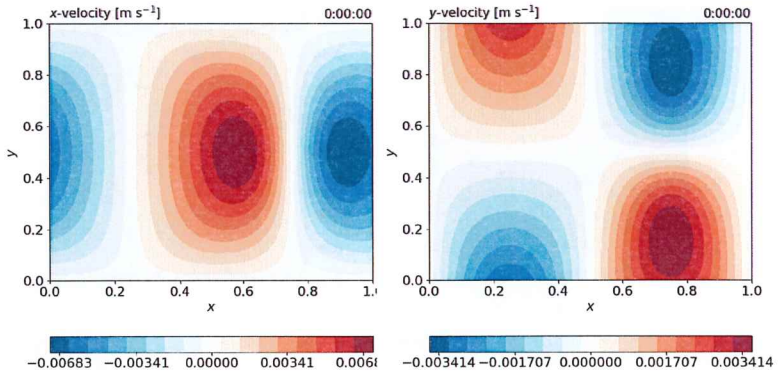
Shankar conditions: The first set of initial and boundary conditions are the ones used by Shankar. [4]

The following equations describe the Shankar test case fully and Fig. 9a and Fig. 9b visualizes the initial condition.

Boundary conditions:	**Initial conditions:**
$f(x, y, t) = 0$	$u_0(x, y) = \begin{cases} 0, & \text{in } [0.5, 1.0] \times [0.5, 1.0] \\ 1, & \text{otherwise} \end{cases}$
$g(x, y, t) = 0$	$v_0(x, y) = \begin{cases} 1, & \text{in } [0.5, 1.0] \times [0.5, 1.0] \\ 0, & \text{otherwise} \end{cases}$
Other parameters:	**Domain:**
Viscosity: $\varepsilon = 0.01$	$D = [0, 2] \times [0, 2]$
	$T = 0.6$

(6)

[4] https://ch.mathworks.com/matlabcentral/fileexchange/38087-burgers-equation-in-1d-and-2d Accessed: 25.9.18



(a) X-velocity initial condition. (b) Y-Velocity initial condition.

Figure 10: Initial condition for the Zhao test case.

Zhao conditions: The second set of initial and boundary conditions are the same as used as example 1 in Zhao et al. [2011]:

Boundary conditions:

$$\mathrm{f}(x,y,t) = \begin{cases} -2\varepsilon\pi \exp^{-5\pi^2\varepsilon t} \sin(\pi y), \text{ for } x = 0, y \in [0,1] \\ -2\varepsilon\pi \exp^{-5\pi^2\varepsilon t} \sin(\pi y), \text{ for } x = 1, y \in [0,1] \\ 0, \text{ for } x \in [0,1], y = 0 \\ 0, \text{ for } x \in [0,1], y = 1 \end{cases}$$

$$\mathrm{g}(x,y,t) = \begin{cases} 0, \text{ for } x = 0, y \in [0,1] \\ 0, \text{ for } x = 1, y \in [0,1] \\ -\varepsilon\pi \exp^{-5\pi^2\varepsilon t} \sin(2\pi x), \text{ for } x \in [0,1], y = 0 \\ \varepsilon\pi \exp^{-5\pi^2\varepsilon t} \sin(2\pi x), \text{ for } x \in [0,1], y = 1 \end{cases}$$

Other parameters:

Viscosity: $\varepsilon = 0.01$

Initial conditions:

$$\mathrm{u}_0(x,y) = \frac{-4\varepsilon\pi \cos(2\pi x)\sin(\pi y)}{2+\sin(2\pi x)\sin(\pi y)}$$

$$\mathrm{v}_0(x,y) = \frac{-2\varepsilon\pi \sin(2\pi x)\cos(\pi y)}{2+\sin(2\pi x)\sin(\pi y)}$$

Domain:

$\mathrm{D} = [0,1] \times [0,1]$

$\mathrm{T} = 1$

(7)

Notable about this set of initial and boundary condition is that they admit exact solutions. The exact solutions as provided in Zhao et al. [2011] are:

$$u(x,y,t) = -2\varepsilon \frac{2\pi \exp^{-5\pi^2\varepsilon t} \cos(2\pi x)\sin(\pi y)}{2+\exp^{-5\pi^2\varepsilon t}\sin(2\pi x)\sin(\pi y)}$$
$$v(x,y,t) = -2\varepsilon \frac{\pi \exp^{-5\pi^2\varepsilon t} \sin(2\pi x)\cos(\pi y)}{2+\exp^{-5\pi^2\varepsilon t}\sin(2\pi x)\sin(\pi y)} \quad (8)$$

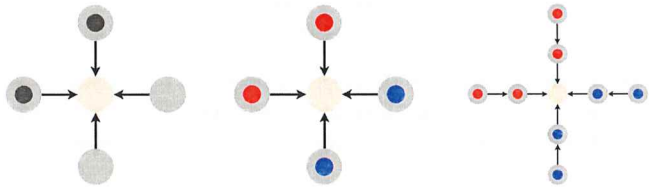
4.1.2 Implementation details

Stencil details: Both the Shankar and Zhao conditions can be solved using different numerical stencils for the computation. To show some variance and flexibility three stencils can be used to solve the test case.

The forward-backward stencil combines forward finite differences to compute the time derivatives with backward finite differences to compute the first-order space derivatives and a second-order centered scheme to compute the second-order derivatives. See Fig. 11a for a visual representation of this stencil.

The other two stencil~~s~~ are called upwind schemes, since they use the values only from the direction the velocity is coming from. The first-order upwind scheme uses the ~~directly~~ DIRECT neighboring grid points like the forward-backward stencil. The difference is that only the values from upwind are used in the actual computation for the next value. See Fig. 11b for a visual representation of this stencil.

The third-order upwind stencil needs two neighboring grid points in each direction to increase the accuracy of the approximated space derivative. See Fig. 11c for a visual representation of this stencil.



(a) Forward-backward (b) Upwind first order (c) Upwind third order

Figure 11: Schematic representation for three possible stencils for the Burger's equation. The large gray circle indicate the grid points necessary for the computation of the Laplacian for the diffusion part of the Burger's equation. The small dark gray circle indicate grid points necessary for the backward finite difference. The red and blue small circles indicate grid points needed for the upwind computations in the case of positive velocity or negative velocity respectively.

+ SPECIFY WHICH IS THE POSITIVE / NEGATIVE DIRECTION FOR EACH AXIS

4.1.3 Validation

Changing stencil computation from serial to distributed can cause various errors to appear. Therefore, it is important to validate the solutions obtained using the domain

decomposition library against reference solutions.

Comparison with the exact solution is a useful tool to validate any stencil code. The Burger's equation and specifically the Zhao setup presented before provide such reference and exact solutions.

Figure 12 shows these validation results. The first four plots in Fig. 12 show the spatial distribution of the error of either the domain decomposed solution (left panel) or the reference solution (right panel). This is an actual error since it compares the computed solutions against the analytical, exact solution.

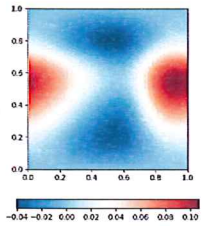
This comparison is done to verify that the addition of internal boundaries between subdivisions does not add to the spatial error distribution of the computation. These particular plots were generated using 16 subdivisions but the same plots for 8, 4, or 32 subdivisions show the same results.

Additionally, Fig. 12e and Fig. 12f show the spatial distribution of the difference between the reference solution and the domain decomposed solution. There is a difference between the reference solution and the domain decomposed solution because decomposing the domain necessarily changes the ordering of some computations causing the difference due to floating point operations not guaranteeing bit identical results. The spatial distribution of this difference is not identical between the velocity u (shown in Fig. 12e) and velocity v (shown in Fig. 12f)

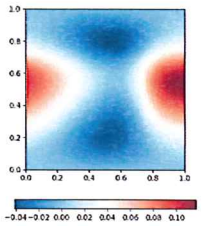
Not shown here is a plot comparing different setups of the domain decomposed solution e.g. the difference between all subdivisions belonging to the same partition (i.e. only local communication) and subdivisions belonging to multiple partitions (i.e. MPI communication). These plots are not shown because they would show no spatial distribution and instead a difference of zero everywhere. Meaning that for the same number of subdivisions the way of communicating their boundaries between each other has no influence on the computation and produces bit-identical results as expected.

Error time evolution: The previous plots have focused on the spatial distribution of differences because any coding errors in the subdivision boundary exchange would clearly show up in these comparisons. However, changing the code from serial to domain decomposed could potentially introduce erroneous numerical diffusion. Such an error would increase over many time steps. Therefore, Fig. 13 shows that the error does not increase in later time steps. The absolute error shown here in fact decreases over time since the solution as a whole decreases over time due to physical diffusion in the Burger's equation.

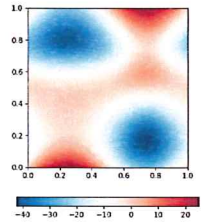
The time evolution of the error in Fig. 13 was measured for the reference solution, a 4 subdivision and 16 subdivision solutions with only local communication, as well as a 16 subdivision solution with MPI communication. However, the behavior of the absolute error over time is so similar between all of these that in the lines in Fig. 13 overlap for



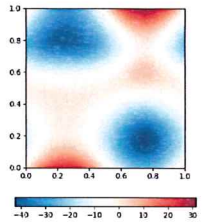
(a) Difference in velocity u field between exact solution and domain decomposed solution averaged over all time steps.



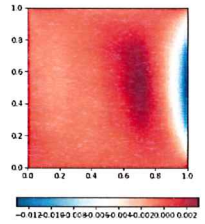
(b) Difference in velocity u field between exact solution and reference solution averaged over all time steps.



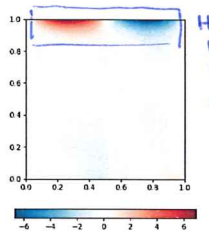
(c) Difference in velocity v field between exact solution and domain decomposed solution averaged over all time steps.



(d) Difference in velocity v field between exact solution and reference solution averaged over all time steps.

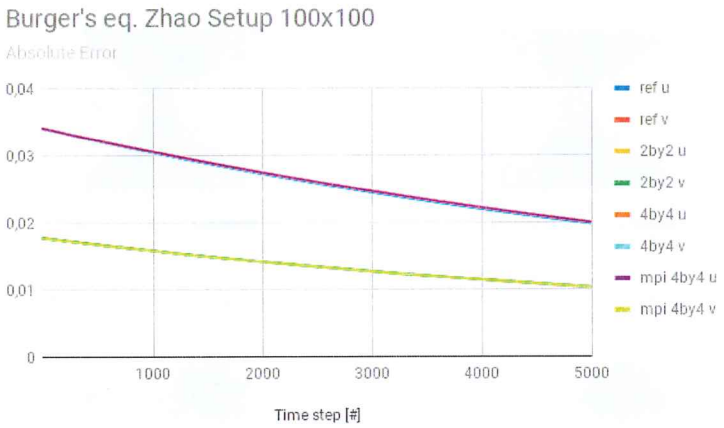


(e) Difference in velocity u field between reference solution and domain decomposed solution averaged over all time steps.



(f) Difference in velocity v field between reference solution and domain decomposed solution averaged over all time steps.

Figure 12: Burger's equation Zhao Setup. Exact solution refers to the analytical solution shown in Eq. (8). Reference refers to solutions computed using the original, serial Python and GridTools4Py version. Lastly, domain decomposed solution refers to solutions computed using the domain decomposition library. The domain size for this validation is 100 by 100 grid points and the computations are run for 1000 time steps.



IS THIS AN INTEGRAL NORM ERROR? SPECIFY...

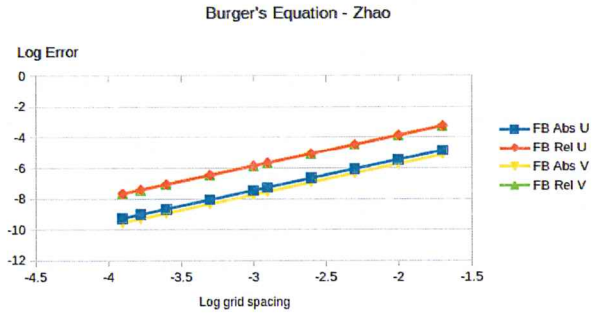
Figure 13: Time evolution of the absolute error in a Burger's equation Zhao setup for the reference code and a few different versions of the domain decomposed code.

all u velocity solutions and all v velocity solutions.

The difference to the exact solution has a spatial distribution because it originates in the discretization of the analytical initial condition and computations. This relationship between the error and the grid discretization can be used to validate the domain decomposed solutions by performing a grid refinement study.

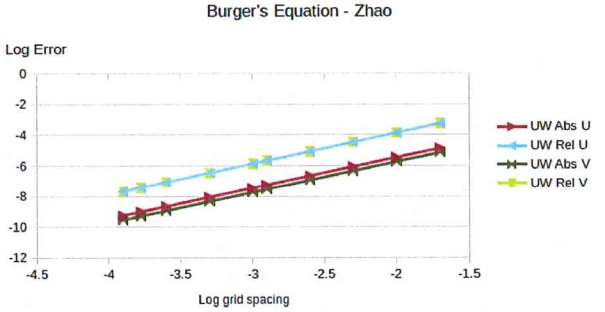
Grid refinement study is a method to validate code most commonly used in computational fluid mechanics. In a grid refinement study the error of a computation is calculated on various grid sizes that are made smaller by a factor. It is most common to half the grid spacing per measurement. If this refinement is done in the asymptotic range of convergence for a stencil then the leading term of the truncation error dominates the overall error. This means that if error for each grid spacing is plotted in a log-log plot the slope of the line ~~is~~ REPRESENT the measured order of accuracy, ~~and~~ WHICH if the implementation is correct, should be equal to the theoretical order of accuracy of the stencil.

Figure 14 shows such a grid refinement study for the Burger's equation Zhao setup. Note that both the forward-backward and the upwind stencil have an order of accuracy of 1 (2). The upwind third-order stencil should have, as the name suggests, an order of accuracy of 3. However, this is only the order of accuracy of the spatial stencil. Since the Burger's equation Zhao setup uses a very simple first-order ~~in~~ time discretization, the measured order of accuracy for the third-order upwind scheme is limited by the time discretization.

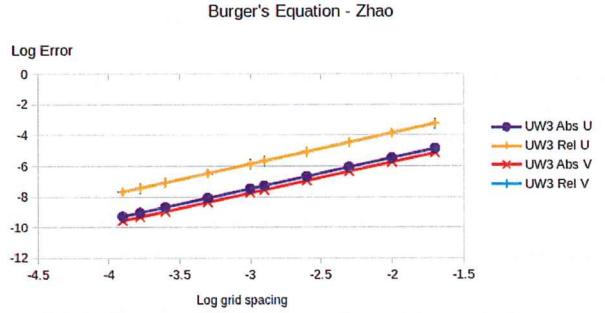


(a) Grid refinement for the forward-backward stencil.

ADD REFERENCE LINES



(b) Grid refinement for the upwind stencil.



(c) Grid refinement for the upwind third-order stencil.

Figure 14: Grid refinement study of the Burger's equation Zhao setup for the three stencils described before.

4.1.4 Performance and Scalability

The following paragraphs outline the setup and results of a few experiments conducted on the supercomputer cluster called Greina at CSCS.

Baseline and limits The baseline for all scaling experiments is the serial GridTools4Py code. By running this stencil code on increasing grid sizes the limit for the serial baseline is determined to be somewhere between grids of size 24 thousand by 24 thousand (576 million grid points) and 32 thousand by 32 thousand (1.024 billion grid points). The baseline can compute results for the 24000×24000 grid but runs into memory errors for the 32000×32000 grid.

The baseline code allocates 2 grid sized NumPy arrays for the grid spacing used in the boundary condition computation and 6 grid sized NumPy arrays for the two velocity fields. These NumPy arrays use `np.float64` for their data type which uses 8 byte per array item. So in theory the baseline should use $24000 \cdot 24000 \cdot 8 \cdot 8\text{B} = 34.33\text{GB}$ of memory for the fields.

However, the SLURM scheduling system on Greina reports for the 24 thousand by 24 thousand test case much larger memory usage. Specifically, it reports a maximum resident set size (maxRSS) of 60.73 GB and a maximum virtual memory size (maxVMSize) of 74.24 GB. ~~Which,~~ This is consistent with the observation that for larger grids the baseline runs out of memory for nodes with 64 GB of memory.

This difference of a roughly a factor 2 between theoretical memory usage and measured memory usage is mostly likely a result of the non-optimized, prototype GridTools4Py NumPy back-end as the only available back-end during this thesis. At a later time this back-end NumPy back-end will be replaced by the actual performance optimized C++ back-end of GridTools.

Therefore, the scalability experiments using this test case have a hard memory limit of 24000×24000 grid points on nodes with 64 GB of memory. This memory limit also restricts the scope of the scalability experiments. For strong scaling having a baseline of just 24000×24000 grid points combined with a non-optimized back-end results in very unsatisfactory strong scaling efficiency for experiments using multiple nodes and multiple processing units per node. Examples of strong scaling on various different hardware are presented in the next subsection.

Additionally, the advantages of domain decomposition should present more clearly in weak scaling since domain decomposition is mainly used to make it feasible to run stencil codes on grids that are larger than the memory of a single node.

Weak scaling For the weak scaling experiments the grid size per node was kept constant at $24000 \times 24000 = 576000000$. The experiments were performed on 1, 2, 4, and 8 nodes on the Greina system of CSCS. Figure 15 shows the results for 2, 4, 8, or 16 tasks

per node. After the initial increase in time when scaling from a single node to multiple the times stay more or less constant as expected.

The decrease in runtime from single to 2 nodes for the 16 tasks per node experiment is an outlier and is probably the result of the heterogeneous nodes on Greina i.e. the hardware variation on the nodes influence the performance of the weak scaling and not just the performance of the domain decomposition library.

The heterogeneous nature of the nodes on Greina is explored more thoroughly in the next set of experiments.

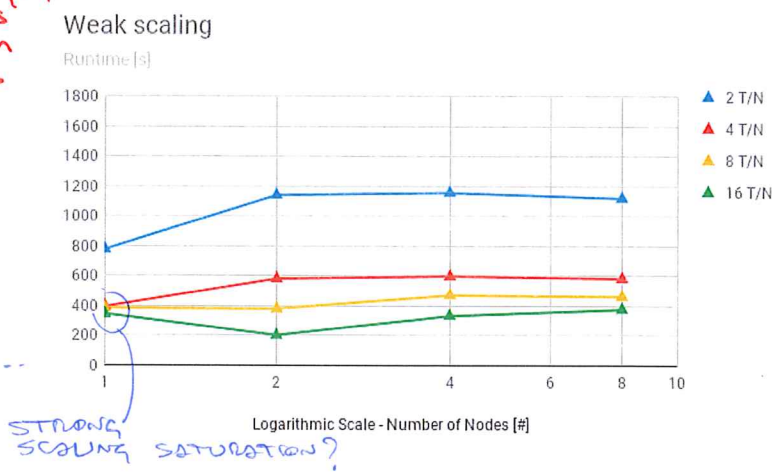


Figure 15: Weak scaling for different number of tasks per node run on 1, 2, 4, and 8 nodes.

4.2 Case 2: Burger's equation with heterogeneous load balancing

Part of the motivation to use graph partitioning methods for domain decomposition was that these methods can balance heterogeneous loads. This section outlines how the domain decomposition library can be used to distribute subdivisions balanced for heterogeneous systems.

Name	Memory [GB]	CPUs [#]	CPU frequency [GHz]
greina4	64	18	2.3
greina6	64	12	2.2
greina8	64	10	2.2
greina11	128	24	2.2
greina13	128	24	2.5
greina14	128	24	2.5
greina15	128	32	2.1
greina16	128	64	2.1
greina17	64	12	3.5
greina28	128	40	2.2
greina31	196	64	2.1
greina32	196	64	2.1

Table 2: Performance statistics of the various Greina nodes used in these experiments.

4.2.1 Heterogeneous system:

The Greina cluster at CSCS is such a heterogeneous system. Table 2 outlines the various hardware and performance differences between nodes.

To determine the factors to balance the load on these nodes, single node strong scaling experiments were performed. In these single node experiments the domain size was again fixed to 24000×24000. For strong scaling the domain size stays constant even for increased number of processing units. For each node the number of tasks was increased by a factor of 2 from the single, reference task up to 128 tasks.

Figure 16 shows the large differences between the performance speedup of various nodes. This figure also shows the fast flattening of the speedup for this small problem size.

Nonetheless, the measurements used to generate Fig. 16 was also used to determine the best configuration for the heterogeneous load for each node shown in Table 3.

It is noteworthy, that the tasks per node with the best speedup on many nodes exceeds the number of CPUs. This can be seen as evidence that processing units are idle if only one task is assigned to them. This is most likely the result of non-efficient overlap between the communication and computation caused by the implementation of the domain decomposition library.

Efficiently overlapping communication and computation is a key part of good performance in multi-node stencil codes.

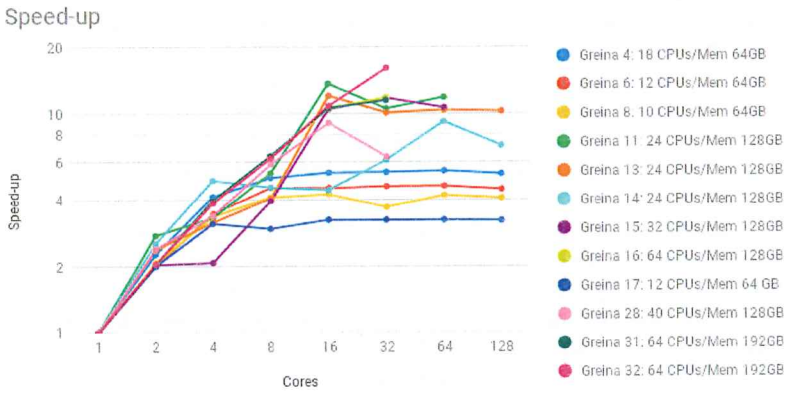


Figure 16: Strong scaling - speedup for the various Greina nodes.

4.2.2 Dynamic load balancing - procedure:

Performing domain decomposition for heterogeneous systems requires a few manual steps outlined in the following paragraph.

The Metis graph partition library can balance heterogeneous loads according to target weights. These target weights have to be specified by the user in a simple text file with the following structure:

partition_1 - partition_x = .p1
partition_y - partition_z = .p2
...

Where partition_number correspond to MPI ranks and therefore the range of partitions should be the size of the number of tasks assigned to that particular node. Therefore, the file will contain one line for each node with partition ranges increasing by the number of tasks assigned to that node. The right hand side i.e. p1 and p2 are percentages of the total load that should be partitioned into the corresponding partition ranges.

Name	Tasks per Node with best speedup
greina4	8
greina6	8
greina8	16
greina11	16
greina13	16
greina14	64
greina15	32
greina16	32
greina17	16
greina28	16
greina31	32
greina32	32

Table 3: Table of the task per node ratio that reached the best speedup in the single node experiments.

These percentages are the main parameter for heterogeneous load balancing. One way to set them is to compute the percentage of subdivisions per task are assigned to the specific nodes. This way all nodes should have the same amount of subdivisions but not the same amount of tasks, which is similar to the single node setup that determined the optimal number of tasks for the node previously.

With the target weights, the Metis library can be called with the option `-tpwgts`. Unfortunately, currently PyMetis does not support the target weight option. Therefore, this step requires the actual Metis library and for the domain decomposition library pre-process to save the source graph in the Metis format. Once the pre-process produces the Metis formatted source graph, the partitioning can be done with the following command:

```
gpmetis subdomains_metis.dat np -tpwgts="target_weights.dat" -contig
```

where `np` is the total number of partitions or tasks.

The Metis library then produces the same partitioning file as PyMetis would.

In addition to the dynamic load balanced partitioning file also the cluster scheduler and MPI need files to use the correct load balancing. Specifically, MPI needs the list of nodes to use and the cluster scheduler needs to assign the correct MPI ranks to the correct node.

For SLURM this can be done with a combination of the `--nodelist=[comma-separated-nodelist]` and `--distribution=arbitrary` option. The arbitrary distribution option

requires setting the environment variable `SLURM_HOSTFILE` to a file containing the host name for the MPI rank corresponding to the line number of the file.

Such a host name to rank file can easily be generated from the partitioning file by replacing the numbers with the assigned host name.

With the dynamically load balanced partitioning file and the host-to-rank file the remaining procedure to run a stencil code stays the same.

4.2.3 Example results

4.3 Case 3: Shallow Water Equations on a Sphere (SWES)

The shallow water equations are like the Burger's equation a well known set of equations used widely in testing and studying implementation of numerical methods. The shallow water equation are used to compute the velocities and height of a homogeneous, incompressible, and inviscid shallow layer of fluid.

4.3.1 Case description

The shallow water equations consist of two horizontal momentum equations and one mass continuity equation. These equations can be written in many forms, for example the advective form for spherical components as described in Williamson et al. [1992]:

$$\begin{aligned} \frac{\partial u}{\partial t} + \mathbf{v}\cdot\nabla u - \left(f + \frac{u}{a}\tan(\theta)\right)v + \frac{g}{a\cos(\theta)}\frac{\partial h}{\partial \lambda} &= 0 \\ \frac{\partial v}{\partial t} + \mathbf{v}\cdot\nabla v + \left(f + \frac{u}{a}\tan(\theta)\right)u + \frac{g}{a}\frac{\partial h}{\partial \theta} &= 0 \\ \frac{\partial h^*}{\partial t} + \mathbf{v}\cdot\nabla h^* + \frac{h^*}{a\cos(\theta)}\left(\frac{\partial u}{\partial \lambda} + \frac{\partial v\cos(\theta)}{\partial \theta}\right) &= 0\,, \end{aligned} \tag{9}$$

where f is the Coriolis parameter, g is the gravitational constant, and a is the radius of the sphere e.g. the radius of Earth.

Seven test cases are outlined in Williamson et al. [1992] to facilitate standardized testing of the shallow water equation solvers. A modified version of two of these were used in this thesis.

The most significant modification is the reduction of complexity by limiting the spherical domain to a band ranging from 85° to −85° to avoid more complex pole treatment. The artificial boundary that is created by this simplification uses Neumann conditions i.e. setting the flux of all fields between the north and south boundaries to zero.

In the east and west the grid on a sphere uses periodic boundaries.

IN THE LATITUDINAL DIRECTION

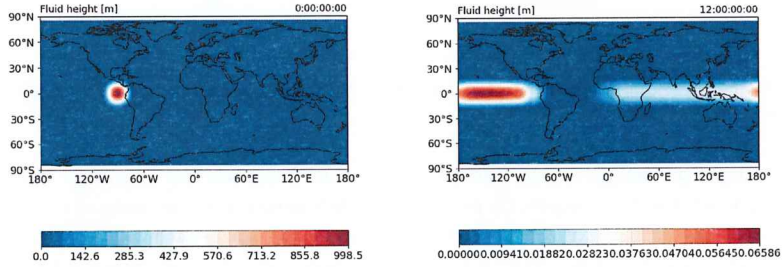
SWES initial conditions 1: The first test case described in the test suite from Williamson is the advection of a cosine bell and is designed to test only the advection component of the shallow water equations.

This is done by prescribing an advecting wind instead of solving the first two equations shown in (9). Specifically, the two ~~static~~ wind ~~fields~~ and the initial height field are given as:

STATIONARY COMPONENTS

$$\begin{aligned} u &= u_0\left(\cos(\theta)\cos(\alpha) + \sin(\theta)\cos(\lambda)\sin(\alpha)\right) \\ v &= -u_0\sin(\lambda)\sin(\alpha) \\ h(\lambda,\theta,t=0) &= \begin{cases} (h_0/2)\left(1+\cos(\pi r/R)\right) & \text{if } r < R \\ 0, & \text{otherwise} \end{cases}, \end{aligned} \tag{10}$$

where α is the angle between the axis of the solid body rotation and the polar axis and is set to 0.0 for these tests, h_0 is set to $1000m$, $u_0 = 2\pi a$ for a 12 days period, and θ and λ are the spherical coordinates.



(a) Height field for SWES initial condition 1.

(b) Spatial distribution of the absolute difference between reference and domain decomposed solutions averaged over 12 days simulation time.

Figure 17: Shallow Water Equation (SWES) test case 1, initial condition and validation results.

The cosine bell is created in the following way:

$$\begin{aligned} r &= a\arccos\left(\sin(\theta_c)\sin(\theta) + \cos(\theta_c)\cos(\theta)\cos(\lambda-\lambda_c)\right) \\ (\lambda_c,\theta_c) &= (3\pi/2, 0) \\ R &= a/3 \end{aligned} \tag{11}$$

This initial cosine bell height field is shown in Fig. 17a.

Advection only stencil: As mentioned this test case uses static wind fields and as a consequence has only to use one stencil for the advection of the height field. For the advection of the height field this test case computes the static velocity fields at staggered, mid-points between the grid points. Thus the advection only stencil can use the non-staggered velocity fields to compute the height field at the same mid-points. The staggered velocity fields are then used in combination with these intermediate mid-point height field values to advect the non-staggered height field accordingly.

In total this stencil uses 10 fields to perform the advection, including grid spacing fields needed due to the grid layout on the sphere.

Validation The test case is set up so that the cosine bell moves around the sphere in 12 days. Unfortunately, for this test case there does not exist an analytical solution. So here only a comparison with the result from the reference code is used to validate the domain decomposed version.

The time-averaged spatial distribution of the difference between the reference and domain decomposed solutions can be seen in Fig. 17b. Figure 17b clearly shows that there is a difference between the two versions that grows in time and is localized to the path of the cosine bell. However, the difference is many magnitudes smaller than the values of the height field and therefore falls into the error tolerance of the solution.

SWES initial conditions 2: The second setup used is the third test case in the test suite from Williamson and describes a steady state nonlinear zonal geostrophic flow with compact support.

The setup of the two initial velocities and height fields in this case involves the following equations:

Initial value constants:

$$w = 7.848e-6$$
$$K = 7.848e-6$$
$$h0 = 8e3$$
$$R = 4.0$$

Intermediate values:

$$A = 0.5w\left(2\omega + w\right)\cos\left(\theta\right)^2 + 0.25K^2 \cdot \cos\left(\theta\right)^{2R}\left(\left(R+1\right)\cos\left(\theta\right)^2 + \left(2R^2 - R - 2\right) - 2R^2\cos\left(\theta\right)^{-2}\right)$$
$$B = \left(2\left(\omega + w\right)K\right)\left(\left(R+1\right)\left(R+2\right)\right) \cdot \cos\left(\theta\right)^R\left(R^2 + 2R + 2 - \left(R+1\right)^2\cos\left(\theta\right)^2\right)$$
$$C = 0.25K^2\cos\left(\theta\right)^{2R}\left(\left(R+1\right)\cos\left(\theta\right)^2 - R + 2\right)$$

Actual initial fields:

$$h = h0 + \left(a^2A + a^2B\cos\left(R\phi\right) + a^2C\cos\left(2R\phi\right)\right)g$$
$$u = aw\cos\left(\theta\right) + aK\cos\left(\theta\right)^{R-1}\left(R\sin\left(\theta\right)^2 - \cos\left(\theta\right)^2\right)\cos\left(R\phi\right)$$
$$v = -aKR\cos\left(\theta\right)^{R-1}\sin\left(\theta\right)\sin\left(R\phi\right) \tag{12}$$

In the end these equations result in the initial height field shown in Fig. 18a.

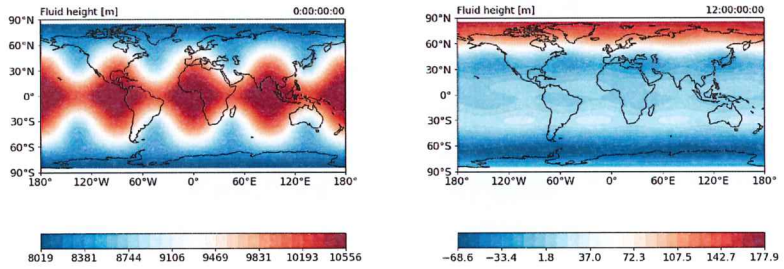
Lax-Wendroff and diffusion stencil: In contrast to the first SWES test case this test case uses non-static velocity fields. Specifically, it uses the well-known Lax-Wendroff method to advect the velocity fields as well as the height field.

In addition, the test case also applies diffusion to the two velocity fields and the height fields. The diffusion stencil has a halo of 2 in all directions, while the Lax-Wendroff method has a halo of 1 in all directions. Because of this, the global boundary for the diffusion has to be larger than for the Lax-Wendroff stencil. The serial, reference code handles this by copying the field values into extended grids before applying the diffusion stencils.

However, for larger grids this increases the memory needed drastically. Copying the fields in the code that uses the domain decomposition library would also involve manually looping over the local subdivisions and performing the copy for each subdivision separately.

To avoid these inconveniences the domain decomposition version of the SWES code applies the diffusion on a smaller subset of the fields. This means the outermost values on the global boundary are not diffused. Overall the effect of these non-diffused grid points

at the boundary should be insignificant. Especially, since the north / south boundary are already an approximation using the Neumann condition.



(a) Height field for SWES initial condition 2.

(b) Spatial distribution of the difference between reference and domain decomposed solutions averaged over 12 days simulation time.

Figure 18: Shallow Water Equation (SWES) test case 2, initial condition and validation results.

Validation: Similar to the first SWES test case the second test does also not provide an analytical solution and is also designed to be periodic with a 12 day cycle.

Figure 18b shows the spatial distribution of the difference between the reference and domain decomposed solutions averaged over 12 days. Again in contrast to the validation for the Burger's equation this plot shows a clear spatial distribution of the difference. That the largest difference occurs at both the south and north boundary might be an indication that the different methods to apply diffusion has some influence on the overall solution. However, also note that again the difference between the two versions is magnitudes smaller than the values of the height field as seen for example in initial conditions shown in the left panel of the same figure.

5 Conclusion

VERY NICE JOB! :)