Computational Science and Engineering
(Int. Master's Program)

Technische Universität München

Master's Thesis

Parallel localized radial basis function methods for the shallow water equations on the sphere

Author:	Igor Tominec
1st examiner:	Doc. Dr. Elisabeth Larsson
2nd examiner:	Prof. Dr. Hans-Joachim Bungartz
Thesis handed in on:	March 31, 2017

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

March 31, 2017 Igor Tominec

Acknowledgments

This project has been much more than a normal master project: it has very quickly evolved into a research adventure, the student-supervisor meetings have at the same time become my weekly highlights. In addition to that, my supervisor Docent Elisabeth Larsson has taken care of the discussions to be thrilling, of the humour to be at its finest and of this thesis and me to become better. Elisabeth, the amount of my gratitude to you for all of that and more can not be expressed using finite terms. In this spirit, the execution of Algorithm 0.1 starts as of the moment when you stop reading this line.

Algorithm 0.1

```
while (true)
  shoutoutloud <- 'Thank you! :-)'
end
```

Remark 0.2 *The loop in Algorithm 0.1 can be terminated upon request.*

Thanks to Grady Wright who generously shared his expertise on the shallow-water-equations in the most critical moments.

Afshin Zafari gave me a much needed support on performing tricks with *SuperGlue*, which I highly appreciate and am grateful for.

A big thanks for arranging the printing of this work goes to my dear friend Makis Chourdakis.

There is a very special person to whom I would also like to thank. Her name is Eva. She is my crime-partner, my cuddler and my favorite girlfriend. I think she deserves a gold medal for being able to stick with me.

Taxpayers of Germany, Slovenia and Sweden have substantially contributed to the completion of my master studies and this master thesis. Their support will not be forgotten.

The computations were performed on resources provided by SNIC through Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX) under Project SNIC 2017/7-34 and Project SNIC 2016/7-79.

The living expenses while staying abroad were covered by Slovene human resources development and scholarship fund.

Contents

Part I.

Introduction and Theory

1. Introduction

Recent destabilization of weather patterns is already causing unpleasant consequences for the humanity of the present world. That brings a sound reason to strengthen current research being done on numerical methods that help specialists from the field of geophysics to predict both, short- and long-term weather dynamics. The resulting algorithms should be as accurate as possible, but at the same time slim in terms of runtime and fully employable in terms of running them on several computing resources in parallel.

Definition 1.1 *Let $\underline{x} = (x, y, z)$ be an arbitrary location on the surface of the Earth (sphere), t denote the time, $\underline{u} = \underline{u}(\underline{x}, t)$ represent the wind field, $h = h(\underline{x}, t)$ the geopotential height, $f = f(\underline{x})$ the Coriolis force and g the gravitational constant.*

The shallow-water equations in Cartesian coordinate system then read as:

$$\begin{aligned} \underline{u}_t &= -P\big[(\underline{u} \cdot P\nabla)\underline{u} + (\underline{x} \times \underline{u})f + gP\nabla h)\big] \\ h_t &= -P\nabla \cdot (h\underline{u}), \end{aligned}$$

where P is a projection operator onto the spherical surface [6].

The shallow-water equations observed on the surface of the sphere serve as one of the most used benchmarking models of climate. Many already existing numerical approaches for its solution are available and new ones are being developed frequently, but their computational complexity is often high since they are based on global dependencies between discretization elements. For the same reason, their potential for parallelization is poor. Our mission is to develop a parallel simulation tool from scratch, tackling the spatial discretization using a localized version of the family of Radial basis function methods for solving partial differential equations called *Radial basis function – partition of unity method (RBF-PUM)*. The investigations so far indicate a high accuracy when dealing with smooth problems for an acceptable computational cost invested. However, no parallel approach for using *RBF-PUM* on PDEs has yet been investigated, but the literature briefly suggests the method to be embarrasingly parallel for interpolation problems [2], which is a base for PDE problems. In this sense, we use a shared memory task-based parallel framework *Superglue* [21], which has so far shown good results when applied to similiar problems. Finally, the spatial solution is in time forwarded by an explicit method, the 4th order *Runge-Kutta*. The solutions with respect to accuracy and runtime which are obtained using our *RBF-PUM* framework are compared to an already existing implementation using *Radial basis functions generated finite differences (RBF-FD)* [22] [5].

The main questions that are to be addressed throughout this work are:

- Is *RBF-PUM* appropriate for approaching nonlinear hyperbolic problems in combination with explicit time-stepping schemes?
- Are there any benefits delivered in comparison with similiar methods which were already applied to the shallow-water equations?

- How does *RBF-PUM* in combination with explicit time-stepping scale when implemented as a parallel algorithm?

In Chapter 2 we give an introduction to *RBF-PUM* which is our main method in question, but also to *RBF-FD* since it serves as a reference when performing the comparison. Both of these methods are based on a global RBF method (*RBF-Direct*), also described in this chapter.

Chapter 3 gives an introduction to the task-based parallel paradigm and to the parallel framework *SuperGlue*.

Chapter 4 deals with the space discretization of the shallow-water equations using *RBF-PUM*. Here, the differential operators are expanded in continuous and *RBF-PUM* sense. We also provide observations on how to form an open cover on the sphere, which hyperviscosity approach we choose and link the overlap of patches to the stability criteria. At the end, we outline the sequential algorithm of the derived numerical scheme.

Chapter 5 handles the strategy for parallel implementation of the sequential algorithm described at the end of Chapter 4, i.e. how the problem is split into tasks with certain dependencies. We also present a dense form for performing local matrix-vector products.

Chapter 6 checks how the implemented numerical scheme performs with respect to accuracy. Results are compared to *RBF-FD* [5] [22].

Chapter 7 tests how does the implemented parallel simulation tool perform with respect to runtime and scalability. Again, the results are compared to results obtained using *RBF-FD* [22].

In Chapter 8 we draw the conclusions based on Chapter 6 and Chapter 7. Lastly, we outline the possibilities for future research in Chapter 9.

2. Numerical methods

This chapter describes the family of Radial basis function methods which are appropriate for approximation of partial differential equations. Global RBF approximation is performed using *RBF-Direct*, while *RBF-FD* and *RBF-PUM* are its localized versions. The focus within this work is mainly put on *RBF-PUM*, but also on *RBF-FD* since this is the method upon which we compare our *RBF-PUM* solutions.

2.1. Global RBF approximation (RBF-Direct)

The basic tool is going to be an interpolant to data obtained at scattered points $\underline{x}_k \in \mathbb{R}^d, k = \{1, 2, ..., N\}$ formulated as,

$$s(\underline{x}) = \sum_{k=1}^{N} c_k \phi(||\underline{x} - \underline{x}_k||), \tag{2.1}$$

where $||\cdot||$ is an Euclidean norm on $\mathbb{R}^d$ and $\phi_k = \phi(||\underline{x} - \underline{x}_k||)$ is a chosen (conditionally) positive definite *radial function*.

Definition 2.1 *Let $\phi : \mathbb{R}^d \to \mathbb{R}$. Then $\phi(\underline{x}) = \phi(||\underline{x}||)$ is called a radial function.*

Remark 2.2 *We extend the latter definition denoting that $\underline{x}$ are nodes distanced from a node $\underline{x}_i$: $\phi(||\underline{x} - \underline{x}_i||) = \phi(\underline{x} - \underline{x}_i)$. This is the final notation going to be used for the rest of this document.*

Taking (2.1) and substituting $s(\underline{x})\Big|_{\underline{x}=\underline{x}_k} = f(\underline{x}_k)$, $\underline{x} \in \mathbb{R}^d$, $k = \{1, 2, ..., N\}$ where $f(\underline{x}_k)$ are known values evaluated in scattered points $\underline{x}_k$, gives us a linear system of equations:

$$\underbrace{\begin{pmatrix} \phi(||\underline{x}_1 - \underline{x}_1||) & \phi(||\underline{x}_1 - \underline{x}_2||) & \dots & \phi(||\underline{x}_1 - \underline{x}_N||) \\ \vdots & \ddots & \vdots & \\ \phi(||\underline{x}_N - \underline{x}_1||) & \phi(||\underline{x}_N - \underline{x}_2||) & \dots & \phi(||\underline{x}_N - \underline{x}_N||) \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} c_1 \\ \vdots \\ c_N \end{pmatrix}}_{\underline{c}} = \underbrace{\begin{pmatrix} \underline{f}(\underline{x}_1) \\ \vdots \\ \underline{f}(\underline{x}_N) \end{pmatrix}}_{\underline{f}}, \tag{2.2}$$

It was shown in [19] that for $\phi_k = \phi(||\underline{x} - \underline{x}_k||)$ being *positive definite functions*, the matrix $A \in \mathbb{R}^{N\times N}$ is positive definite meaning A is non-singular and therefore a unique solution to (2.2) exists. Moreover A is proven to also be non-singular for certain cases of using *conditionally* positive definite functions [16].

We shall refer to the interpolation method described above as *RBF-Direct*.

2.1.1. Properties of RBF-Direct

The *RBFs* we are interested in for applying to our problems are usually infinitely smooth and such functions posses a shape parameter ε which is of great importance when it comes to stability and convergence.

Definition 2.3 *Let the shape parameter $\varepsilon \to 0$. Then we refer to the regime of resulting $\phi(\underline{x} - \underline{x}_k) = \phi(||\underline{x} - \underline{x}_k||)$ as a* nearly flat basis function regime.

When the *RBFs* are in the near-flat regime then the matrix from (2.2) becomes ill-conditioned [9]. These problems can be avoided using certain methods (*RBF-QR* [15], *Contour-Pade* [11], *RBF-RA* [27]). On the other hand, the advantage of such a regime is in the method being more accurate up to some point. For certain infinitely smooth basis functions (Gaussians, Multiquadrics,..) the method achieves spectral accuracy [17].

We posed all of our equations to be appropriate for scattered points. That is a great advantage of *RBF* over some conventional methods only working with equidistant points. Besides, RBF interpolation is immune to certain problems regarding dimensionality, which is a much wanted property since, for example, interpolation using polynomial basis when using distinct nodes leads to a singular linear system (Mairhuber-Curtis theorem [2]).

That leaves us with the weakness of the method being ill-conditioned and in addition not very practical for applications, considering that computational complexity is high due to A being dense. This downside has been tackled using *localized methods* such as *RBF-FD* (finite differences approach) and *RBF-PUM* (partition of unity approach).

2.1.2. On polynomials and RBF

There exists an alternative formulation of (2.1) where a multivariate polynomial component up to some degree h is added to the interpolant,

$$s(\underline{x}) = \sum_{k=1}^{N} c_k \phi(||\underline{x} - \underline{x}_k||) + \sum_{j=1}^{p} b_j q_j(\underline{x}), \tag{2.3}$$

where b_j are coefficients and q_j are monomials forming the basis ordered in lexicographic order. For example, the highest degree $h = 1$ and number of dimensions $d = 2$ yield a linear basis $q = \{1, x, y\}$ consisting of $p = 3$ multivariate monomials. This approach makes interpolation exact for the order of the highest degree of the resulting polynomial. In order to guarantee a unique solution to (2.3), we have to produce the following p additional constraints. Keeping in mind the example of the basis from above we write: $\sum_{k=1}^{N} c_k = \sum_{k=1}^{N} x_k c_k = \sum_{k=1}^{N} y_k c_k = \ldots = \sum_{k=1}^{N} q_j(\underline{x}_k) c_k = 0$. In this spirit we present the matrix-vector notation of an interpolation problem based on (2.3),

$$\begin{pmatrix} A & Q \\ Q^T & 0 \end{pmatrix} \begin{pmatrix} \underline{c} \\ \underline{b} \end{pmatrix} = \begin{pmatrix} \underline{f} \\ 0 \end{pmatrix}, \tag{2.4}$$

where $\underline{c} = \{c_1, \ldots, c_N\}$, $\underline{b} = \{b_1, \ldots, b_p\}$, $\underline{f} = \{f_1, \ldots, f_N\}$ and submatrix $Q \in \mathbb{R}^{N \times p}$ has elements $Q_{ij} = q_j(\underline{x}_i)$.

That brings at least two beneficial properties [4]; improved accuracy at domain boundaries [8] and (for only including a 0-th degree monomial) improved accuracy of derivative approximations, particularly avoiding oscillatory representations of constant data.

2.1.3. Approximation of the differential operators

Let a linear differential operator L act on $u(\underline{x})$ in the following manner:

$$Lu(\underline{x}) = g(\underline{x}), \quad \underline{x} \in \mathbb{R}^d. \tag{2.5}$$

$$\tag{2.6}$$

Denoting $\underline{x}_c, c = \{1, 2, ..., N\}$ as a "center point" we start seeking an approximation to L using ansatz,

$$u(\underline{x}_c) = \sum_{k=1}^{N} c_k \underbrace{\phi(||\underline{x}_c - \underline{x}_k||)}_{:=\mathrm{A}_{(c,:)},\,\mathrm{A}\in\mathbb{R}^{\mathrm{N}\times\mathrm{N}}} . \tag{2.7}$$

Applying a differential operator L on (2.7) we obtain,

$$Lu(\underline{x}_c) = \sum_{k=1}^{N} c_k \underbrace{L\phi(||\underline{x}_c - \underline{x}_k||)}_{:=\mathrm{B}_{(c,:)},\,\mathrm{B}\in\mathbb{R}^{\mathrm{N}\times\mathrm{N}}} . \tag{2.8}$$

We have to make a few additional observations. Expressing (2.7) and (2.8) in a matrix-vector notation gives us:

$$\begin{aligned} \underline{f} &= A\underline{c}, \\ L\underline{f} &= B\underline{c}, \end{aligned}$$

in the same order. We observe that both equations use the same c. By plugging $c = A^{-1}f$ to the second equation, we get:

$$L\underline{f} = B\underline{c} = BA^{-1}\underline{f}.$$

Since $\underline{f}$ represents $u(x)|_{x=x_c}$, we can deduce that BA^{-1} has to represent the "behavior" of differential operator L. We can write:

$$D = BA^{-1}, \tag{2.9}$$

where D is a *global differentiation matrix*.

2.2. RBF-generated finite differences (RBF-FD)

The main idea behind localization of *RBF* using *Finite differences* is to calculate the solution $u(x)\Big|_{x=x_c}, \forall \underline{x}_c \in \Omega$ to a PDE with certain boundary conditions using neighboring points only, and not all the center points as it is done using *RBF-Direct*.

Definition 2.4 *Let n be a scalar value which represents a number of neighboring points around x_c with x_c included. In the further procedure we aim for $n \ll N$, where N is defined as the number of all scattered center points x_c.*

The general framework for approximating a differential operator in x_c remains the same as in section 2.1.3 (all definitions also apply here), the only difference is that we now deal with n instead of N scattered points.

We are on a mission to produce n weights specific for approximation of L in every x_c. These weights are organized in the *local differentiation matrix d*. Using proper indexing, the weights are then put into the *global differentiation matrix D* which is highly sparse and offers an opportunity to decrease the computational cost when solving $D\underline{f} = \underline{g}$.

Remark 2.5 *The local differentiation matrix in RBF-generated finite differences case lives in $\mathbb{R}^n$ i.e. it is a vector.*

2.2.1. FD weights

Analogously to the well known *Finite Difference method*—where a differential operator is discretized such that the approximation is accurate for the highest-possible level of-polynomials—we are going to construct an approximation of operator L to be accurate for the RBFs $\phi_k = \phi(||x - x_k||)$ which are involved in the stencil approximation. We refer to [7] and write:

$$L\phi(||x_c - x_k||) = \sum_{k=1}^{n} a_k \phi(||x - x_k||). \tag{2.10}$$

This gives us the following system of equations [5]:

$$\underbrace{\begin{pmatrix} \phi(||x_1 - x_1||) & \phi(||x_1 - x_2||) & \dots & \phi(||x_1 - x_n||) \\ \vdots & \ddots & \vdots & \\ \phi(||x_n - x_1||) & \phi(||x_n - x_x||) & \dots & \phi(||x_n - x_n||) \end{pmatrix}}_{\text{A}} \underbrace{\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}}_{\text{a}} = \underbrace{\begin{pmatrix} L\phi(||x_c - x_1||) \\ \vdots \\ L\phi(||x_c - x_n||) \end{pmatrix}}_{\text{z}}, \tag{2.11}$$

with dense matrix $A \in \mathbb{R}^{n \times n}$, weights $a \in \mathbb{R}^n$ and the approximated linear operator $z \in \mathbb{R}^n$. Matrix A and vector z are known, therefore, the *RBF-FD* weights a shall be $a = A^{-1}z$.

Remark 2.6 *We can use these weights for approximation of L in only 1 point (x_c). In order to produce all the weights—which would then describe the approximation of operator L in every center point—one has to solve N such small systems.*

It is common to make the approximation of L also accurate for a certain degree of polynomials in the same way and for the same reasons as indicated in chapter 2.1.2.

2.2.2. Global representation

For the purposes of further investigation, we want to produce a matrix D of size $N \times N$, where the generated weights a are going to be put to a proper row with the proper column index.

Definition 2.7 *We define Γ_c as a subset of global indices $\{1, 2, ..., N\}$ such that it contains indices of neighboring elements of x_c, $c = \{1, 2, ..., N\}$ including the index of x_c. Γ_c is usually obtained by a* kd-tree algorithm.

Definition 2.8 *Let $\tilde{a}_c \in \mathbb{R}^N$, $c = \{1, 2, ..., N\}$ be a global representation of $a_c \in \mathbb{R}^n$ for a corresponding center point x_c. That yields that $\tilde{a}_c$ is partially filled with zeros. The non-zero values are the weights $a_c(i)$, $i = \{1, 2, ..., n\}$ which are put to proper places $\tilde{a}_c(j)$, $j \in \Gamma_c$.*

With the use of latter two definitions, we can acquire the matrix D.

Definition 2.9 *Let $D \in \mathbb{R}^{N \times N}$ be a matrix into which the vector of weights a_c is filled row-wise according to center points x_c: $D(c, :) = a_c$. We shall denote this matrix as a* global differentiation matrix.

2.2.3. Solutions to PDEs

We consider the following PDE:

$$\begin{aligned} Lu(\underline{x}) &= g(\underline{x}), \quad \underline{x} \in \Omega \subset \mathbb{R}^d, \\ u(\underline{x}) &= b(\underline{x}), \quad \underline{x} \in \partial\Omega \subset \mathbb{R}^d. \end{aligned} \tag{2.12}$$

L acting on $u(\underline{x})$ is approximated by the differentiation matrix D acquired above. The remaining work to do is to impose the boundary conditions (Dirichlet in this example) to D by setting the D_{ck} which belong to the boundary to 1 and correcting the corresponding rhs components g_c to b_c. When this is set the solution $u(\underline{x}) \in \Omega$ can be obtained directly by solving

$$D\underline{f} = \underline{g}. \tag{2.13}$$

2.2.4. Properties

We are going to avoid any formalism in this part, since *RBF-FD* only serves us as a reference with which *RBF-PUM* is to be compared, but also due to the fact that some properties were obtained from numerical experiments.

As already mentioned before, the matrix D has a lot of zero elements (usually up to 99%) which is nice in terms of reduced computational costs and memory requirements. RBF-FD produces a matrix with Nn non-zero values, while *RBF-Direct* produces a dense D. Using sparse formats the memory requirement becomes $O(N)$ vs $O(N^2)$. Within convection problems (which is our case of interest), the computational cost of applying D to perform a time step (matrix-vector product) for *RBF-Direct* case is $O(N^2)$ being obviously much more than $Nn \propto O(N)$ within *RBF-FD*. It is true, however, one has to obtain N inverses to matrices A in order to obtain the stencils a_c which form D and the overall cost for it is $Nn^3 \propto O(N)$, but we treat this segment as a reusable preprocessing stage when convective PDEs using explicit time-stepping are approached.

The price for decreased computational cost is usually a lowered order of accuracy and *RBF-FD* is no exception. There are several empirical rates available. For example, [5] provides the (algebraic) convergence rate of approximately $O(\sqrt{n})$ for implementation of the shallow water equations on the sphere, which is not as good as the spectral $O(\sqrt{N})$ originating from *RBF-Direct* applied to the same problem [6]. A curious reader might want to look in [23], where the exact convergence properties were derived using Taylor expansion for $n \leq 6$. Furthermore, the authors have shown that the value of optimal ε does not depend on the distance between center points x_c.

2.3. Radial basis function partition of unity method (RBF-PUM)

The Partition of Unity method [1] has lately been adopted for purposes of localizing *RBF-Direct*. The underlying idea is to subdivide the global domain to many small subdomains and to approximate the solution on every subdomain separately in the *RBF-Direct* fashion. In order to produce a global approximation domain using the subdomains, we apply compactly supported functions on them and sum them up, i.e., we form the *partition of unity*. The result is a method which produces a sparse system matrix and in some cases preserves the good convergence properties of *RBF-Direct*.

2.3.1. Local interpolation

Let us consider a domain $\Omega \ni \underline{x}$ where $\underline{x}$ are scattered nodes. The local regions of it are found by forming an *open cover*.

Definition 2.10 *An open cover of Ω is a family of open sets $\{\Omega_j : j = \{1, 2, ..., M\}\}$ such that $\cup_{j=1}^{M} \Omega_j \supseteq \Omega$. Open cover is open since a union of open sets is open.*

Using the above definition, the local region is denoted as Ω_j. We will often relate to local regions as *patches*.

Remark 2.11 *The usual procedure of obtaining an open cover of Ω is by performing the closest neighbor search using the* kd-tree *algorithm.*

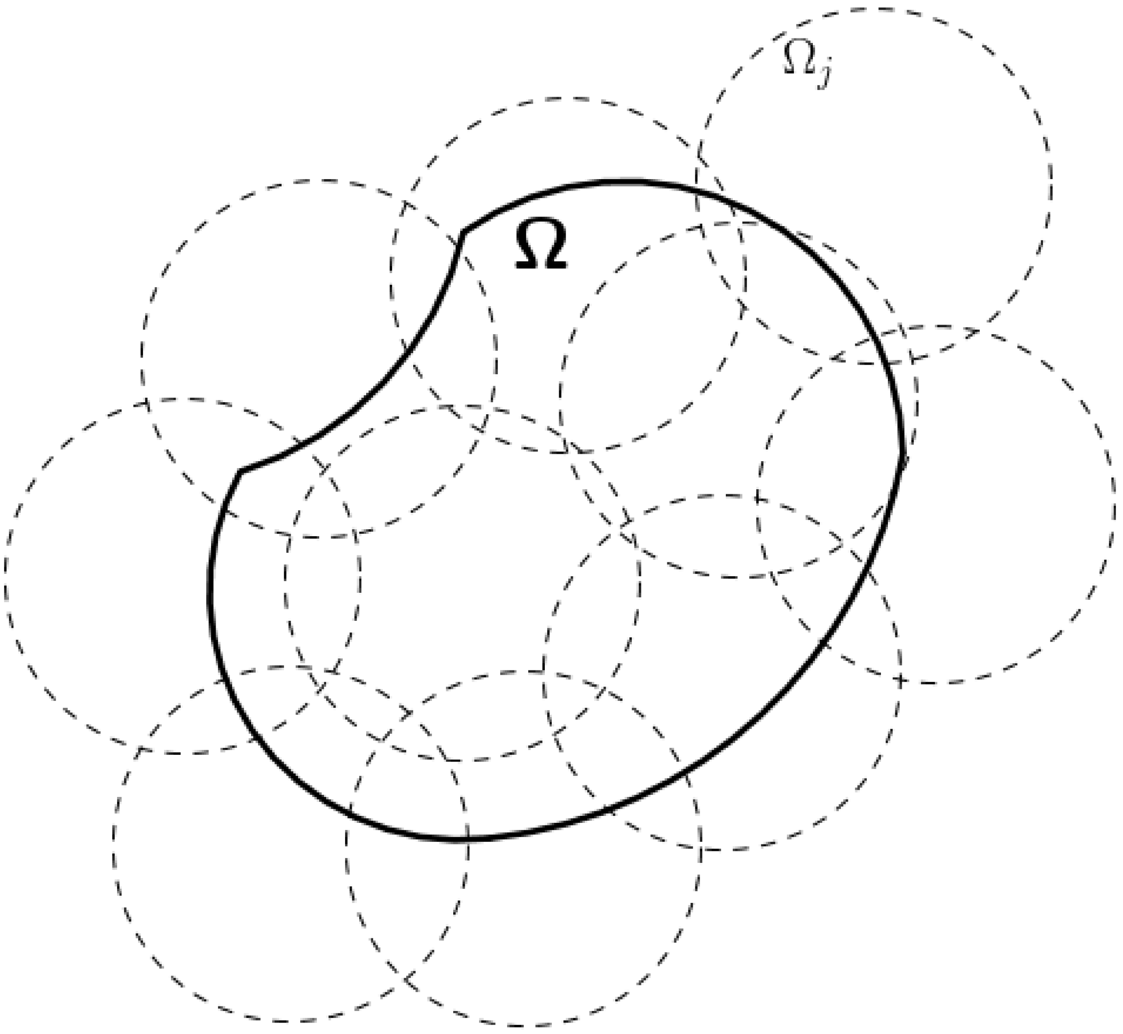

Figure 2.1.: $\cup_{j=1}^{M} \Omega_j$ forming an open cover to Ω.

Every patch Ω_j contains a set of $n_j = n^{(j)} + n_\epsilon^{(j)}$ scattered nodes, where $n_\epsilon^{(j)}$ represents the number of points which lie in the intersection $\Omega_j \cap \Omega_{j \neq i}, i, j = \{1, 2, ..., M\}$ and $n^{(j)}$ denotes the number of points having origin in only j-th patch ($\Omega_j \cap \Omega_{j \neq i} = \emptyset, i, j = \{1, 2, ..., M\}$). With respect to the global point of view Ω contains $N \gg n_j$ scattered nodes $X = \{\underline{x}_i\}_{i=1}^{N}$.

Remark 2.12 *Having a look at Figure 2.1 we notice that $n_\epsilon^{(j)} \to N$ gives us $\Omega_j \to \Omega$, meaning, the bigger overlap between patches brings smaller locality (more populated global matrices).*

To construct an interpolation on a single patch, we use (2.1) from *RBF-Direct* and write:

$$s_j(\underline{x}) = \sum_{k=1}^{n_j} c_k^{(j)} \phi(||\underline{x} - \underline{x}_k||), \quad \underline{x} \in \Omega_j. \tag{2.14}$$

We refer to $s_j(\underline{x})$ as a *local interpolant*. The equivalent matrix-vector representation reads:

$$A^{(j)} \underline{c}^{(j)} = \underline{s}^{(j)}, \tag{2.15}$$

where, similarly as in (2.2) but now in a local sense, $A_{ik}^{(j)} = \Phi(||\underline{x}_i - \underline{x}_k||)\Big|_{\{i,k\}=1}^{(n_j)}$, $\underline{c}^{(j)} \in \mathbb{R}^{n_j}$ and for interpolating data given as $f_j(\underline{x})$, $\underline{s}^{(j)} = \underline{f}^{(j)} \in \mathbb{R}^{n_j}$.

2.3.2. Towards global interpolation

We shall now use the local interpolants in such a way that they will together form a global interpolation,

$$s(\underline{x}) = \sum_{j=1}^{M} w_j(\underline{x}) s_j(\underline{x}),\ \underline{x} \in \Omega, \tag{2.16}$$

where M is the number of patches covering the whole domain, $s_j(\underline{x})$ is an interpolant defined on Ω_j and $w_j : \Omega_j \to \mathbb{R}$ are compactly supported C^k, $k \geq 0$ weight functions forming the *partition of unity* i.e.:

$$\sum_{j=1}^{M} w_j(\underline{x}) = 1,\ x \in \Omega. \tag{2.17}$$

Due to the properties of w_j pointed out above we observe that $w_j(\underline{x}_i) = 0, \forall \underline{x}_i \notin \Omega_j$ which gives a reduced number of summands in the (2.17) and (2.16),

$$\sum_{\underline{x}_i \in \Omega_j} w_j(\underline{x}_i) = 1. \tag{2.18}$$

The weight functions are often defined using Shepard's method [2],

$$w_j(\underline{x}) = \frac{\Phi_j(\underline{x})}{\sum_{k=1}^{M} \Phi_k(\underline{x})}, \tag{2.19}$$

where $\Phi_j(\underline{x})$ are compactly supported functions on Ω_j. We evaluate these functions such that:

$$\Phi\left(\frac{||\underline{x} - \underline{X}_j||}{R_j}\right) = \Phi(||\underline{x} - \underline{X}_j||), \tag{2.20}$$

where $\underline{X}_j$ is the center of j-th patch and R_j is the radius measured from $\underline{X}_j$ to the closure of Ω_j. That assures us they are compactly defined on Ω_j.

Remark 2.13 *$\Phi_j(\underline{x})$ can be arbitrarily chosen as long as they are positive, provide compact support and are b-times differentiable if we are going to approximate a linear operator L of order b.*

We can now indicate that using the partition of unity on a set of local interpolation problems the global approximation space can be formed. Using (2.16), (2.17) and the fact that the value $f(\underline{x}_i)$ is to be used within the local interpolant such that $s_j(\underline{x}_i) = f(\underline{x}_i)$, we write:

$$\begin{aligned} s(\underline{x}_i) &= \sum_{j=1}^{M} w_j(\underline{x}_i) s_j(\underline{x}_i) \\ &= \sum_{j=1}^{M} w_j(\underline{x}_i) f(\underline{x}_i) \\ &= f(\underline{x}_i) \sum_{j=1}^{M} w_j(\underline{x}_i) \\ &= f(\underline{x}_i) \cdot 1 = f(\underline{x}_i). \end{aligned}$$

The continuity of global solution is assured by letting $s_j(\underline{x}_k) = s_i(\underline{x}_k),\ \forall \underline{x} \in \Omega_j \cap \Omega_i$.

2.3.3. Solutions to PDEs

Similarly to the approach described in subsection 2.2.3 we consider the PDE (2.12) and look for an approximation $\tilde{u}(\underline{x})$ to solution $u(\underline{x})$ using similiar approach as in (2.16) for sets of known points $f(\underline{x})$ and $g(\underline{x})$. Using $s(\underline{x}_i) = f(\underline{x}_i), \quad \forall \underline{x}_i \in \Omega$ and $s(\underline{x}_i) = g(\underline{x}_i), \quad \forall \underline{x}_i \in \partial\Omega$ we write:

$$\begin{aligned} L\tilde{u}(\underline{x}_i) &= \sum_{j=1}^{M} L\Big[w_j(\underline{x}_i)\tilde{u}_j(\underline{x}_i)\Big] = f(\underline{x}_i),\ \forall \underline{x}_i \in \Omega, \\ \tilde{u}(\underline{x}_i) &= \sum_{j=1}^{M} w_j(\underline{x}_i)\tilde{u}_j(\underline{x}_i) = g(\underline{x}_i),\ \forall \underline{x}_i \in \partial\Omega, \end{aligned} \tag{2.21}$$

where $\tilde{u}_j(\underline{x}_i)$ denotes a local approximation on a patch $\Omega_j,\ j = \{1, .., n_j\}$ given in (2.14).

Remark 2.14 *One should keep in mind that application of L to $\tilde{u}(\underline{x}_i)$ has to be expanded using the* Leibniz rule. *For a quick demonstration let us consider $L = \Delta$. Then $\Delta\Big[w_j(\underline{x}_i)\tilde{u}_j(\underline{x}_i)\Big] = \Delta w_j(\underline{x}_i)\tilde{u}_j(\underline{x}_i) + 2\nabla w_j(\underline{x}_i) \cdot u_j(\underline{x}_i) + w_j(\underline{x}_i)\Delta u_j(\underline{x}_i)$.*

We introduce the matrix-vector notation of the equations above. Defining some constructs is inevitable.

Definition 2.15 *Let $P^{(j)} : \mathbb{R}^{n_j \times n_j} \to \mathbb{R}^{N \times N}$ be a local permutation operator for which it holds,*

$$\begin{aligned} \Big[P^{(j)}(A^{(j)})\Big]_{ik} &= 0, \quad \forall \{\underline{x}_i, \underline{x}_k\} \notin \Omega_j, \\ \Big[P^{(j)}(A^{(j)})\Big]_{ik} &= A^{(j)}_{lh}, \quad \forall \{\underline{x}_i, \underline{x}_k\} \in \Omega_j, \end{aligned} \tag{2.22}$$

where $A^{(j)}$ is a local matrix based on local approximation described in 2.15, $[i,k] = \{1, ..., N\},\ [l,h] = \{1, ..., n_j\},\ j = \{1, ..., M\}$.

By using this definition we can obtain the final global matrix D summing up the new $P^{(j)}$ matrices:

$$D = \sum_{i=1}^{M} P^{(j)}\Big(A^{(j)}\Big). \tag{2.23}$$

In other words, the permutation operator takes a local $n_j \times n_j$ matrix as an argument and creates a global $N \times N$ matrix initially filled with zeros. The local indeces $\{i, j\}$ are then mapped to the global ones $\{l, h\}$ such that relation between local and global approximation domain is maintained. Furthermore, the local elements that are mapped to the same $\{l, h\}$—that means that they are contained in more than one patch—are summed up. The concrete implementation of P depends on what kind of index ordering (for example snake-wise ordering) we choose and what kind of geometry are we dealing with (for example a sphere).

Definition 2.16 *Analogously to Definition 2.15, let* $p^{(j)} : \mathbb{R}^{n_j} \to \mathbb{R}^N$ *be again a local permutation operator being appropriate for obtaining global representation of local vectors usually used in global assembly of local right-hand-sides. Using a local vector* $\underline{f}^{(j)} = (f(\underline{x}_1), ..., f(\underline{x}_{n_j}))$, $j = \{1, ..., M\}$ *the following holds:*

$$\begin{aligned} p^{(j)}(f^{(j)})_i &= 0, \quad \forall \underline{x}_i \notin \Omega_j, \\ p^{(j)}(f^{(j)})_i &= (f^{(j)})_h, \quad \forall \underline{x}_i \in \Omega_j, \end{aligned} \tag{2.24}$$

where $i = \{1, ..., N\}$, $h = \{1, ..., n_j\}$, $j = \{1, ..., M\}$.

Using Definition 2.16, the final global vector $\underline{f}$ is then obtained by,

$$\underline{f} = \sum_{i=1}^{M} p^{(j)}\Big(\underline{f}^{(j)}\Big). \tag{2.25}$$

Definition 2.17 *Let the weight matrix* $W^{(j)} \in \mathbb{R}^{n_j \times n_j}$ *be defined such that its diagonal elements become* $W^{(j)}_{ii} = w_j(\underline{x}_i)$, $\underline{x}_i \in \Omega_j$, $i = \{1, ..., n_j\}$, $j = \{1, ..., M\}$.

Turning the focus back to the first equation of (2.21), we would like to find the global matrix-vector form in the spirit of the latter three definitions. Assuming that local matrices $A^{(j)}$ already exist and that $L(W^{(j)}A^{(j)})$ gives a matrix-based expansion of the linear operator L acting on this product (see Remark 2.14), we write:

$$D^L \underline{c} = \underline{f}, \tag{2.26}$$

where,

$$\begin{aligned} D^L &= \sum_{j=1,\, \underline{x} \in \Omega}^{M} P^{(j)}\Big[L(W^{(j)}A^{(j)})\Big] \\ \underline{c} &= (c_1, ..., c_N) \\ \underline{f} &= \sum_{j=1,\, \underline{x} \in \Omega}^{M} p^{(j)}\Big[\underline{f}^{(j)}\Big],\ \underline{f}^{(j)} = f(\underline{x}_i)\, \forall \underline{x}_i \in \Omega_j \end{aligned} \tag{2.27}$$

Similarly to this formulation, one can also construct a global linear representation for the second equation of (2.21), which satisfies the boundary conditions:

$$\begin{aligned} D &= \sum_{j=1,\, \underline{x} \in \partial\Omega}^{M} P^{(j)}\Big[W^{(j)}A^{(j)}\Big] \\ \underline{b} &= (b_1, ..., b_N) \\ \underline{g} &= \sum_{j=1,\, \underline{x} \in \partial\Omega}^{M} p^{(j)}\Big[\underline{g}^{(j)}\Big],\ \underline{g}^{(j)} = g(\underline{x}_i)\, \forall \underline{x}_i \in \partial\Omega_j. \end{aligned} \tag{2.28}$$

The matrix-vector equivalent of both equations is then given as,

$$\begin{pmatrix} D^L \\ \hline D \end{pmatrix} \begin{pmatrix} \underline{c} \\ \hline \underline{b} \end{pmatrix} = \begin{pmatrix} \underline{f} \\ \hline \underline{g} \end{pmatrix}. \tag{2.29}$$

We declare this system as the complete one in order to solve the PDE (2.16) using *RBF-PUM*.

2.3.4. Properties

Having a look Section 2.3.2 and Section 2.3.1 we notice that the involved local matrices are of size $n_j \times n_j$. It takes Mn_j^3 operations to assemble the global matrix using the local matrices with respect to (2.16) and (2.14). The assembly of local matrices is embarrasingly parallel and so is the matrix-vector multiplication required to perform one timestep when approaching time-dependent *PDEs* in the method of lines style.

Despite the much better computational complexity of *RBF-PUM* in comparison with *RBF-Direct*, the accuracy still looks promising. It was theoretically shown in [18] that the accuracy preserves the spectral scaling for the case when the number of patches M is fixed and the number of nodes N varies. However, the accuracy becomes high-order algebraic when the number of patches M varies and the number of nodes N is fixed.

Within *RBF-PUM* we still distinguish between center and evaluation points. That means that we can construct approximants for a certain number of center points, solve the resulting linear system and then pick an arbitrary number of evaluation points where we would like to evaulate the solution. This is not the case when using *RBF-FD* and we consider that as a drawback since one always has to solve the linear system if one wants to compute the solution in a point which is not the center point.

3. Parallel methods

3.1. Task-based parallelism

Our aim is to present and use a relatively novel way of constructing parallel programs on shared memory architectures. It is called *task-based parallelism*. A programmer is faced with an easier job developing the parallel code, especially when it comes to dealing with concurrency mechanisms i.e., analysing dependencies between statements and implementing the resulting memory lock/unlock schemes. As a consequence, one is given an opportunity to put less effort to scheduling issues and more effort to the problems of primary importance. The task-parallel framework used in this work is going to be *Superglue* [21].

3.1.1. The SuperGlue framework

SuperGlue [21] is a shared memory framework which uses data versioning for performing dependency-aware task-based parallelization. It is essential to present a brief motivation on why to use such frameworks as *SuperGlue* is.

Parallel programs are commonly constructed using the fork-join model of *OpenMP*, which works such that the user explicitly defines regions where the parallel work by several threads[1] is to be performed. The synchronization of threads is performed by barriers. The paper [14] discusses fork-join models applied for linear algebra problems, which are of our particular interest, and concludes that calling parallel subroutines from a sequential algorithm limits scalability. It suggests that implementing the algorithm as an explicit parallel code which calls sequential subroutines improves performance since it reduces the synchronization points. Furthermore, the same paper also concludes that when the order of computations is decided at runtime [21], this leads to better performance.

SuperGlue avoids the aforementioned bad scenarios because it allows the user to write the code without local synchronization points using prescheduled tasks which call sequential subroutines. The handles associated with tasks help the user to avoid performing the sometimes painful procedure of dependence analysis.

3.1.2. Tasks and handles

A task is an abstraction that describes a small subproblem of the problem, i.e., the dynamics that has to be performed in order to obtain a partial solution. It is up to the user to describe the task and it is crucial that one pays a full attention to that, also keeping in mind the granularity; in the spirit of task-based parallelism that means that one must not identify tasks which would work on either too small or too big amount of data (again, see the example below).

Example 3.1 *Let our problem be the matrix-vector product (*MVP*). We would like to find a certain number of subproblems that together describe the problem. We consider* $A\underline{x} = \underline{f}$, $A \in$

[1]When using the shared memory architecture, the number of threads usually represents the number of cores of the *CPU* performing the computations.

$\mathbb{R}^{n\times n}$, $\underline{x} \in \mathbb{R}^n$, $\underline{f} \in \mathbb{R}^n$. *Writing the product out for the first component of the result gives*

$$\underline{f}_1 = A_{1,1}x_1 + A_{1,2}x_2 + \ldots + A_{1,n}x_n,$$

or in general form

$$\underline{f}_i = A_{i,1}x_1 + \ldots + A_{i,n}x_n = \sum_{i=1}^{n} A_{i,1}x_i,$$

or with other words, the i-th component of an MVP *is a dot product* $A(i,:)\cdot\underline{x}(:)$. *We denote such a dot product as a subproblem of our* MVP *problem. It is legit to ask ourselves now whether we can step a level lower and find a subsubproblem by factorizing the subproblem.*

Let $A(i,:) = \underline{g}(:)$. *We immediately target two crucial operations in* $\underline{g}\cdot\underline{x}$. *These are multiplication of corresponding components of two vectors* $g_i x_i$ *and reduction-style summation of the resulting* n *products.*

We can denote these two operations as subsubproblems of an MVP. *In this particular case, usage of subsubproblems would yield too fine granularity which would cause overhead due to too many memory locks and unlocks. It would also cause* SuperGlue *to perform slower since dependence analysis would take much more time. It is therefore more appropriate to use several dot products as subproblems and describe one task per one subproblem.*

Within *SuperGlue* the tasks are submitted at runtime. Only one thread has the permission to access a certain task. The execution order is defined by the sequence in which the tasks are submitted and the internal dependency analysis, which is (from the user perspective) governed by *handles*. The user has to define handles in order to protect shared data maneuvered by the tasks. There exist three types of accesses that can be associated with handles: *read, write and add*. These describe the type of access to every crucial variable used in a task. We closely refer to [21] and write:

- The *read* type means that the task has to wait for all previous *write* or *add* accesses to finish. After that several tasks can read the same handle concurrently.
- The *write* type forces the task to wait for all previous accesses to finish before it can execute the write operation. The order of write accesses is fixed.
- The *add* type is a sloppier version of *write* type, since it performs exactly the same but the order of add accesses is not fixed. It can be for example beneficial to use when implementing the reduction-style tasks.

Remark 3.2 *If one is faced with a problem where it is known that there will not be any concurrent accesses to a variable, specification of the access type for it can be omitted. When the number of such handles tends towards a high number, this makes the framework run faster.*

Example 3.3 *In order to show how the tasks and handles are used, we present a hello-world example accompanied with a counter which counts how many times the phrase was printed. Every thread is going to print the "hello world" to the screen and at the same time increase the counter by 1. The nature of this problem is such that it can not be broken down into subproblems since the thread printing the string must also increase the value of variable. A reader can grasp how to produce a parallel program using* SuperGlue, *reading comments of the following code:*

```
#include "sg/superglue.hpp"
#include <iostream>

```

```
struct Options : public DefaultOptions<Options> {}; // We inherit the default options from the SG framework.

struct PrintAndIncreaseTask : public Task<Options> { // The task definition. Again, Options are inherited from SG.
  int *counter;
  // Constructor. We define handles and their types here.
  PrintAndIncreaseTask(int *counter_, Handle<Options> &handle_counter) : counter(counter_) {
    register_access(ReadWriteAdd::add, handle_counter); // We specify the add type for handle_counter.
  }
  // The dynamics of a task should always be described in the run() member. We are going to print and increment.
  void run() {
    std::cout << "hello world\n"; // Print.
    ++*counter; // Increase.
  }
};

int main(){ // Runtime.

  int *counter = new int; // We dynamically allocate 1 memory slot for the counter.
  Handle <Options> handle_counter; // We initialize the handle for counter.
  SuperGlue<Options> sg; // The team of worker threads is created here. By default number(threads)=number(cores).
  int n = 10; // Let the number of helloworld prints be n.

  // Therefore, we have to create n tasks.
  for (int i=0; i<n; i++)
    sg.submit(new PrintAndIncreaseTask(counter, handle_counter));

  sg.barrier(); // Explicit synchronization of threads.
  // Now we can print the counter. If everything is fine, then counter = n.
  std::cout << *counter;

  return 0;
}
```

Part II.

Implementation

4. RBF-PUM applied to the shallow-water equations

The goal of this chapter is to provide key steps that have to be performed in order to construct the *RBF-PUM* method for approximation of solutions to partial differential equations on the sphere. Such knowledge is usually not provided in the formulations like the ones given in the previous chapters, nevertheless, it plays an important role if one wants to—like we do—implement the computational framework from scratch.

4.1. RBF-PUM on the sphere

4.1.1. Properties of the open cover appropriate for partition of unity approach

The very first thing required is to form an open cover (*OC*) as defined in 2.10. The following question arises straight away; the *OC* is defined in the continuous space, but what do we have to be careful about when representing it on the computer (discrete space)? Let us construct an artificial *OC* consisting of two overlapping open subsets $\Omega := \Omega_1 \cap \Omega_2$. The intersection of two open subsets obviously exists when they share at least one member. Since the subsets are open, the members on their closure are not contained in them and that means the open subsets must have at least an infinitesimally small overlap in order to share members. But what about in the discrete sense? The same way of thinking applies, but one must take into account that there is no continuoity inside the subsets. That means the overlap must be big enough so that it contains at least one *discrete* member.

Furthermore, since our open cover lives in the discrete world, the question of whether the member is contained in an open subset if it lies on its closure is valid. The answer to it is positive since the machine precision ϵ_d is governing the quality of an open interval represented on the computer. In the case when the closure is defined by the radius of R, we can always argue that points contained in radius $R \pm \epsilon_d$ fall inside the subset.

Lastly, is the discrete open cover formed if the union of open subsets only contains all of the nodes which represent the domain we are discretizing (see Figure below)? The answer is again positive since such behaviour corresponds to the definition of an open cover. But the next question is whether such a discrete open cover is sufficient to perform the *RBF-PUM*, i.e., what happens if all of the nodal members are contained in the union but the union does not cover the whole domain in the continuous sense (see Figure 4.1, left image)? Let us look into equations (2.19, 2.20). The partition of unity weight functions are defined using the radius measured from the center of a patch to its closure. In the case when one wants to use the evaluation nodes that are placed outside of the closure of the discrete *OC*, but are contained inside the closure of the undiscretized domain, the weight functions would be undefined in such regions and the partition of unity would not be formed. We conclude that construction of the elementary discrete open cover is not enough. The patches must not only contain all of the center nodes such that they together form a union, but must also cover the whole space bounded by the domain (see Figure 4.1, right image).

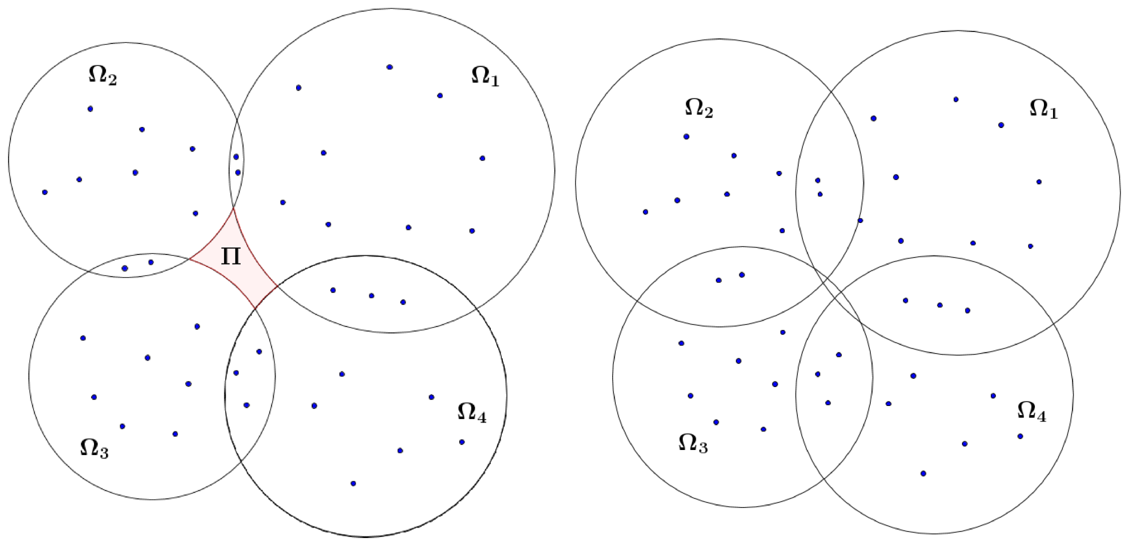

Figure 4.1.: Left: a failed attempt to form a discrete open cover. Right: a good discrete open cover for forming the partition of unity.

4.1.2. Patches

The elementary part of the discrete open cover are subsets (patches). In our approach, we define a patch on the sphere as a boundary of a small spherical cap in the form of the parametric equation.

Definition 4.1 *Let $\underline{x}_p = (a, b, d) \in \mathbb{S}^2$ be a center point of a spherical cap $\Gamma_p \subset \mathbb{S}^2$. Let Σ_p : $ax + by + dz = aT_1 + bT_2 + dT_3$ be a plane tangent to the point $\underline{x}_p$ where $\underline{T} = (T_1, T_2, T_3) = \underline{n} \cdot l$ is a point which lies on the base of a spherical cap, $\underline{n} = \frac{\underline{x}_p}{||\underline{x}_p||_2}$ is a normal direction to the point $\underline{x}_p$, $l = \sqrt{R^2 - r^2} \in (0, R)$ is the distance from origin of the sphere to the base of the spherical cap Γ_p, R is the unit radius of the sphere and r is the base circle radius of the spherical cap. Then the boundary $\partial\Gamma_p$ of the patch (spherical cap) is given by the intersection $\Sigma_p \cap \mathbb{S}^2$.*

Approaching the derivation of the curve $\partial\Gamma_p = \partial\Gamma_p(x, y, z)$ in the parametric form, one has to consider all combinations of components of $\underline{x}_p = (a, b, d)$ being zero or non-zero (except $(0, 0, 0)$). We restrict ourselves to computing three combinations (a, b, d), $(0, b, d)$, $(0, 0, d)$, where $\{a, b, d\} \neq 0$. Using these results we later extend the equations for $\partial\Gamma_p$ to the rest of four cases by substitution of variables.

The case of $\underline{x}_p = (a, b, d)$, $\{a, b, d\} \neq 0$.

Let $\underline{x}_p = (a, b, d)$, $\{a, b, d\} \neq 0$. Using the above definition, we write down the equation of the plane Σ_p as a function $\Sigma_p = \Sigma_p(y, z)$ defining $A = T_1 + \frac{b}{a}T_2 + \frac{d}{a}T_3$:

$$\Sigma_p(y, z) = x = A - \frac{1}{a}\Big(by + dz\Big) \tag{4.1}$$

By substituting the above $\Sigma_p(y, z) = x$ into the equation of the sphere $x^2 + y^2 + z^2 = R$ we enforce the $\Sigma_p \cap \mathbb{S}^2$ and obtain:

$$\Big(A - \frac{1}{a}\Big(by + dz\Big)\Big)^2 + y^2 + z^2 - R = 0 = F(y, z), \tag{4.2}$$

which we solve for y and obtain two solutions,

$$F_{1/2}(z) = \frac{aAb - bdz \mp \sqrt{-a^2\Big(-2aAdz + d^2z^2 + b^2(-R^2+z^2) + a^2(A^2 - R^2 + z^2\Big)}}{a^2+b^2}. \tag{4.3}$$

We can bound z by demanding real solutions to $F_{1/2}(z)$ i.e.:

$$-a^2\Big(-2aAdz + d^2z^2 + b^2(-R^2+z^2) + a^2(A^2 - R^2 + z^2\Big) \geq 0, \tag{4.4}$$

thus, $z \in [z_1, z_2]$, where

$$z_{1/2} = \frac{2aAd \mp \sqrt{4a^2A^2d^2 - 4(a^2+b^2+d^2)(a^2A^2 - a^2R^2 - b^2R^2)}}{2(a^2+b^2+d^2)}. \tag{4.5}$$

Remark 4.2 *The implementation on the computer should be done in a way that z is sampled as $z \in [z_1, z_2]$. After that y can be computed by concatenating $F_{1/2}(z)$ such that $y = [F_1(z); F_2(z)]$ and finally $x = F(y,z)$. The powers of the vectors (independent variables in non-discretized form) (x,y,z) have to be implemented as component-wise products: $z^2 \to (z \odot z)$, $y^2 \to (y \odot y)$, $x^2 \to (x \odot x)$.*

The case of $\underline{x}_p = (0,b,d),\ \{b,d\} \neq 0$.

We follow a procedure similiar to the previous case, but this time we set $a = 0$. This means that $ax = 0$ and therefore we obtain the plane defined as a function of (y,z): $\Sigma_p = \Sigma_p(y,z)$, which allows us to immediately obtain

$$\Sigma_p(z) = y = \frac{b^2\sqrt{-r^2+R^2} + d(d\sqrt{-r^2+R^2} - z\sqrt{b^2+d^2})}{b\sqrt{b^2+d^2}}. \tag{4.6}$$

By plugging it into the equation of the sphere—enforcing $\Sigma_p \cap \mathbb{S}^2$—we obtain $F(x,z) = 0$, which we solve for z and get

$$F(x) = \frac{b^2d\sqrt{-r^2+R^2} + d^3\sqrt{-r^2+R^2} - \sqrt{b^2(b^2+d^2)^2(r^2-x^2)}}{(b^2+d^2)^{(\frac{3}{2})}}. \tag{4.7}$$

As in the previous case for z, we now obtain the constraints for x by demanding the real solution:

$$x_{1/2} = \mp r. \tag{4.8}$$

Again, same Remark 4.2 also holds for this case, but now we evaluate vectors in the sequence x, z, y.

The case of $\underline{x}_p = (0,0,d),\ d = \mp r$

The most simple case is when we are facing only one non-zero component of $\underline{x}_p$. That gives us the following relation:

$$\Sigma_p = z = d. \tag{4.9}$$

We plug $z = d$ to the equation of the sphere and obtain $F(x,y) = 0$, which we solve for x and obtain,

$$x_{1/2} = \mp\sqrt{-y^2 - d^2 + R^2}. \tag{4.10}$$

by demanding the real solution, we obtain constraints on y,

$$y_{1/2} = \mp\sqrt{R^2 - d^2}. \tag{4.11}$$

Remark 4.2 holds for this case as well, but the vectors have to be evaluated in the sequence z, y, x.

All of the other cases.

We do not have to compute new expressions for the four cases which are left. We use a simplified approach by performing the proper substitution based on the three examples described above.

- Let $\underline{x}_p = (\tilde{a}, 0, \tilde{d})$, we seek for solution $\tilde{x}, \tilde{y}, \tilde{z}$. We perform substitutions $a = 0, b = \tilde{a}$ and $d = \tilde{d}$. Using the case of $\underline{x}_p = (0, b, d),\ \{b, d\} \neq 0$ to perform computations, the solution is $\tilde{x} = y,\ \tilde{y} = x,\ \tilde{z} = z$.
- Let $\underline{x}_p = (\tilde{a}, \tilde{b}, 0)$, we seek for solution $\tilde{x}, \tilde{y}, \tilde{z}$. We perform substitutions $a = 0, b = \tilde{b}$ and $d = \tilde{a}$. Using the case of $\underline{x}_p = (0, b, d),\ \{b, d\} \neq 0$ to perform computations, the solution is $\tilde{x} = z,\ \tilde{y} = y,\ \tilde{z} = x$.
- Let $\underline{x}_p = (\tilde{a}, 0, 0)$, we seek for solution $\tilde{x}, \tilde{y}, \tilde{z}$. We perform substitutions $a = 0, b = 0$ and $d = \tilde{a}$. Using the case of $\underline{x}_p = (0, 0, d),\ d \neq 0$ to perform computations, the solution is $\tilde{x} = z,\ \tilde{y} = x,\ \tilde{z} = y$.
- Let $\underline{x}_p = (0, \tilde{b}, 0)$, we seek for solution $\tilde{x}, \tilde{y}, \tilde{z}$. We perform substitutions $a = 0, b = 0$ and $d = \tilde{b}$. Using the case of $\underline{x}_p = (0, 0, d),\ d \neq 0$ to perform computations, the solution is $\tilde{x} = y,\ \tilde{y} = z,\ \tilde{z} = x$.

4.1.3. The discrete open cover

Now that we know what the parametric equation of the patches looks like, we can start to form the open cover. We present two approaches. The first one is based on [13] and the second one is constructed using the properties of the chosen nodes and performing the nearest neighbors search using the *kd-tree* algorithm.

We closely refer to [13] and write: let r be the radius of a patch, n the number of nodes in a patch, N a number of all nodes on the sphere, M a number of patches and q an overlap factor. Then M and r are computed as,

$$r \approx 2\sqrt{\frac{n}{N}} \tag{4.12}$$

$$M \approx \left\lceil q\frac{N}{n} \right\rceil. \tag{4.13}$$

An alternative approach is based on properties of quasi-uniformly distributed nodes. Taking into consideration the maximum determinant and minimum energy nodes, we notice that an arbitrary point x_k always has 6 or 7 neighbors that are located in the region with radius $\zeta_j + \delta_j,\ j = \{1, ..., M\}$, δ_j being some small value. In order to obtain $\zeta_j + \delta_j$ we have to find the maximal distance measured from 6 ($\underline{x}_i,\ i = \{1, ..., 6\}$) or 7 ($\underline{x}_i,\ i = \{1, ..., 7\}$) such neighbors to every center of the patch $x_k,\ k = 1, ..., M$: $\zeta_j + \delta_j = \max ||x_i - x_k||$. If we use this distance as the $r_j = \frac{\zeta_j + \delta_j}{2}$, the union between k-th and the neighboring patches is guaranteed. Unfortunately numerical experiments show that computing the maximal distance between every patch center and 6 neighboring centers does not form a proper discrete open cover as described at the begining of this section. However, using 7 neighboring centers has formed a correct open cover in all of the examined cases. The downside of using 7 neighbors is that the overlaps between adjacent patches are too big. We tackle this problem by observing the distance between 6 neigboring centers $\max ||d_6||$ and 7 neighboring centers $\max ||d_7||$ separately. Obtaining the radius of the j-th patch as a distance somewhere in between these two distances should make the overlaps smaller, but still form the discrete open cover appropriate for *RBF-PUM*. We write:

$$r^{(j)} = \max ||d_6|| + \omega(\max ||d_7|| - \max ||d_6||),\ \omega \in (0,1), \tag{4.14}$$

where $\max ||d_f|| := ||\underline{x}_i - \underline{x}_k||$, $i = \{1,..,f\}$, $k = \{1,...,M\}$ is the maximum distance measured from point $\underline{x}_k$ to its f closest neighbors $\underline{x}_i$. When calculating the radius as above, we consider $f = \{6, 7\}$. The proposed approach costs us $O(M \log(M))$ operations using the *kd-tree* algorithm. We consider that as acceptable since the bottleneck of *RBF-PUM* is based on obtaining the inverses of local interpolation matrices $A^{(j)}$ which costs $O(Mn^3)$ operations.

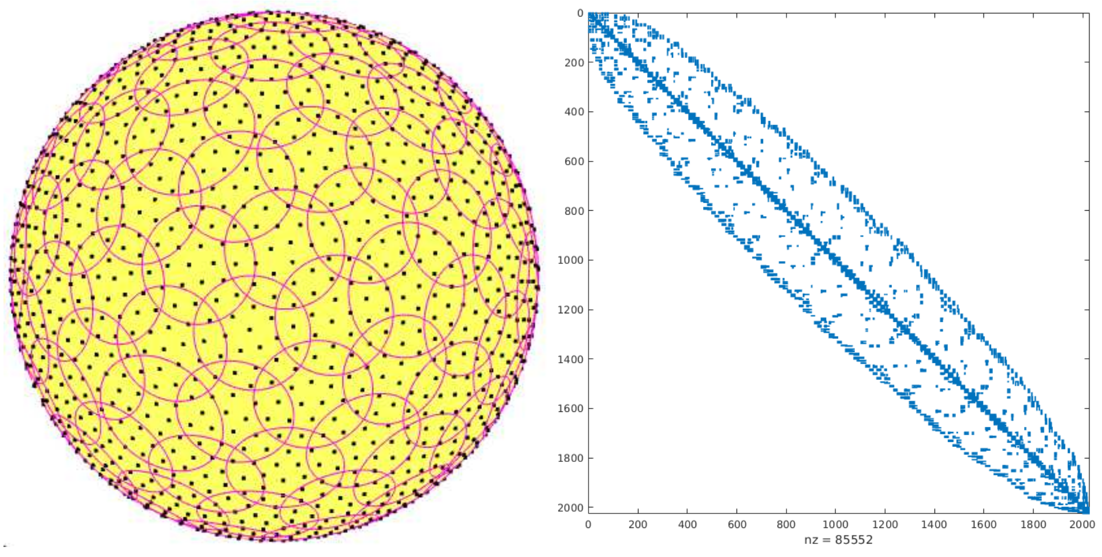

Figure 4.2.: Left: A discrete open cover of the unit sphere with $N = 2025$ maximum determinant nodes and $M = 128$ patches which are based on minimum energy center points. The open cover is based on the nearest neighbor search approach using $\omega = 0.5$. The sphere and the nodes (without patches) were drawn using the *spherepts* library [26]. Right: The corresponding global differentiation matrix which has 2.09% non-zero elements.

4.2. The continuous shallow water equations

We present the *RBF-PUM* based spatial discretization of the target model which we discuss in this work; the shallow water equations (SWE). Let $\underline{x} = (x, y, z)$ be an arbitrary location on the sphere, t denote the time, $\underline{u} = \underline{u}(\underline{x}, t) = (u_1(\underline{x}, t), u_2(\underline{x}, t), u_3(\underline{x}, t))$ represent the wind field, $h = h(\underline{x}, t)$ the geopotential height, $f = f(\underline{x})$ the Coriolis force and g the gravity. We then write down the *SWE* as follows,

$$\begin{aligned} \underline{u}_t &= -P\Big[(\underline{u} \cdot \underline{P}\nabla)\underline{u} + (\underline{x} \times \underline{u})f + g\underline{P}\nabla h)\Big] \\ h_t &= -\underline{P}\nabla \cdot (h\underline{u}), \end{aligned} \tag{4.15}$$

where the projection operator $\underline{P}$ [5] is added on top of the model and takes care that the computed gradients are the surface gradients on the sphere by projecting the gradient evaluated in $\underline{x}$ to the plane tangent to the sphere in $\underline{x}$. It is given as,

$$P = \begin{pmatrix} 1-x^2 & -xy & -xz \\ -xy & 1-y^2 & -yz \\ -xz & -yz & 1-z^2 \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}. \tag{4.16}$$

Before starting with the discretization, we have to expand the operators. Let $\nabla = (\nabla_x, \nabla_y, \nabla_z)$, where $\nabla_i = \frac{\partial}{\partial i}$. The first component of the matrix-vector product $\underline{P}\nabla$ then gives us

$$\Big[\underline{P}\nabla\Big]_{\vec{i}} = (1-x^2)\nabla_x - (xy)\nabla_y - (xz)\nabla_z = p_x \cdot \nabla. \tag{4.17}$$

Similarly to the first component, we also obtain the other two components of $\underline{P}\nabla$ and write $\underline{P}\nabla = (p_x \cdot \nabla, p_y \cdot \nabla, p_z \cdot \nabla)$, which we use when we expand the operator $(\underline{u} \cdot \underline{P}\nabla)$:

$$(\underline{u} \cdot \underline{P}\nabla) = u_1(p_x \cdot \nabla) + u_2(p_y \cdot \nabla) + u_3(p_z \cdot \nabla). \tag{4.18}$$

The $\vec{i}$ component of the advection operator $(\underline{u} \cdot \underline{P}\nabla)$ acting on $\underline{u}$ will then be:

$$\Big[(\underline{u} \cdot \underline{P}\nabla)\underline{u}\Big]_{\vec{i}} = u_1(p_x \cdot \nabla)u_1 + u_2(p_y \cdot \nabla)u_1 + u_3(p_z \cdot \nabla)u_1. \tag{4.19}$$

In the same way one can also write down the $\vec{j}$ and $\vec{k}$ components.

Focusing now on the second equation of (4.15), we expand the dot product $\underline{P}\nabla \cdot (h\underline{u}) = \underline{P}\nabla \cdot (hu_1, hu_2, hu_3)$ as follows:

$$\begin{aligned} \underline{P}\nabla \cdot (h\underline{u}) &= (p_x \cdot \nabla)(hu_1) + (p_y \cdot \nabla)(hu_2) + (p_z \cdot \nabla)(hu_1) \\ &= (p_x \cdot \nabla)hu_1 + h(p_x \cdot \nabla)u_1 + (p_y \cdot \nabla)hu_2 + h(p_y \cdot \nabla)u_2 + (p_z \cdot \nabla)hu_3 + h(p_z \cdot \nabla)u_3 \\ &= u_1(p_x \cdot \nabla)h + u_2(p_y \cdot \nabla)h + u_3(p_z \cdot \nabla)h + h\big((p_x \cdot \nabla)u_1 + (p_y \cdot \nabla)u_2 + (p_z \cdot \nabla)u_3\big) \\ &= (\underline{u} \cdot \underline{P}\nabla)h + h(\underline{P}\nabla \cdot \underline{u}). \end{aligned} \tag{4.20}$$

We are now prepared to write the fully expanded version of (4.15). Having a quick look at it we notice that all of the summands in the first equation are $\mathbb{R}^3$ vectors. We can therefore afford to break down the equation in 3 equations, each representing one spatial component. By performing that, our life is going to become easier when programming the discretized version of the system on the computer. We use (4.19, 4.17, 4.20) and write:

$$\begin{aligned} \begin{pmatrix} (u_1)_t \\ (u_2)_t \\ (u_3)_t \end{pmatrix} &= -\underline{P} \begin{pmatrix} u_1(p_x \cdot \nabla)u_1 + u_2(p_y \cdot \nabla)u_1 + u_3(p_z \cdot \nabla)u_1 + (\underline{x} \times \underline{u})_{\vec{i}} f + g(p_x \cdot \nabla)h \\ u_1(p_x \cdot \nabla)u_2 + u_2(p_y \cdot \nabla)u_2 + u_3(p_z \cdot \nabla)u_2 + (\underline{x} \times \underline{u})_{\vec{j}} f + g(p_y \cdot \nabla)h \\ u_1(p_x \cdot \nabla)u_3 + u_2(p_y \cdot \nabla)u_3 + u_3(p_z \cdot \nabla)u_3 + (\underline{x} \times \underline{u})_{\vec{k}} f + g(p_z \cdot \nabla)h \end{pmatrix} \\ h_t &= -\Big[u_1(p_x \cdot \nabla)h + u_2(p_y \cdot \nabla)h + u_3(p_z \cdot \nabla)h + h\big((p_x \cdot \nabla)u_1 + (p_y \cdot \nabla)u_2 + (p_z \cdot \nabla)u_3\big)\Big], \end{aligned} \tag{4.21}$$

and conclude that there are two operators which have to be discretized:

$$u_j(p_i \cdot \nabla),\; j = \{1, 2, 3\},\; i = \{x, y, z\},$$

and

$$(p_i \cdot \nabla),\; i = \{x, y, z\}.$$

Since they are similiar, we can at first discretize the second one and then extend it so that it will correspond to the first one.

4.3. The semi-discrete shallow water equations

Referring to (4.21) we have a look at the operator $(p_x \cdot \nabla)$ and apply it to the equation (2.16) in order to obtain the differentiation matrix which approximates ∇_x.

Let us expand the differential operator acting on the product using the Leibniz rule,

$$\begin{aligned} (p_x \cdot \nabla)s(\underline{x}) &= \sum_{j=1}^{M}(p_x \cdot \nabla)\big[w_j(\underline{x})s_j(\underline{x})\big] \\ &= \sum_{j=1}^{M}(p_x \cdot \nabla)w_j(\underline{x})s_j(\underline{x}) + w_j(\underline{x})(p_x \cdot \nabla)s_j(\underline{x})\big], \end{aligned} \tag{4.22}$$

where the expansion of $(p_x \cdot \nabla)$ is the same as in (4.17). Our current goal is to obtain the above equation in the matrix form. We start by constructing some useful definitions.

Definition 4.3 *Let us recall Definition 2.17. We extend $W^{(j)}$ to $W^{(j,d)}$, $d = \{x, y, z\}$ where its diagonal components are defined as $W^{(j,d)}_{ii}$, $i = \{1, ..n_j\} = \nabla_d w(\underline{x}_i)$. With other words, $W^{(j,x)}$ represents differentiatied weight function with respect to x, evaluated in the points which belong to the j-th patch.*

Definition 4.4 *Using (2.8) we at first extend $B \in \mathbb{R}^{N\times N}$ (the global derivative matrix) to $B^{(j)} \in \mathbb{R}^{n_j\times n_j}$ (the local derivative matrix), and then $B^{(j)}$ to $B^{(j,d)}$, where its elements $B^{(j,d)}_{i,k}$ are defined as $B^{(j,d)}_{i,k} = \nabla_d\Phi(||x^{(j)}_i - x^{(j)}_k||)$, $d = \{x, y, z\}$, $i = \{1, .., n_j\}$, $k = \{1, .., n_j\}$.*

The local interpolation matrix $A^{(j)}$ and the local permutation operator $R^{(j)}$ are given in Equation (2.15) and Definition 2.15 respectively. By referring to (2.9), one can deduce that the local differentiation matrix which approximates a local differential operator (4.18) is

$$(u \cdot \underline{P}\nabla)\Big|_{\Omega_j} = \Big(u_1(p_x \cdot \nabla) + u_2(p_y \cdot \nabla) + u_3(p_z \cdot \nabla)\Big)_{\Omega_j} \approx D^{(j)} = K^{(j)}A^{-1(j)}. \tag{4.23}$$

Note that $K^{(j)}$ is some local matrix operator, which expands the local differential operator $D^{(j)}$ as,

$$D^{(j)}A^{-1(j)} = \underbrace{\Big(\underline{u}^{(j)}_1 \odot K^{(j,x)} + \underline{u}^{(j)}_2 \odot K^{(j,y)} + \underline{u}^{(j)}_3 \odot K^{(j,z)}\Big)}_{K^{(j)}} A^{-1(j)}. \tag{4.24}$$

Here, written in generalized form, $\underline{u}_i = (u_i(\underline{x}_1), .., u_i(\underline{x}_{n_j}))$, $i = \{1, 2, 3\}$ is a vector representing the function $u_i : \mathbb{R}^3 \to \mathbb{R}$ evaluated in $\underline{x}_k$ and $K^{(j,d)}A^{-1(j)}$, $d = \{x, y, z\}$ is a local approximation to $(p_d \cdot \nabla)$, $i = \{1,2,3\}$, $d = \{x,y,z\}$. According to that, every discretized component of the projected gradient $\underline{P}\nabla$ can be obtained from:

$$(p_d \cdot \nabla)^{(j)} \approx K^{(j,d)} A^{-1(j)} = \Big(\operatorname{diag}(\underline{p}_{1d}) F^{j,x} + \operatorname{diag}(\underline{p}_{2d}) F^{j,y} + \operatorname{diag}(\underline{p}_{3d}) F^{j,z} \Big) A^{-1}, \quad (4.25)$$

where p_{id}, $i = \{1,2,3\}$ are components of $p_d, d = \{x,y,z\}$ from (4.16) and $F^{(j,d)}$ is the matrix-vector based expansion of ∇_d acting on the $w_j(\underline{x}) s_j(\underline{x})$ product in the same fashion as in (4.22). Using Definition 4.3 and Definition 4.4 we write:

$$F^{(j,d)} = W^{(j,d)} A^{(j)} + W^{(j)} B^{(j,d)}. \quad (4.26)$$

That brought us to the level where we can obtain the semi-discrete version of (4.21):

$$\begin{aligned} \begin{pmatrix} (u_1)_t \\ (u_2)_t \\ (u_3)_t \end{pmatrix} &= -\underline{P} \begin{pmatrix} D^{(j)}\underline{u}_1 + \underline{f} \odot (\underline{x} \times \underline{u})_{\vec{i}} + \underline{g} \odot K^{(j,x)} A^{-1(j)} \underline{h} \\ D^{(j)}\underline{u}_2 + \underline{f} \odot (\underline{x} \times \underline{u})_{\vec{j}} + \underline{g} \odot K^{(j,y)} A^{-1(j)} \underline{h} \\ D^{(j)}\underline{u}_3 + \underline{f} \odot (\underline{x} \times \underline{u})_{\vec{k}} + \underline{g} \odot K^{(j,z)} A^{-1(j)} \underline{h} \end{pmatrix} \\ h_t &= -\Big[D^{(j)}\underline{h} + \underline{h} \odot \Big(K^{(j,x)}\underline{u}_1 + K^{(j,y)}\underline{u}_2 + K^{(j,z)}\underline{u}_3 \Big) \Big], \end{aligned} \quad (4.27)$$

which is not final yet since models which are of pure hyperbolic nature—such as ours—are prune to instabilities when advanced in time by numerical explicit time-stepping algorithms which we aim to use in the further chapters.

4.3.1. The A^{-1} hyperviscosity approach

Semi-discretized schemes such as (4.27) lead to a system of ODEs written in general terms as $u_t = -Du$, where the discretized hyperbolic operator D may trigger high Fourier frequencies which are of numerical origin and have to be damped in order to obtain the stable propagation in time. To do that, the usual approach is to add an artificial diffusion of high order (hyperviscosity) to the right-hand-side of the equation using some high-order Laplacian Δ^k, which damps only the non-physical frequencies of D and leaves the physically relevant ones intact.

We choose to diverge from the classical approach using Δ^k, since a simpler approach was introduced in [10] which does not require a decision on which order of Laplacian k is the most appropriate one in order to get stability and at the same time does not make the nearly pure advection too damped. This approach has been succesfully tested on the sphere and is based on the inverse of *RBF* interpolation matrix.

Definition 4.5 *Let $A^{(j)}$ be a positive-definite local* RBF *interpolation matrix with elements $A^{(j)}_{i,k} = ||\underline{x}_i - \underline{x}_k||$, $\underline{x} \in \Omega_j \subset \mathbb{S}^2$. Recalling (2.19) and Definition 2.17 the local discrete hyperviscosity operator $H^{(j)}$ is given as $H^{(j)} = W^{(j)} A^{(j,-1)}$.*

We rewrite (4.27) by adding a hyperviscosity term, scaled by a very small factor of μ:

$$\begin{aligned} \begin{pmatrix} (u_1)_t \\ (u_2)_t \\ (u_3)_t \end{pmatrix} &= -\underline{P} \begin{pmatrix} D^{(j)}\underline{u}_1 + \underline{f} \odot (\underline{x} \times \underline{u})_{\vec{i}} + \underline{g} \odot K^{(j,x)} A^{-1(j)} \underline{h} \\ D^{(j)}\underline{u}_2 + \underline{f} \odot (\underline{x} \times \underline{u})_{\vec{j}} + \underline{g} \odot K^{(j,y)} A^{-1(j)} \underline{h} \\ D^{(j)}\underline{u}_3 + \underline{f} \odot (\underline{x} \times \underline{u})_{\vec{k}} + \underline{g} \odot K^{(j,z)} A^{-1(j)} \underline{h} \end{pmatrix} - \mu \begin{pmatrix} H^{(j)}\underline{u}_1 \\ H^{(j)}\underline{u}_2 \\ H^{(j)}\underline{u}_3 \end{pmatrix} \\ h_t &= -\Big[D^{(j)}\underline{h} + \underline{h} \odot \Big(K^{(j,x)}\underline{u}_1 + K^{(j,y)}\underline{u}_2 + K^{(j,z)}\underline{u}_3 \Big) \Big] - \mu H^{(j)}\underline{h}. \end{aligned} \quad (4.28)$$

It is noted in [10] that the magnitudes of the eigenvalues of A follow a pattern based on even powers of the shape parameter ε.

order of magnitude	$O(\varepsilon^0)$	$O(\varepsilon^2)$	$O(\varepsilon^4)$	$O(\varepsilon^6)$	$O(\varepsilon^8)$	...
number of eigenvalues	1	3	5	7	9	...

Let a magnitude of k-th $k = \{1, .., n\}$ eigenvalue of $A^{(j)} \in \mathbb{R}^{n\times n}$ be $|\lambda_k|$ and let $\varepsilon \to 0$. According to the table listed above, many $|\lambda_k|$ will be small, while few of them will be large. It can be shown that corresponding eigenvectors get more oscillatory when $|\lambda_k|$ decreases [10]. Now, considering the inverse, the eigenvalues of $A^{(j,-1)}$ are the inverses of $|\lambda_k|$: $|\gamma_k| = \frac{1}{|\lambda_k|}$ and furthermore, since $A^{(j)}$ is positive-definite, they are also positive. The pattern that holds for $|\gamma_k|$ is then —

order of magnitude	$O(\frac{1}{\varepsilon^0})$	$O(\frac{1}{\varepsilon^2})$	$O(\frac{1}{\varepsilon^4})$	$O(\frac{1}{\varepsilon^6})$	$O(\frac{1}{\varepsilon^8})$	...
number of eigenvalues	1	3	5	7	9	...

—which means that for $\varepsilon \to 0$, there will be many large eigenvalues and few of them which will be small. The same pattern, given in the table, will also hold for how oscillatory the corresponding eigenvectors are. Therefore, by adding a term $-\mu H^{(j)}\underline{u}$ to the right-hand-side of the convective PDE $u_t = -Du$ we leave all of the physically relevant modes intact but suppress nearly all of the high-oscillatory modes. Experiments showing that were performed in [10].

4.3.2. Observations on the overlap of patches

Problems of the hyperbolic nature such as the shallow-water equations are, demand a special attention with respect to the characteristics of the wind field. The numerical scheme based on the explicit time-stepping and a certain spatial discretization, has to work in a way to not approximate the solution advanced in time based on the information from the opposite direction of the wind field. When using *RBF-PUM* in combination with explicit time-stepping methods, one has to be aware of that since the open cover of the domain Ω can be obtained using a mild overlap of patches Ω_j, which, as we argue in this section, yields a non-sufficient wind support.

Wind support criteria

Numerical experiments specifically for *RBF-PUM* applied to the shallow-water equations show that we are able to get a sufficient wind support (and by that stability) only when the overlap factor q (average number of patches to which a single node belongs) is sufficiently large, that is, when the definition below applies.

Definition 4.6 *Let Q_j, $j = \{1, .., M\}$ be a set of neighbouring patches to a patch Ω_j. Let r be the number of patches that belong to Q_j and Q_{jk}, $k = \{1, .., r\}$ a single patch k from the neighbourhood j. Then the condition for the sufficient wind support of* RBF-PUM *applied to a hyperbolic PDE on a sphere in combination with explicit time-stepping is,*

$$\bigcap_{k=1}^{r} Q_{jk} \neq \emptyset \quad \forall j. \tag{4.29}$$

Having been put on the surface of a sphere, we can not say when this requirement is fulfilled due to non-uniformity of the nodes, however, we notice that when using *Minimum Energy nodes*, every node has either $r = 6$ or $r = 7$ neighbours with a quasi-uniform distance. This means the overlap requirement will start to hold when every node inside Ω_j will be—on average—also contained inside more than $q = 3$ patches from the corresponding set of neighboring patches Q_j. To be on the safe side, we use $q = 4$ and $q = 3.807$ for our numerical experiments.

Reasoning for the wind support criteria

Reasoning for the overlap requirement can be given from the down- and up-wind perspective. Considering an open cover of domain Ω with a mild overlap of patches Ω_j, some of the nodes inside Ω_j get a down-wind support and some of them get up-wind support. This does not lead to stability, since we would like to have a uniform wind support all over the domain. In case when the overlap between $\Omega_j,\ j = \{1, .., M\}$ is sufficiently large (Definition 4.6), we can not talk about diverse wind support anymore, since there are no nodes which would only be supported by a single patch. That gives a globally connected domain and by that a unique wind support.

RBF-PUM compared to *RBF-FD*

Since our goal is to compare *RBF-PUM* and *RBF-FD* implementations between each other, we present an analogy between them in the spirit of wind support criteria which is—in contrast to *RBF-PUM*—naturally imposed in *RBF-FD* and also the standard *Finite difference method*.

In certain cases, every patch can be looked upon as a stencil, since similarly to *Finite difference method*, a derivative in one point is calculated upon its neighbors which are members of a stencil or analogously to our case, members of a patch. There is, however, one very important difference between a stencil and a patch. According to Taylor series expansion, with other words considering FD method using polynomial basis functions, a stencil is naturally derived such that a derivative in k-th node is always approximated based on a weighted value of $k-1$-th node, $k-1$-th node is always approximated based on a weighted value of $k-2$-nd node and so on. Considering the one-dimensional case and a stencil of size n this means that consecutive rows in a differentiation matrix D will be shifted for $\lceil \frac{n}{2} \rceil$ columns, making D a skew-symmetric matrix. By that we also notice that a derivative in a node k is based upon $\lceil \frac{n}{2} \rceil$ number of nodes in a stencil that approximated a derivative in a node $k-1$.

Considering a single non-overlapped patch Ω_j and its arbitrary point $\underline{x} \in \Omega_j$, we approximate a derivative in $\underline{x}$ based on all members of Ω_j including $\underline{x}$ itself. Let us now construct a small overlap with Ω_{j-1} crossing Ω_j on the left and Ω_{j+1} crossing Ω_j on the right, where $\Omega_{j-1} \cap \Omega_{j+1} = \emptyset$. We again look at $\underline{x} \in \Omega_j$ and again notice that a derivative in $\underline{x}$ is based on all members contained in Ω_j, however, since $\Omega_j \cap \Omega_{j-1} \neq \emptyset$ and at the same time $\Omega_j \cap \Omega_{j+1} \neq \emptyset$ we observe that a derivative in $\underline{x} \in \Omega_j$ is also approximated based on information from Ω_{j-1} and Ω_{j+1}. This brings us to the point where we can deduce that drawing an analogy between *Finite Differences* and *RBF-PUM* can help us in understanding the behavior of *RBF-PUM*. The key difference between them when it comes to overlapping of stencils or overlapping of patches is that we have an ability to form an arbitrary overlap of Ω_j and its neighboring patches, while, as already denoted above, neighboring *RBF-FD* stencils have a natural overlap of $\lceil \frac{n}{2} \rceil$. That is the reason there are no stability problems with respect to the wind direction when *RBF-FD* is used.

The overlap within *RBF-FD* is acknowledged during weight calculation, which has no influence on the runtime (matrix-vector multiplications) when an explicit time-stepping method is applied to the semi-discretization. On the other hand, the overlapping for *RBF-PUM* is significant during the runtime of explicit time-stepping since we continuously compute local results for each patch separately and then average them in the partition of unity sense in order to get a global representation. That means the cost of *RBF-PUM* in comparison with *RBF-FD* is expected to be q-times larger when the setup is based on explicit time-stepping. Taking into account the stability criteria presented above, this means the cost is $q > 3$-times larger.

4.4. Linearized shallow water equations

In order to perform stability tests we write down the linearized shallow-water equations around the initial condition by referring to [6]. The approximate solutions $\hat{\underline{u}}$ and $\hat{h}$ are given as:

$$\begin{aligned}\hat{\underline{u}} &= \underline{u}_{init} + \delta \underline{u}_f + O(\delta^2)\\ \hat{h} &= h_{init} + \delta h_f + O(\delta^2),\end{aligned}$$

where $\underline{u}_{init}$, h_{init} are initial conditions and $\underline{u}_f$, h_f are the intermediate solutions in time. We can plug in this expansion to (4.21) and obtain the linearized system:

$$\begin{aligned}\begin{pmatrix}(u_{f,1})_t\\(u_{f,2})_t\\(u_{f,3})_t\end{pmatrix} &= -\underline{P}\begin{pmatrix}\underline{u}_f\cdot\underline{P}\nabla u_{init,1}+\underline{u}_{init}\cdot\underline{P}\nabla u_{f,1}+(\underline{x}\times\underline{u}_f)_{\vec{i}}f+g(p_x\cdot\nabla)h\\ \underline{u}_f\cdot\underline{P}\nabla u_{init,2}+\underline{u}_{init}\cdot\underline{P}\nabla u_{f,2}+(\underline{x}\times\underline{u}_f)_{\vec{j}}f+g(p_y\cdot\nabla)h\\ \underline{u}_f\cdot\underline{P}\nabla u_{init,3}+\underline{u}_{init}\cdot\underline{P}\nabla u_{f,3}+(\underline{x}\times\underline{u}_f)_{\vec{k}}f+g(p_z\cdot\nabla)h\end{pmatrix}\\ (h_f)_t &= -\big[\underline{u}_f\cdot\underline{P}\nabla h_{init}+\underline{u}_{init}\cdot\underline{P}\nabla h_f+h_f\underline{P}\nabla\cdot\underline{u}_{init}+h_{init}\underline{P}\nabla\cdot\underline{u}_f\big].\end{aligned} \tag{4.30}$$

Remark 4.7 *Since we now talk about a linear system, one can discretize the above system of equations and express it in a matrix format in order to obtain the eigenvalues of a single linear right-hand-side differential operator acting on $\underline{u}_f$ and h_f.*

4.5. Sequential algorithm

In order to formulate (4.27) on the computer, one must also take care of the discretization in time. We choose to combine the spatial discretization with a well known explicit time-stepping method, the 4th order Runge-Kutta method. Putting the focus on the most expensive routines of the program, we use the integrated profiler of Matlab and observe that the time-stepping algorithm takes over 80% of the runtime needed to perform the simulation for 400 timesteps. Furthermore, inside *RK4*, the most costly routine is the one which evaluates the right-hand-side of the semi-discrete problem $u_t = -Du$. We denote it by *evalRHS*. This brings a motivation to divide the algorithm into two parts. The preprocessing steps (part one, Figure 4.3) produce the initial conditions, reordering of nodes, patch structure and *RBF-PUM* differentiation matrices. The second part (Figure 4.4) is the time-stepping algorithm.

```
initialize input data: N, n, M, nodes(N), nodes(M)

nodes(M) <- Reorder nodes(M) (patch center points) such that they correspond ↩
    to snake-ordering on the sphere.

for $j = 1$ to $M$
  $\partial\Gamma_p$ <- generate a boundary of a spherical cap (patch closure)
  patches[j].members <- find nodes(N) which are bounded by $\partial\Gamma_p$, store global ↩
      indeces
end
nodes(N) <- Reorder nodes(N) (domain nodes) such that nodes contained in one ↩
    patch are put as tightly together as possible.

for $j = 1$ to $M$
  patches[j].members <- find new global indeces based on nodes(N) reorder
  patches[j].{Dx, Dy, Dz} <- calculate PDE operators based on RBF-PUM ↩
      differentiation matrices
  patches[j].{u1, u2, u3, h} <- find local members of initial condition ↩
      vectors {u1, u2, u3, h}
end
```

Figure 4.3.: The preprocessing part of the algorithm.

```
initialize dt, steps, mu, initial conditions={u1, u2, u3, h}
for $t = 1$ to steps
  [k1(u1), k1(u2), k1(u3), k1(h)] <- evalRHS(Sol(:))
  [Sol1(u1), Sol1(u2), Sol1(u3), Sol1(h)] <- arraySum(Sol(:), dt/2 * k1(:))

  [k2(u1), k2(u2), k2(u3), k2(h)] <- evalRHS(Sol1(:))
  [Sol2(u1), Sol2(u2), Sol2(u3), Sol2(h)] <- arraySum(Sol(:), dt/2 * k2(:))

  [k3(u1), k3(u2), k3(u3), k3(h)] <- evalRHS(Sol2(:))
  [Sol3(u1), Sol3(u2), Sol3(u3), Sol3(h)] <- arraySum(Sol(:), dt * k3(:))

  [k4(u1), k4(u2), k4(u3), k4(h)] <- evalRHS(Sol3(:))

  [Sol(u1), Sol(u2), Sol(u3), Sol(h)] <- arraySumRK4(Sol(:), k1(:)*dt/6, 2*k2↩
     (:)*dt/6, 2*k3(:)*dt/6, k4(:)*dt/6)
end
```

Figure 4.4.: The time-stepping part of the algorithm.

5. RBF-PUM in a parallel setup

5.1. Towards the parallel algorithm

Having the sequential algorithm already divided into two sections runtime-wise, we decide to parallelize the most expensive part: time-stepping. By increasing the number of time steps, the execution time of the preprocessing part does not increase since it is independent of number of time steps. Enforcing parallelism on it would therefore bring a much smaller effect in comparison when parallelism is enforced on the time-stepping.

We are confronted with a question whether the time-stepping part is parallelizable at all, i.e., whether we can find a sufficient number of variables which can be at least to some extent treated independently thread-wise.

5.1.1. The idea

A look at the dependency graph in Figure 5.1 unfortunately tells us that every node posesses a dependency with its neighboring node and furthermore, every $k_i,\ i = \{1, 2, 3, 4\}$ also has a dependency with the final statement h. Keeping such structure as it is means that our algorithm is not parallelizable, at least not on the level of *RK4* intermediate time steps and full time steps to which we refer as *Level 0*. This can partially be avoided by dividing the global vectors into global blocks. Then, the same graph will not anymore apply to the whole, global data structure, but to the i-th block instead, which can—with the choice of good scheduling—result in parallelism also being possible to perform on the *Level 0*. We conclude that *RK4* is data-parallel.

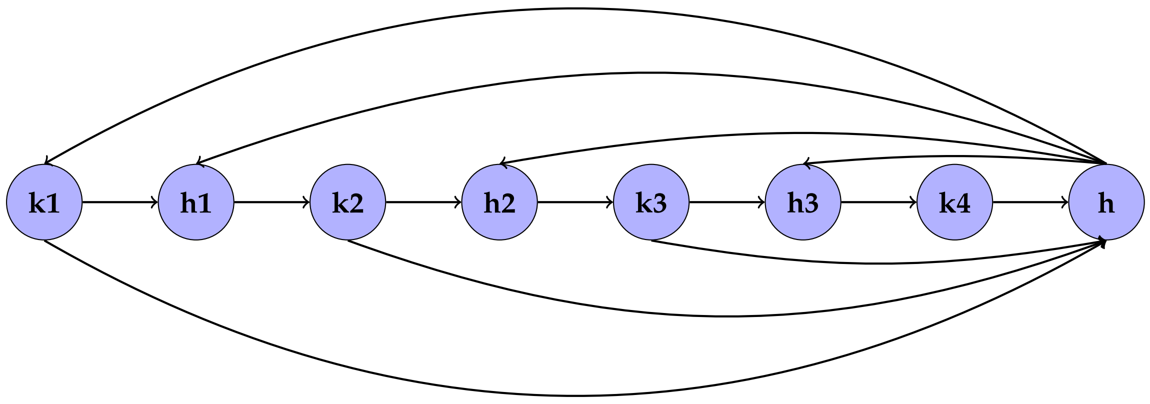

Figure 5.1.: The dependency graph of the 4th order Runge-Kutta method briefly described in Figure 4.3 on lines (18-31).

We exploit the property of *RBF-PUM* which allows us to keep the data structures local when approaching problems using explicit time-stepping schemes, avoiding the set up of the global differentiation matrix (i.e. solving a problem $\frac{\partial u}{\partial t} = -Du$) and also avoiding the sparse matrix-vector product which is known to be heavily bounded in performance by the memory bandwidth. Instead, the problem reads $\frac{\partial u}{\partial t} = -p\big[D^j u\big]$ where several local and dense matrix-vector products are performed independently on the level of patches and gathered via the permutation operator p (given in Definition 2.16) to the global right-hand-side vector. When propagating the solution in time, the algorithm for obtaining the *RHS* roughly behaves as a scatter-MVP-gather combination. We refer to the independent matrix-vector products on the levels of patches as a parallelism on *Level 1*.

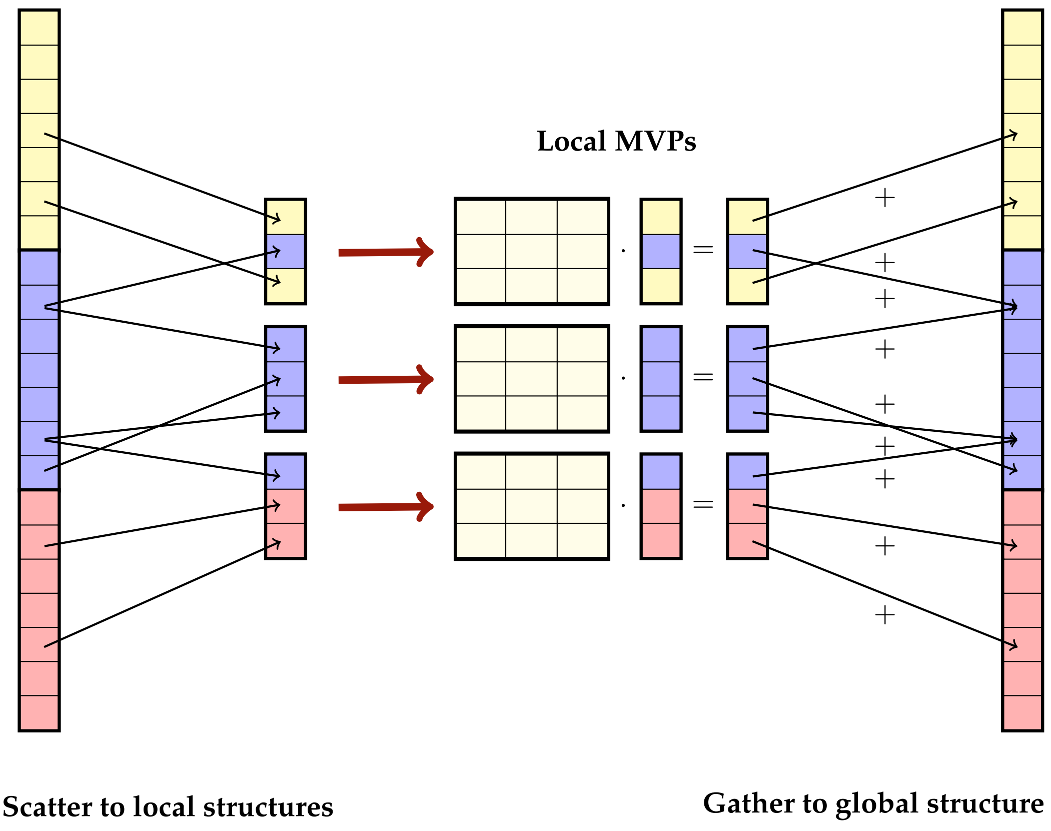

Figure 5.2.: A closer view to the evalRHS routine: one global vector, divided to $blocksize = 3$ blocks, is scattered to patches $\Omega_{\{1,2,3\}}$, more precisely either to $u^{(local)}_{1,i,in}$, $u^{(local)}_{2,i,in}$, $u^{(local)}_{3,i,in}$ or $h^{(local)}_{i,in}$, presented in Section 5.3. These are then used within local (patch level) matrix-vector products with local differentiation matrices. The local result is written to $u^{(local)}_{1,out}$, $u^{(local)}_{2,out}$, $u^{(local)}_{3,out}$ or $h^{(local)}_{out}$. At the end the result is gathered to the same vector where the input data was read from.

5.2. Identification of tasks

As soon as the data is scattered to patches, the local matrix-vector products become independent of each other. The expected runtime of scatter and gather routines is much lower than of the *MVP* routines, so instead of putting the scatter-mvp-gather work in one task, we choose the natural decomposition in three tasks. This way, more scatter and gather routines will be able to run in parallel since their execution will not be contained in the big runtime of the *MVP* routine. With other words, density of tasks will become higher which results in more work being available for threads to be performed in parallel.

Having a look at Figures 5.2 and 4.1, the members of a patch are usually (to some extent) clustered inside the global vector. Defining blocks of the global vector should therefore increase the density of tasks handled by a thread, since the access restriction (dependency) will be put on a substantially smaller amount of data, letting some other threads access some other blocks in parallel.

Based on that we present the following tasks:

- **blasMagic:** performs the local matrix-vector products for a group of patches,
- **scatterRHS:** scatters data from blocks of a global vector to the proper local vectors stored in the patch structure,
- **gatherRHS:** gathers data from a group of local result vectors stored in the patches structure to the proper blocks of the global vector with the addition operation,
- **arraySum:** sums up two blocks of two global vectors,
- **arraySum_RK4:** performs the final *RK4* block-wise summation of several global vectors.

Such decomposition of tasks should result (as mentioned before) in parallelism being effective on two levels:

- **Level 0:** parallelism due to calling scatterRHS, gatherRHS, arraySum and arraySum_RK4 routines on several blocks of the global arrays,
- **Level 1:** parallelism due to calling blasMagic on groups of patches.

According to Figure 5.2, *level 1* contains dependencies based on scatterRHS and gatherRHS. On the other hand, the involved dependencies on *level 0* are all of the tasks listed above. Since there are less dependencies on *level 1* and since it is expected that blasMagic routine is the longest task (in terms of runtime) among all tasks, we can assume that the biggest contribution towards parallelism is to be obtained within this level. The contribution of *level 0* should be visible in the overlap between calculations of (at least) two timesteps.

5.3. Identification of dependencies

For every task presented above we define handles and assign them data which they protect, along with the access type.

We firstly describe the index sets used for handling the data structure:

- $g = \{1, .., 4\}$ denotes wind field components and geopotential height: u_1, u_2, u_3 and h respectively,
- $i = \{1, .., 4\}$ denotes the temporal *Runge-Kutta 4* evaluation of the right-hand-side k_i,
- $f = \{1, .., 3\}$ denotes the temporal *Runge-Kutta 4* solution Sol_f,
- $r = \{1, .., blocks\}$ denotes a single block of any global array.

Then we use these in the following description of tasks and handles:

- **scatterRHS:**
  - read: for $g = \{1, .., 4\}$, $r = \{1, .., blocks\}$: $Sol[g][r]$,
  - add: for every patch in a patch group: $u_{1,k,in}^{(local)}$, $u_{2,k,in}^{(local)}$, $u_{3,k,in}^{(local)}$, $h_{k,in}^{(local)}$.
- **blasMagic:**
  - read: for every patch in a patch group and for $i = \{1, .., 4\}$: $u_{1,i,in}^{(local)}$, $u_{2,i,in}^{(local)}$, $u_{3,i,in}^{(local)}$, $h_{i,in}^{(local)}$,
  - add: for every patch in a patch group and for $i = \{1, .., 4\}$: $u_{1,i,calctmp}^{(local)}$, $u_{2,i,calctmp}^{(local)}$, $u_{3,i,calctmp}^{(local)}$, $h_{i,calctmp}^{(local)}$, $u_{1,i,calc}^{(local)}$, $u_{2,i,calc}^{(local)}$, $u_{3,i,calc}^{(local)}$, $h_{i,calc}^{(local)}$.
- **gatherRHS:**
  - read: for every patch in a patch group and for $i = \{1, .., 4\}$: $u_{1,i,calc}^{(local)}$, $u_{2,i,calc}^{(local)}$, $u_{3,i,calc}^{(local)}$, $h_{i,calc}^{(local)}$,
  - write: for $g = \{1, .., 4\}$, $i = \{1, .., 4\}$ $r = \{1, .., blocks\}$: $k[g][i][r]$.
- **arraySum:**
  - read: for $g = \{1, .., 4\}$, $i = \{1, .., 4\}$, $r = \{1, .., blocks\}$: $k[g][i][r]$, $Sol[g][r]$,
  - add: for $g = \{1, .., 4\}$, $f = \{1, .., 3\}$, $r = \{1, .., blocks\}$: $Sol_{tmp}[g][f][r]$.
- **arraySumRK4:**
  - read: for $g = \{1, .., 4\}$, $i = \{1, .., 4\}$, $r = \{1, .., blocks\}$: $k[g][i][r]$,
  - write: for $g = \{1, .., 4\}$, $r = \{1, .., blocks\}$: $Sol[g][r]$.

As seen from the list, we have to avoid the commutative-type *add* in *gatherRHS* and *arraySum* in order to keep the algorithm correct. All of the other write operations can be done independent of the order using the access *add*.

5.4. Strategies for performing local matrix-vector products

Matrix-vector products for evaluating the right-hand-side of the generalized problem $u_t = -Du$ can in our case (4.28) be performed:

- in the most straightforward way i.e. in 16 evaluations of MVP (4 equations, 4 MVPs per equation),
- in a dense form. By that they can be evaluated in a certain amount of matrix-matrix products. For example, we can transform 16 MVPs to 4 MMPs without increasing computational complexity and introducing additional sparsity.

The dense form can bring a better cache performance since more variables can be reused inside one evaluation of an MMP. The equation (5.1) shows how one such MMP is performed in order to obtain the gradient of wind field $\underline{u}$ and geopotential height h in the x-direction.

$$\underbrace{\begin{pmatrix} D_x \end{pmatrix}}_{\in\mathbb{R}^{n\times n}} \underbrace{\begin{pmatrix} u_1^T & u_2^T & u_3^T & h^T \end{pmatrix}}_{\in\mathbb{R}^{n\times 4}} = \underbrace{\begin{pmatrix} D_x u_1^T & D_x u_2^T & D_x u_3^T & D_x h^T \end{pmatrix}}_{\in\mathbb{R}^{n\times 4}} \tag{5.1}$$

The second decision that has to be made is what kind of routine to use for performing MVPs or MMPs. There are several linear algebra libraries available. We decided to test *BLAS* since it is known to be the most robust one, however, it is lacking optimization features based on the newest computer architectures. This is the reason why we also chose to look at *Intel MKL*, which already exploits modern vectorization resources such as *AVX*.

Part III.

Results

6. Numerical results

Specifying a configuration of the constructed numerical scheme, we validate our implementation of the *RBF-PUM* framework applied to the shallow-water equations on the sphere by performing differentiation on the surface of the sphere and later test three of the typical benchmarking problems: *Flow over an isolated mountain*, *Rossby-Haurwitz wave test* and *Propagation of a highly nonlinear wave*. Our solutions are compared against solutions produced by an *RBF-FD* framework given in [22]. Observations on accuracy are based on [5].

6.1. Functions

Recalling Definition 2.1 and Remark 2.2, both concerning the radial (basis) functions, we decide to use Gaussian functions as the radial basis functions for our numerical tests:

$$\phi(||\underline{x} - \underline{x}_k||) = e^{-\varepsilon^2 r^2}, \tag{6.1}$$

where ε is a shape parameter and $r = r(\underline{x}, \underline{x}_k)$, $\{\underline{x}, \underline{x}_k\} \in \mathbb{R}^n$ is an Euclidean distance $r = \sqrt{(x_1 - x_{k1})^2 + (x_2 - x_{k2})^2 + ... + (x_n - x_{kn})^2}$.

Next, recalling (2.19) and (2.20) we choose compactly supported Wendland functions $\Psi \in C^2$ as the partition of unity weight functions:

$$\Psi(r) = 4(\rho r + 1)(1 - \rho r)_+^4, \tag{6.2}$$

where r is an Euclidean distance and ρ^{-1} is the radius of a patch Ω_j.

6.2. Nodes

We choose two kinds of quasi-uniform points on the sphere.

- *Maximum determinant* [20][26] points as a set of global nodes of size N,
- *Minimum energy* [3][26] points as a set of patch (spherical cap) center points of size M.

6.3. Differentiation on the sphere

Given a C^∞ test function:

$$f = f(x, y, z) = \cos(3x^2 + 5y^2 + 8(z - 1)^2), \tag{6.3}$$

with surface gradient [25],

$$\nabla_x f = 2x(3x^2 + 5y^2 + 8z(z - 1) - 3\sin(3x^2 + 5y^2 + 8(z - 1)^2)), \tag{6.4}$$
$$\nabla_y f = 2y(3x^2 + 5y^2 + 8z(z - 1) - 5\sin(3x^2 + 5y^2 + 8(z - 1)^2)), \tag{6.5}$$
$$\nabla_z f = 2z(3x^2 + 5y^2 + 8(z - 1)^2(z + 1)\sin(3x^2 + 5y^2 + 8(z - 1)^2)). \tag{6.6}$$

we compare the solution obtained using *RBF-PUM* surface differentiation matrices, to the analytical solution given above in order to find out:

- what kind of effect does the overlap factor q (number of patches to which one node belongs on average) have on the accuracy of approximation of the gradient,
- which local shape parameter $\frac{\varepsilon}{\sqrt{N}}$ to choose in order to get the best accuracy.

Such tests make sense to perform since the surface gradient is a basic element of the shallow-water equations. Tests can also offer us a good starting point for configuring the *RBF-PUM* framework.

Remark 6.1 *Discrete* RBF-PUM *surface gradient is given in (4.25) and (4.26).*

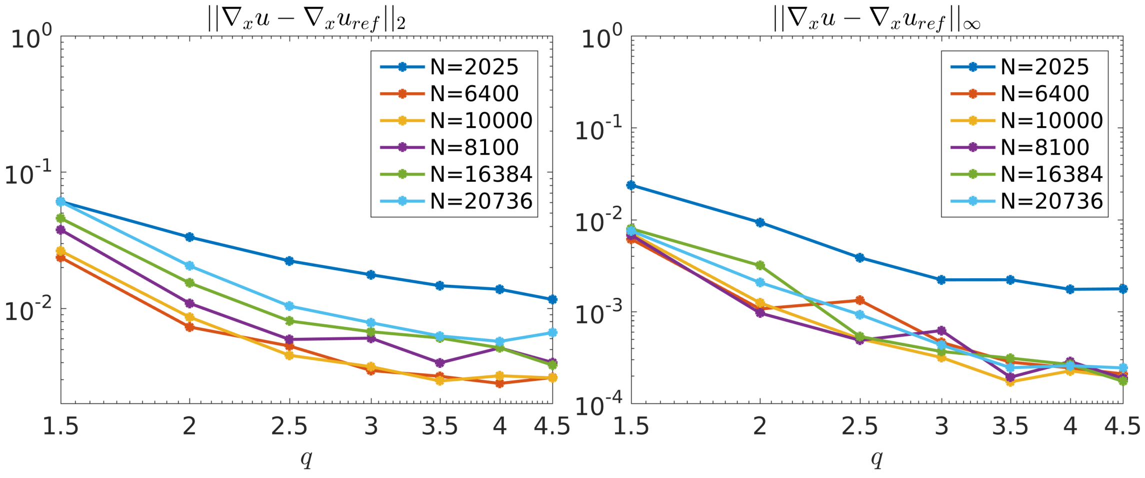

Figure 6.1.: Surface gradient as a function of overlap factor q and number of global nodes N ($||e||_2$ on the left, $||e||_\infty$ on the right).

It can be seen from Figure 6.1 that the slope of accuracy tends to achieve an equilibrium state at $q \approx 3$ i.e. when every neighboring patch of a patch Ω_j intersects Ω_j in its geometrical center. Based on that, we decided to choose $q > 3$, that is $q = 4$ to test the accuracy of the surface gradient in Figure 6.2. Such q also fulfills the stability criteria given in Section 4.3.2, which in this case means we can be sure our solution to a hyperbolic problem can be stabilized using a hyperviscosity term and also that it will at the same time have as high accuracy as possible.

Among $\frac{\varepsilon}{\sqrt{N}} = \{0.034, 0.047, 0.064\}$ we found out that the best accuracy was obtained when $\frac{\varepsilon}{\sqrt{N}} = 0.047$ (Figure 6.2). Most importantly, we have validated that approximation of the surface gradient has a convergent solution for all of tested $\frac{\varepsilon}{\sqrt{N}}$ with comparable accuracies. Due to an interpolation matrix A (2.2) being naturally ill-conditioned, this might not be the case for certain $\frac{\varepsilon}{\sqrt{N}}$. Also, we can now be sure that the implementation of *RBF-PUM* discretization is fine. Knowing that, we move on with performing tests on shallow-water equations.

6.4. Initial conditions

Although we use Cartesian coordinate system throughout this work, the initial conditions [24] are sometimes obtained in spherical coordinates and then converted to Cartesian coordinates.

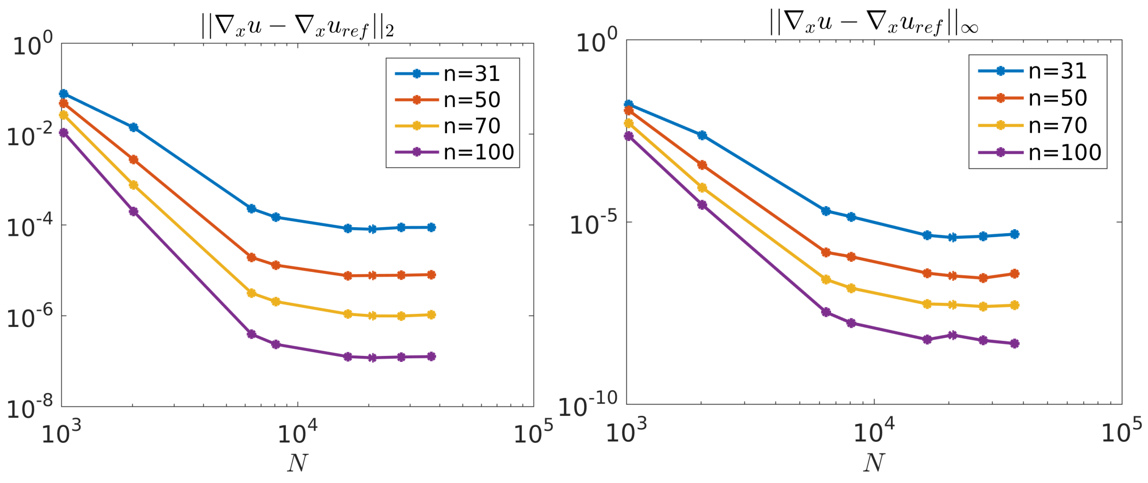

Figure 6.2.: Convergence plots ($||e||_2$ on the left, $||e||_\infty$ on the right) for obtaining the surface gradient as a function of global nodes N. The ratio $\frac{\varepsilon}{\sqrt{N}}$ is fixed as 0.047, the overlap factor is fixed as $q = 4$ (based on previous observations).

6.4.1. Flow over an isolated mountain (Test case 5)

Test case 5 (*TC5*) [24] shows how prone a numerical scheme is to ringing of gravity waves (also called *Gibbs phenomena*) throughout a 15 day simulation [5]. This is caused by two factors: the mountain forcing term which is undifferentiable and also by using a high-order spatial discretization.

Let $a = 6.37122 \cdot 10^6$ m be the radius of the Earth, $\alpha = 0$ the angle of rotation measured from the equator, $u^{(0)} = 20\,\frac{\text{m}}{\text{s}}$ the speed of the rotation, $\omega = 7.292 \cdot 10^{-}5\,\frac{1}{\text{s}}$ the rotation rate of the Earth, $g = 9.80616\,\frac{\text{m}}{\text{s}^2}$ the gravitational constant on the Earth and $G_0 = g \cdot 5960$ m the initial geopotential height. The geopotential height $h = h(\underline{x})$ and the wind field $\underline{u} = (u_1(\underline{x}), u_2(\underline{x}), u_3(\underline{x}))$ are then given as:

$$\begin{aligned}
h(\underline{x}, 0) &= -\Big(a\omega u^{(0)} - \frac{(u^{(0)})^2}{2}\Big)(-x\sin(\alpha) + z\cos(\alpha))^2, \\
u_1(\underline{x}, 0) &= u^{(0)}(-y\cos(\alpha)), \\
u_2(\underline{x}, 0) &= u^{(0)}(x\cos(\alpha) + z\sin(\alpha)), \\
u_3(\underline{x}, 0) &= u^{(0)}(-y\sin(\alpha)). \qquad (6.7)
\end{aligned}$$

The C^0 forcing term – Cone mountain

Let $\theta \in [0, 2\pi]$ represent longitude and $\phi \in [0, \pi]$ latitude on the sphere. Then $G = G(\theta, \phi)$ is the profile of a mountain given by:

$$G = G_{0m}\Big(1 - \frac{r}{R}\Big)_+, \qquad (6.8)$$

where $G_{0m} = 2000$ m is the height of a mountain, $R = \frac{\pi}{9}$, and $r^2 = \min[R^2, (\phi - \phi_c)^2 + (\theta - \theta_c)^2]$, (θ_c, ϕ_c) being longitudinal and latitudinal coordinates of the center of a mountain.

Remark 6.2 *We use Cartesian coordinates for computing the initial condition. In order to compute the forcing term $G = G(\theta, \phi)$ on the computer, we can transform all Cartesian nodes $(\underline{x})$ to their spherical equivalent (θ, ϕ) using function* cart2sph *in Matlab and evaluate G using these*

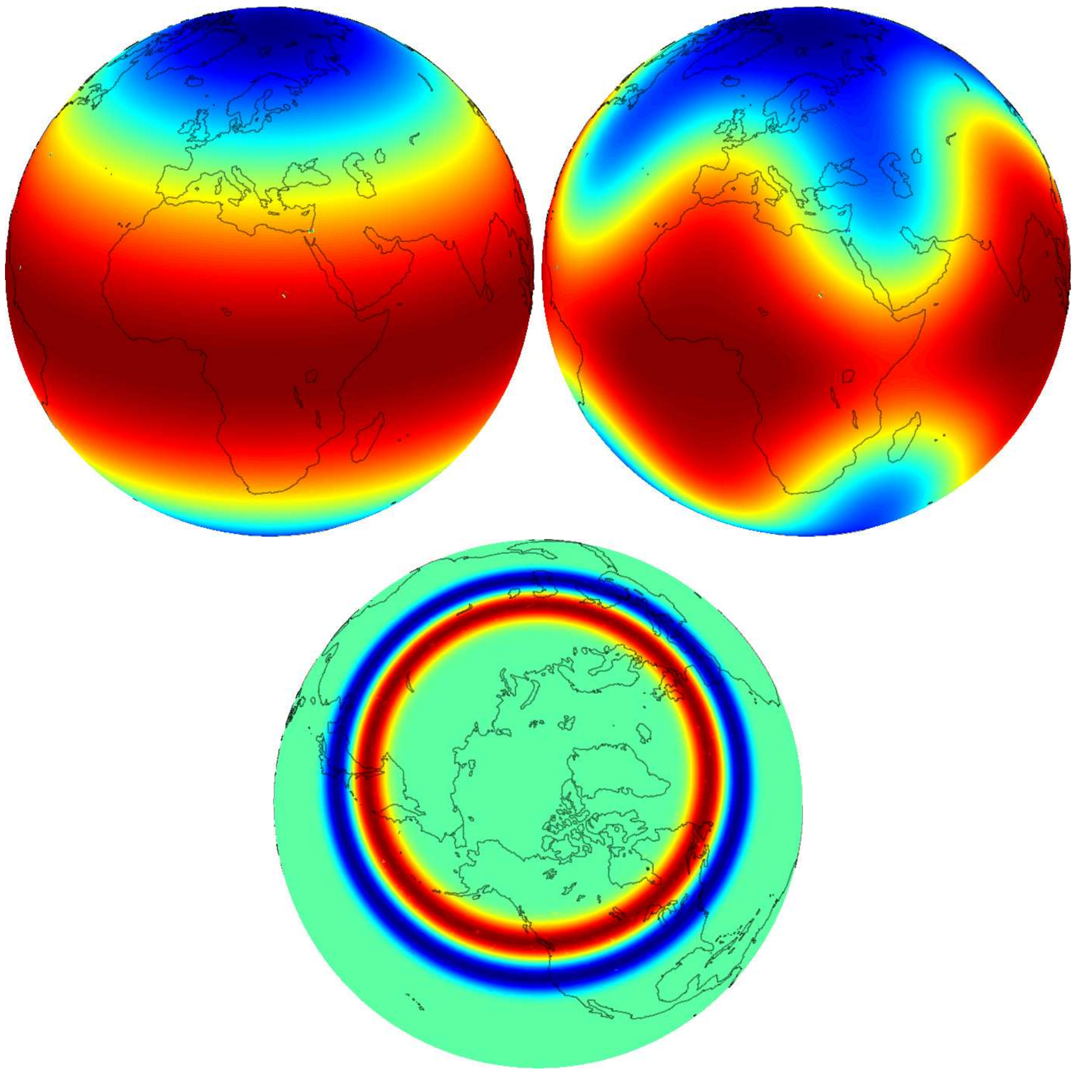

Figure 6.3.: Initial conditions. First row, left: Flow over an isolated mountain (initial geopotential height h). First row, right: Rossby-Haurwitz wave test (initial geopotential height). Second row: Evolution of a highly nonlinear wave (initial vorticity). The worldmap was drawn using the *spherepts* library [26].

coordinates. Since we are dealing with a scalar field, the values of G evaluated in (θ, ϕ) *are the same as the ones that would be evaluated in* $(\underline{x})$.

We include the profile of the mountain in our numerical scheme by treating G as a forcing term added to the right-hand-side of the equation for the geopotential height h in (4.28).
The term $D^{(j)}h$ transforms as:

$$D^{(j)}\underline{h} \to D^{(j)}(\underline{h} - \underline{G}), \tag{6.9}$$

and the term $\underline{h} \odot \left(K^{(j,x)}\underline{u}_1 + K^{(j,y)}\underline{u}_2 + K^{(j,z)}\underline{u}_3\right)$ of the same equation (4.28) becomes:

$$\begin{aligned} &\underline{h} \odot \left(K^{(j,x)}\underline{u}_1 + K^{(j,y)}\underline{u}_2 + K^{(j,z)}\underline{u}_3\right) \to \\ \to \quad &(\underline{h} - \underline{G} + G_0) \odot \left(K^{(j,x)}\underline{u}_1 + K^{(j,y)}\underline{u}_2 + K^{(j,z)}\underline{u}_3\right). \end{aligned} \tag{6.10}$$

6.4.2. Rossby-Haurwitz wave test (Test case 6)

In contrast with Test case 5, the Rossby-Haurwitz wave test *(TC6)* [24] is fully described in spherical coordinates. Using a wave of wavenumber four, the initial condition is also the solution after one revolution of the Earth, but only for the nondivergent barotropic vorticity equation. In this case, the wave pattern is propagating from west to east without change in shape. Shallow-water equations can only approximate this behaviour, constantly generating instabilities in the flow.

Let again $a = 6.37122 \cdot 10^6$ m be the radius of Earth, $\alpha = 0$ angle the of rotation measured from the equator, $u^{(0)} = 20\,\frac{\mathrm{m}}{\mathrm{s}}$ the speed of the rotation, $\omega = 7.292 \cdot 10^{-}5\,\frac{1}{\mathrm{s}}$ the rotation rate of the Earth, $g = 9.80616\,\frac{\mathrm{m}}{\mathrm{s}^2}$ the gravitational constant on the Earth, $K = 7.848 \cdot 10^{-6}\,\frac{1}{\mathrm{s}}$, $R = 4$ the wave number, $G_0 = g \cdot 5960$ m the initial geopotential height. This time the wind field is given in latitudinal and longitudinal components: $\underline{u} = (u_1(\theta,\phi), u_2(\theta,\phi))$. One can then write:

$$\begin{aligned} u_1(\theta,\phi) &= a\omega\cos(\theta) + aK\cos^{R-1}(\theta)(R\sin^2(\theta) - \cos^2(\theta))\cos(R\phi), \\ u_2(\theta,\phi) &= -aKR\cos^{R-1}(\theta)\sin(\theta)\sin(R\phi). \end{aligned} \tag{6.11}$$

Geopotential height is also a function of longitude and latitude $h = h(\theta,\phi)$ and is given as:

$$\begin{aligned} h(\theta,\phi) &= a^2A + a^2B\cos R\phi + a^2C\cos 2R\phi \qquad (6.12) \\ A &= \frac{1}{2}K(2\omega + K)\cos^2\theta + \frac{1}{4}K^2\cos^{2R}\theta\big[(R+1)\cos^2(\theta) + ... \\ ... &\ +R - 2 - 2R^2(-1 + \cos^{-2}\theta)\big] \\ B &= \frac{1}{(R+1)(R+2)}2K(\omega + K)\cos^R\theta\big[R^2 + 2R + 2 - (R+1)^2\cos^2\theta\big] \\ C &= \frac{1}{4}K^2\cos^{2R}\theta\big[(R+1)\cos^2\theta - R + 2\big]. \end{aligned}$$

Remark 6.3 *We omit the description of the analytic approach for transforming a vector field from spherical to Cartesian coordinates. On the computer, the Cartesian wind field* $\underline{u} = (u_1(\underline{x}), u_2(\underline{x}), u_3(\underline{x}))$ *can be transformed from the spherical wind field* $\underline{u} = (u_1(\theta,\phi), u_2(\theta,\phi))$ *using* sph2cartvec *function of Matlab. Geopotential height* h *can be computed in the same way as the forcing term of Test case 5 in Remark 6.2 is.*

6.4.3. Evolution of a highly nonlinear wave

The literature [12] describes this test as the most challenging one since it is producing a complex vortical dynamics. As described in [5], the evolution of a highly nonlinear wave is made with rapid energy transfer over a short period of time. That requires good handling of sharp gradients.

Let the background flow $u = u(\theta)$ be only a function of latitude θ. We write:

$$u(\theta) = \begin{cases} 0 & \text{for } \theta \le \theta_0 \\ \frac{u_{max}}{e_n}\exp\left[\frac{1}{(\theta-\theta_0)(\theta-\theta_1)}\right] & \text{for } \theta_0 < \theta < \theta_1 \\ 0 & \text{for } \theta \ge \theta_1 \end{cases} \tag{6.13}$$

where $u_{\max} = 80\frac{\mathrm{m}}{\mathrm{s}}$, $\theta_0 = \frac{\pi}{7}$, $\theta_1 = \frac{\pi}{2} - \theta_0$, $e_n = \exp(-4(\theta_1 - \theta_0)^2)$.

Let $h_0 = 10158.295\text{m}$ be a reference height and let all of the other represented constants have the same values as in *Flow over an isolated mountain* test case. For $-\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}$ the inital geopotential height $h = h(\theta)$ is then given as:

$$h(\theta) = h_0 + \frac{1}{g} \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} u(\theta')[2a\Omega\cos(\theta') + \tan(\theta')u(\theta')d\theta']. \tag{6.14}$$

The instability is generated by perturbing the height field $h(\theta)$, using Gaussians multiplied with a cosine which forces the perturbation to go to zero at the poles [5]. We write the perturbation $h'(\lambda, \theta)$ as:

$$h'(\lambda,\theta) = 120\cos(\theta)\exp\Big[-\Big(\frac{\lambda}{3}\Big)^2\Big]\exp\Big[-\Big(\frac{\pi/4-\theta}{15}\Big)^2\Big], \qquad \text{for} -\pi < \lambda < \pi. \tag{6.15}$$

Remark 6.3 concerning conversion of spherical vector field to the Cartesian vector field also applies here.

6.5. Effect of the chosen hyperviscosity approach

Definition 4.5 gives us the hyperviscosity approach of which effects we analyze on the linearized shallow-water equations presented in (4.30). Eigenvalues of the single linearized differential operator are observed with respect to *4th order Runge-Kutta* stability region.

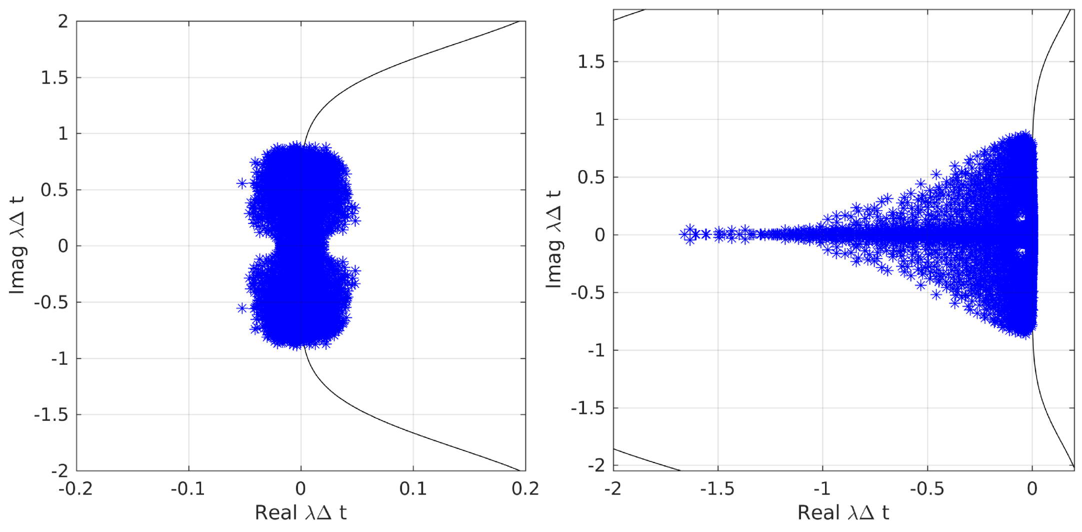

Figure 6.4.: Both cases consider the initial condition of Rossby-Haurwitz wave test, using $\Delta t = 300$, $N = 2025$, $n = 100$, $q = 4$, $\varepsilon = 3.061$. Left: Eigenvalues bounded by RK4 stability region when no hyperviscosity is added. Right: Eigenvalues bounded by RK4 stability region when hyperviscosity scaled with $\mu = 10^{-13}$ is added. Note that Δt in the right case could be slightly larger and RK4 stability region would still not be violated.

From Figure 6.4 we see that the unstable eigenvalues are propagated from the right complex half-plane to the left one. As a result, the eigenvalues for the given input data are now in the stable region. We conclude that the chosen hyperviscosity approach has a proper effect, which gives a promise of positive outcome for different input data—essentially the number of global nodes N and number of patch nodes n. From practical experience from the experiments we found that when higher global node numbers N are considered ($N > 36864$), it is in some cases hard to find the parameter μ in combination with a timestep Δt that would make the method stable. However, we were still able to stabilize the method for many nodesets with $N \leq 36864$. For that reason and due to its simplicity we decided to use it in this thesis.

Remark 6.4 *Additional experiments showed that the given hyperviscosity approach is more likely to have a proper effect with respect to stability when applied to* RBF-PUM *spatial discretization of the transport equation.*

6.6. Final results: RBF-PUM compared to RBF-FD

Comparing the solution obtained using *RBF-PUM* to the solution obtained using *RBF-FD* in [5], we target differences in visualized solutions, convergence orders and accuracies. *RBF-FD* was already compared to schemes which are considered as robust and are used at renowned institutions performing research on climate [5].

These are:

- (DG) Discontinuous Galerkin spectral element model (National Center for Atmospheric Research in Colorado, USA),
- (DWD-SH) Spectral transform shallow water model based on spherical harmonics (Deutscher Wetterdienst—German National Weather Service).

By knowing how *RBF-PUM* compares to *RBF-FD* and knowing how *RBF-FD* compares to *DG* and *DWD-SH* we can get an information of how *RBF-PUM* compares to *DG* and *DWD-SH*. The results of these two methods are considered as the ground truth. In cases *Flow over an isolated mountain* and *Rossby-Haurwitz wave test* we analyze the error of geopotential height $h = h(\underline{x}, t)$. In *Propagation of highly nonlinear wave test* case, the error is visually observed based on vorticity $V = V(\underline{x}, t)$. In all cases, we observe the normalized error given as:

$$\begin{aligned} ||e||_2 &= \frac{||u_{\mathrm{PUM}} - u_{\mathrm{FD}}||_2}{||u_{\mathrm{FD}}||_2} \\ ||e||_\infty &= \frac{||u_{\mathrm{PUM}} - u_{\mathrm{FD}}||_\infty}{||u_{\mathrm{FD}}||_2}, \end{aligned}$$

u being either h or V.

Remark 6.5 *The solutions using* RBF-FD *were generated using a solver given in [22].*

6.6.1. Flow over an isolated mountain (Test case 5)

Figure 6.5 shows the visual comparison between *RBF-FD* and *RBF-PUM* for $N = 27556$, $n = 100$. We do not observe any differences with respect to that.

Next, having a look at Figure 6.6 we observe only first order of convergence. The reason for that is the presence of the mountain forcing term (6.8) which is undifferentiable, but we still have to differentiate it when it is applied to the scheme. This causes Gibbs phenomena (ringing gravity waves throughout space and time) which limits the scheme in convergence and accuracy. The same order of convergence is also obtained when *RBF-FD* is compared to *DG* and *DWD-SH* [5]. The best accuracy of *RBF-PUM* is $||e||_2 \approx 9 \cdot 10^{-4}$.

N	M	ε	μ	Δt
6400	256	5.57	$1 \cdot 10^{-13}$	300
10000	400	6.93	$1 \cdot 10^{-13}$	300
16384	656	9.11	$2 \cdot 10^{-13}$	100
20736	830	10.06	$3 \cdot 10^{-13}$	50
27556	1103	11.46	$3 \cdot 10^{-13}$	50

N	ε	γN^{-k}	Δt
6400	5.1	$-0.2 \cdot N^{-8}$	300
10000	6.4	$-0.2 \cdot N^{-8}$	300
16384	8.2	$-0.2 \cdot N^{-8}$	300
20736	9.2	$-0.2 \cdot N^{-8}$	300
27556	10.6	$-0.2 \cdot N^{-8}$	300

Table 6.1.: Flow over an isolated mountain, the tables (left: *RBF-PUM*, right: *RBF-FD*) list parameter configurations which were used for the study. Parameters $n = 100$ (number of nodes in a patch (*RBF-PUM*), stencil size (*RBF-FD*)) and $t_{\text{final}} = 15$days (the time at which we observe the error) were fixed for both methods and are not given in the tables. N stands for the number of global nodes, M for the number of patches, ε for the shape parameter, μ for the amount of hyperviscosity applied in the A^{-1} approach, Δ for timestep and γN^{-k} for the factor γ applied to a Laplacian of Δ^k, where k is its order. We use $k = 8$ for our tests.

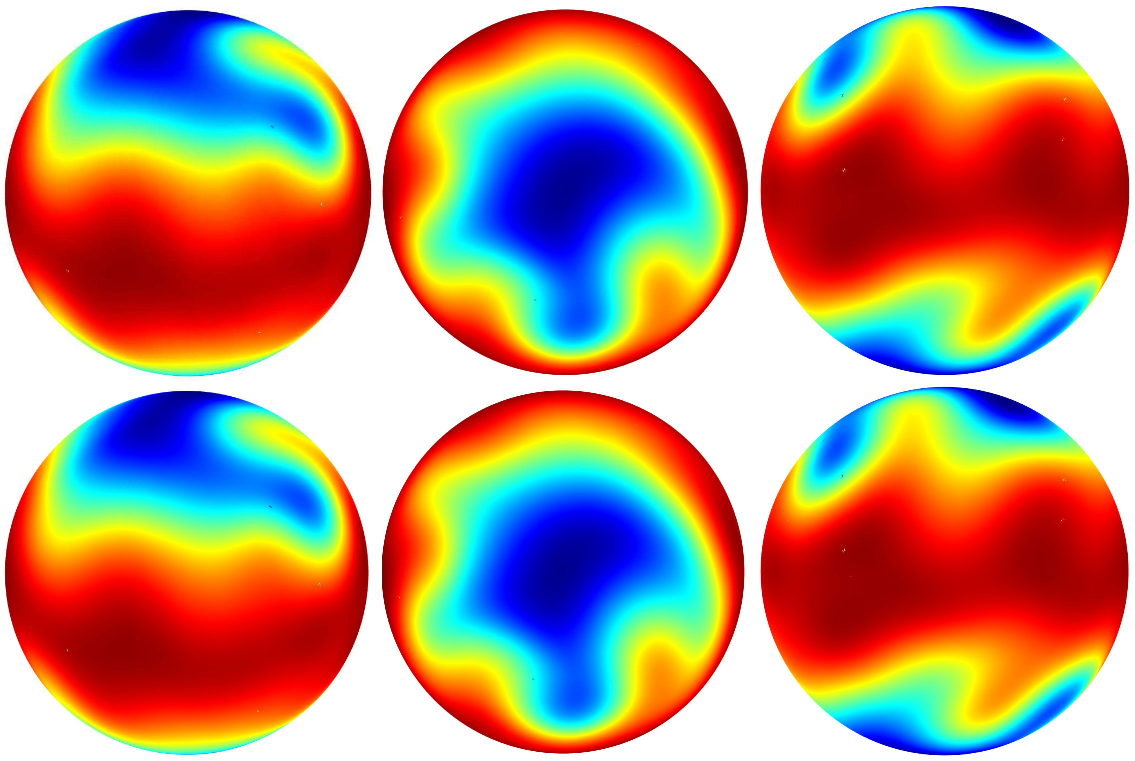

Figure 6.5.: Flow over an isolated mountain test case (*TC5*) after $t = 15$ days. RBF-PUM solution is shown in row 1, RBF-FD in row 2. Columns show the geopotential height $h(\underline{x}, 15)$ recorded from different angles. Note that the color scale is kept constant for all images.

Accuracy of *RBF-FD* compared to *DG* using the same $N = 27556$ was $||e||_2 \approx 4 \cdot 10^{-5}$, and $||e||_2 \approx 2 \cdot 10^{-4}$ when compared to *DWD-SH, T426* [5]. It is shown in *RBF-FD* self-convergence figures [5] that using a larger stencil size does not deliver a better accuracy due to Gibbs phenomena, which is present because of the high order of the method (approximately 9th order for $n = 101$). We can therefore assume our comparison between

RBF-FD using $n = 31$ and *RBF-PUM* using $n = 100$ to be valid. In any case, we do not expect the accuracy of *RBF-PUM* to be higher if n would be lowered to $n = 31$, which only confirms the validity.

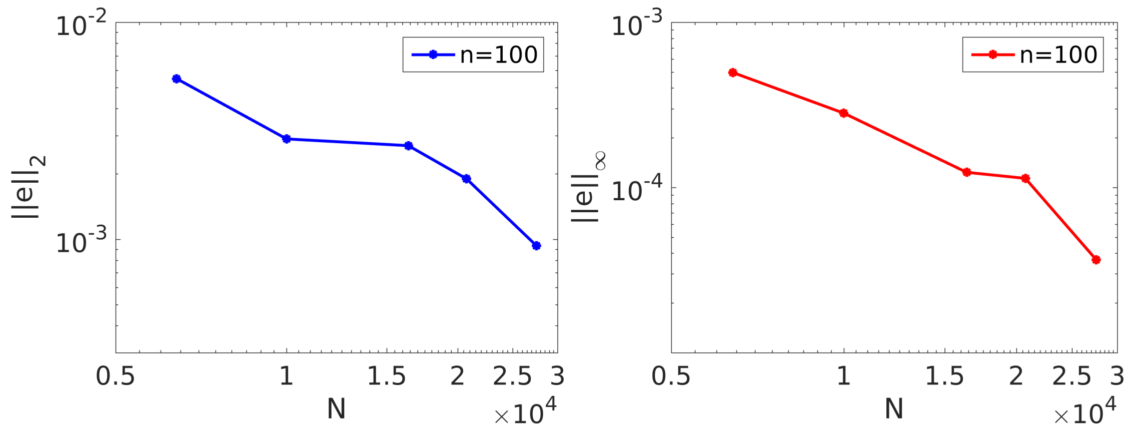

Figure 6.6.: Flow over an isolated mountain test: error of *RBF-PUM* after $t = 15$ days, with *RBF-FD* as a reference.

6.6.2. Rossby-Haurwitz wave test (Test case 6)

The measurements are performed based on parameters given in Table 6.2.

N	M	ε	μ	Δt
2025	81	3.14	$1 \cdot 10^{-13}$	300
6400	256	5.57	$1 \cdot 10^{-13}$	300
10000	400	6.93	$1 \cdot 10^{-13}$	300
16384	656	9.11	$4 \cdot 10^{-13}$	50
20736	830	10.06	$3 \cdot 10^{-13}$	50
27556	1103	11.46	$3 \cdot 10^{-13}$	50

N	ε	γN^{-k}	Δt
2025	2.9	$-0.2 \cdot N^{-8}$	300
6400	5.1	$-0.2 \cdot N^{-8}$	300
10000	6.4	$-0.2 \cdot N^{-8}$	300
16384	8.2	$-0.2 \cdot N^{-8}$	300
20736	9.2	$-0.2 \cdot N^{-8}$	300
27556	10.6	$-0.2 \cdot N^{-8}$	300

Table 6.2.: Rossby-Haurwitz wave test, the tables (left: *RBF-PUM*, right: *RBF-FD*) list parameter configurations which were used for the study. Parameters $n = 100$ (number of nodes in a patch (*RBF-PUM*), stencil size (*RBF-FD*)) and $t_{\text{final}} = 15$days (the time in which we observe the error) were fixed for both methods and are not given in the tables. N stands for number of global nodes, M for number of patches, ε for the shape parameter, μ for the amount of hyperviscosity applied to A^{-1} approach, Δ for timestep and γN^{-k} for the factor γ applied to a Laplacian of Δ^k, where k is its order. We use $k = 8$ for our tests.

From Figure 6.7 we see that solutions obtained using both methods look the same. We do not notice any diffusion effect that would be caused by hyperviscosity.

Looking at Figure 6.8 we determine order of convergence to be one. That also corresponds to the order obtained when *RBF-FD* is compared to reference solutions in [5]. The best accuracy we obtain for *RBF-PUM* using *RBF-FD* as a reference is $||e||_2 \approx 4 \cdot 10^{-3}$ at $N = 27556$ and $n = 100$. On the other hand, *RBF-FD* at the same N and $n = 101$ has the accuracy of $||e||_2 \approx 2 \cdot 10^{-3}$ compared to *DWD-SH* and $||e||_2 \approx 3 \cdot 10^{-3}$. This means the error of *RBF-PUM* is larger, but still competitive with respect to *RBF-FD*.

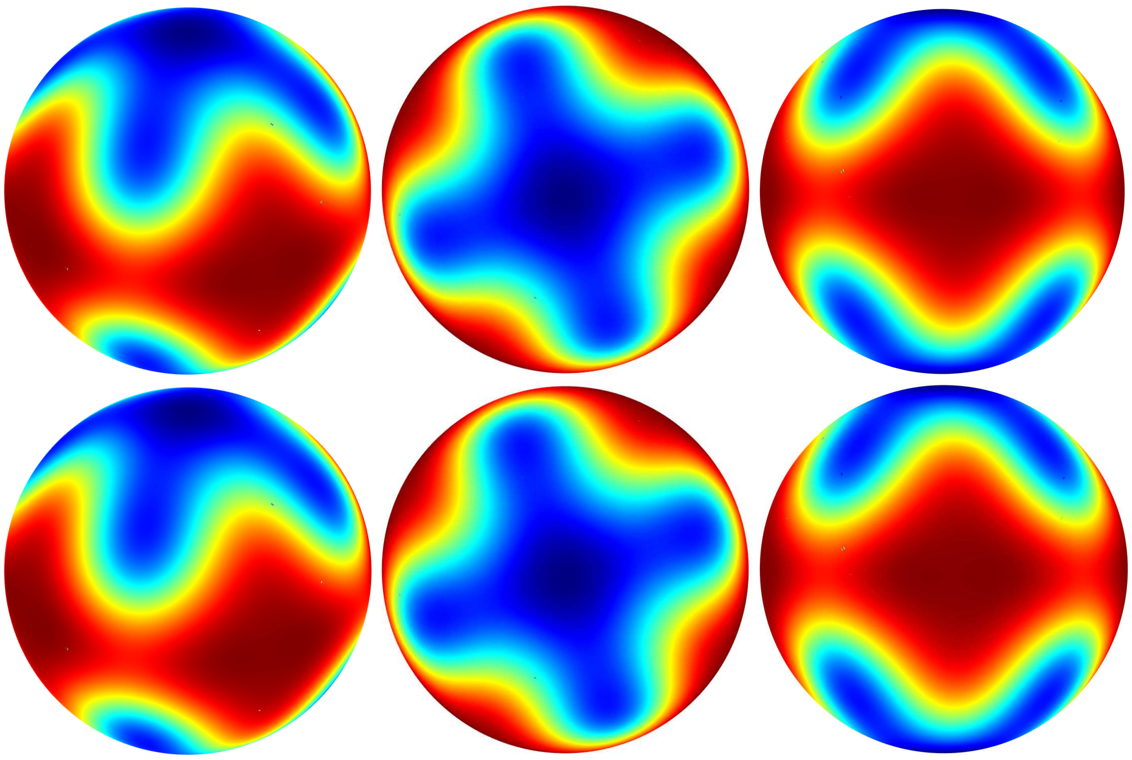

Figure 6.7.: Rossby-Haurwitz wave test case after $t = 15$ days. RBF-PUM solution is shown in row 1, RBF-FD in row 2. Columns show the geopotential height $h(\underline{x}, 15)$ recorded from different angles. Note that color scale is kept constant for all images.

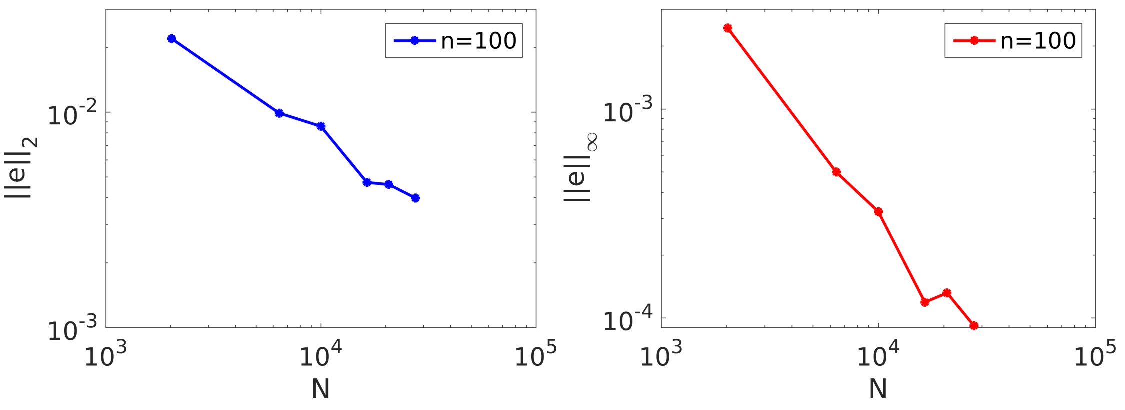

Figure 6.8.: Rossby-Haurwitz wave test: error of *RBF-PUM* after $t = 15$ days, with *RBF-FD* as a reference.

6.6.3. Evolution of a highly nonlinear wave

Parameters for obtaining results within this test are given in Table 6.3. The goal is to compare vorticities $V(\underline{x}, t)$ of both solutions after 6 days of simulation. Let $\underline{u} = (u_1(\underline{x}, t), u_2(\underline{x}, t), u_3(\underline{x}, t))$ be the wind field. Vorticity $V = V(\underline{x}, t)$ is then given as a component-wise sum of the curl of the wind field.

$$V = (\nabla \times \underline{u})\vec{i} + (\nabla \times \underline{u})\vec{j} + (\nabla \times \underline{u})\vec{k}. \tag{6.16}$$

N	M	ε	μ	Δt
2025	81	3.14	$2 \cdot 10^{-13}$	100
6400	256	5.57	$2 \cdot 10^{-13}$	100
10000	400	6.93	$2 \cdot 10^{-13}$	100
16384	656	9.11	$5 \cdot 10^{-13}$	60
27556	1103	11.46	$2 \cdot 10^{-13}$	30
36864	1475	13.52	$6 \cdot 10^{-13}$	40

N	ε	γN^{-k}	Δt
2025	2.9	$-0.3 \cdot N^{-8}$	200
6400	5.1	$-0.3 \cdot N^{-8}$	200
10000	6.4	$-0.3 \cdot N^{-8}$	200
16384	8.2	$-0.3 \cdot N^{-8}$	200
20736	9.2	$-0.3 \cdot N^{-8}$	200
36864	11.7	$-0.3 \cdot N^{-8}$	200

Table 6.3.: Highly nonlinear wave, the tables (left: *RBF-PUM*, right: *RBF-FD*) list parameter configurations which were used for the study. Parameters $n = 100$ (number of nodes in a patch (*RBF-PUM*), stencil size (*RBF-FD*)) and $t_{\text{final}} = 15$days (the time in which we observe the error) were fixed for both methods and are not given in the tables. N stands for number of global nodes, M for number of patches, ε for the shape parameter, μ for the amount of hyperviscosity applied to A^{-1} approach, Δ for timestep and γN^{-k} for the factor γ applied to a Laplacian of Δ^k, where k is its order. We use $k = 8$ for our tests.

From Figure 6.9 we can see that solutions after 6 days look very similiar. The edges of the wave, however, look more jagged in *RBF-PUM* case. It is also noticeable that the wave is more diffused in *RBF-PUM* case, which is expected since it was shown in [10] that classical Δ^k approach—which is used in *RBF-FD*—act as a better hyperviscosity term than A^{-1}—which is used in *RBF-PUM*—does. Our experiments showed that *RBF-PUM* and *RBF-FD* share the properties of keeping the proper wave pattern even in low resolutions ($N = 6400$). According to [5], this is not the case for *DG* and *DWD-SH*.

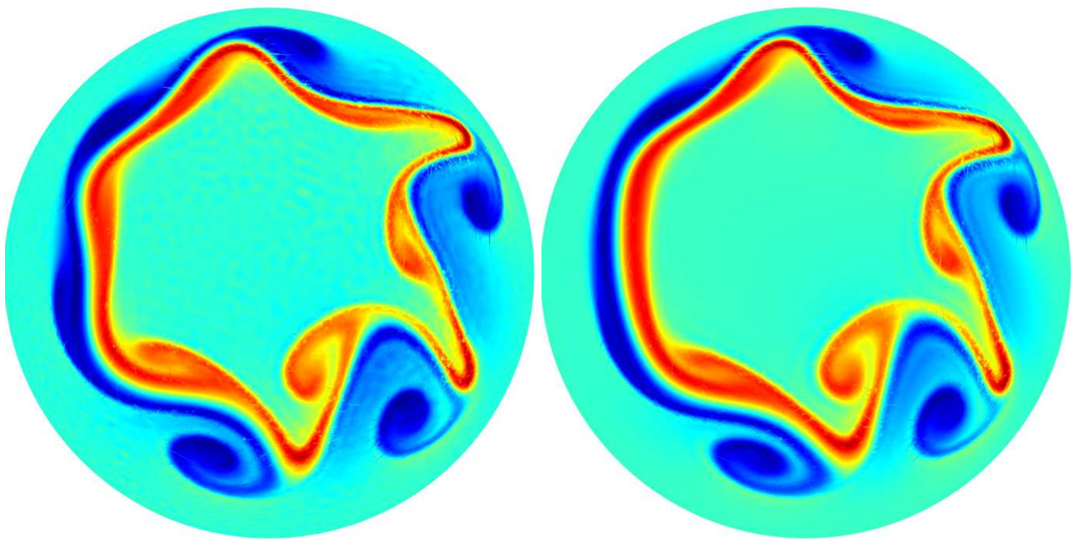

Figure 6.9.: Highly nonlinear wave test case after $t = 6$ Earth revolutions. Left: RBF-PUM. Right: RBF-FD. Figures show the vorticity after $t_{\text{end}} = 6$ days recorded from different angles. Note that color scale is kept constant for all images. The image is produced using $N = 36864$ nodes. The artefacts do not result from the solution, but are a consequence of triangularizing the mesh during visualization.

7. High-performance computing results

Due to reasonable error terms inspected in the section with numerical results, we would like to check how does *RBF-PUM* compare to *RBF-FD* with respect to sequential runtime and parallel scalability on shared-memory architecture. That is sensible since the ratio between runtime and accuracy could still be in favor of *RBF-PUM*. The tests are performed on a single *Uppmax-Rackham* node with two Intel Xeon E5-2630 v4—Broadwell architecture—CPUs, each offering ten computational cores.

7.1. Memory jumps

Optimization with respect to memory jumps is known to be an important requirement for producing an efficient (parallel) algorithm. Referring to Section 5.2 we remember that our patches are divided in patch groups and the global vectors into global blocks. That means that a single group of patches usually works on several blocks of global vectors. In case the selected blocks are distanced far from each other we are already facing unnecessary memory jumps. Furthermore, since there is always an overlap of patches present due to the nature of *RBF-PUM* (for our case we are obliged to form non-negligible overlap—Section 4.3.2), that means that neighboring patches in a patch group can share blocks from global vectors. A single patch group is therefore given an opportunity to access fewer global blocks and by that reduce the jumping.

7.1.1. Jumps reduction

We would firstly like to achieve that members of neighboring patches inside the global structures are ordered together tightly. This can be obtained by snake-ordering the patch center nodes such that a patch in the northern-most position gets an index 1 (Ω_1), its closest neighbor gets an index 2 (Ω_2), the closest neighbor of Ω_2 (without considering Ω_1) gets an index 3 and so on. An example is given in Figure 7.1, where it can also be seen that our snake ordering algorithm does not work perfectly for small amount of patch centers, but does perform well when the resolution is higher. That is reasonable since quasi-uniform nodes are likely to be more uniform when the number of nodes tends towards infinity. At the same time less and less heuristics is needed in order to not hit a wrong snake-neighbour.

On the second level, we would also like that members of a single patch are put together as tightly as possible. This can be done by following the patches in the snake order performed on the first level and reordering the global structure V accordingly i.e. V = [members(Ω_1), members(Ω_2), ..., members(Ω_M)]. By performing that and considering patch groups instead of a single patch, we can be sure the patch group contains an ordered set of global blocks to which a patch member has an access, but also that there will be much less unique blocks since several patch members inside a patch group have to access certain global blocks which are the same (overlaps). The result of patch centers being at first snake ordered (first level) and then reordered within the second level can be seen in Figure 7.2. We notice that when there is no ordering applied, one patch consid-

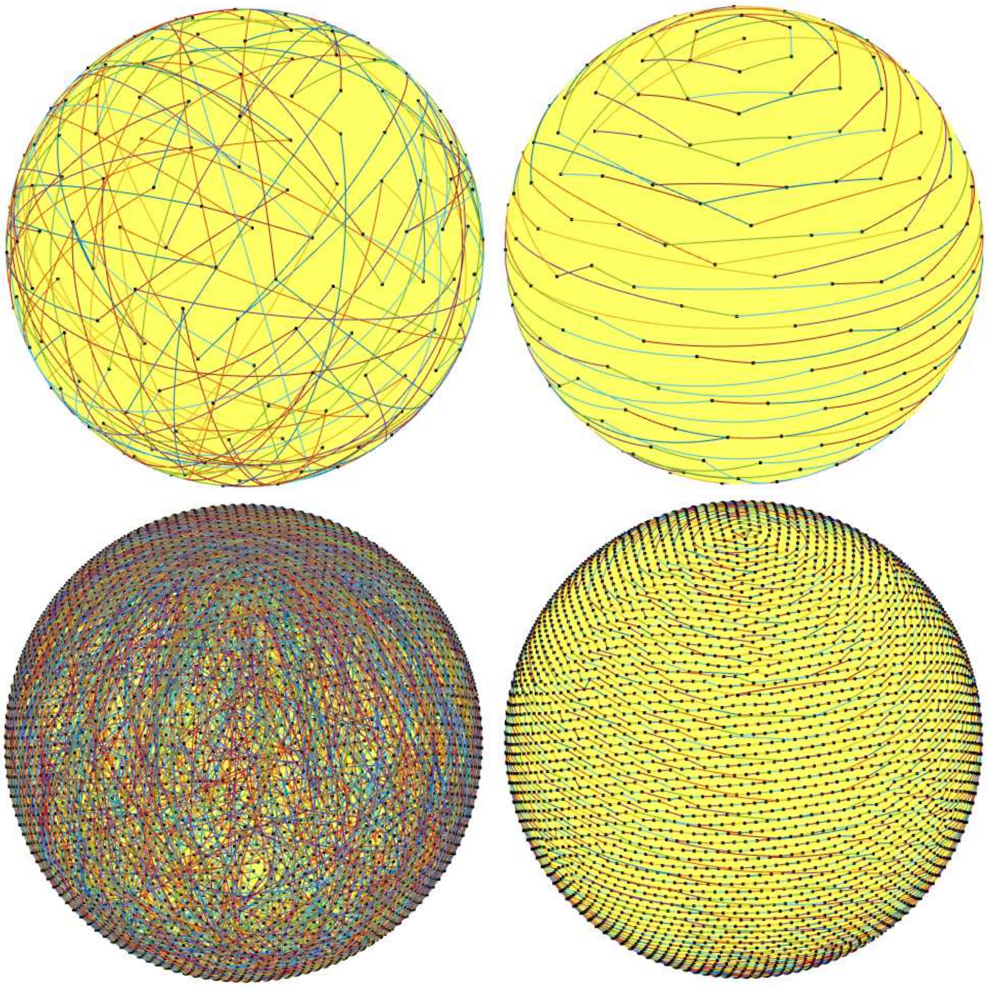

Figure 7.1.: Rows 1 and 2 consider cases of $\{N = 6400,\ M = 256\}$ and $\{N = 115718,\ M = 6241\}$ respectively. Left column shows the unordered set of patch centers, right column shows the snake-ordered set of patch centers. In this case, the sphere and the nodes on it were drawn using *spherepts* library [26].

ers many more global blocks than in the ordered case which causes much more memory jumps.

7.2. RBF-PUM benchmark

Considering our task-based parallel simulation of *Flow over an isolated mountain* case, using global nodes $N = 155718$, patch size $n = 100$, number of patches $M = 6229$ and overlap factor $q = 3.807$, we compare two ways (described in Section 5.4) of computing the right-hand-side of (4.27). The straightforward form (16 MVPs) is computed using *dgemv* routines and the dense form (4 MMPs) using *dgemm* routines. The *dgemm* routine is run in a setup using 150 patch groups and 422 global array blocks, and *dgemv* in a setup using 250 patch groups and 422 global array blocks. Furthermore, both, the *dgemv* and the *dgemm* routines are implemented using *BLAS* and *Intel MKL* libraries.

The results presented in Figure 7.3 show strong scaling and runtime curves. *BLAS* - *dgemm* scales the best among all approaches (having a speedup of approximately 11.75

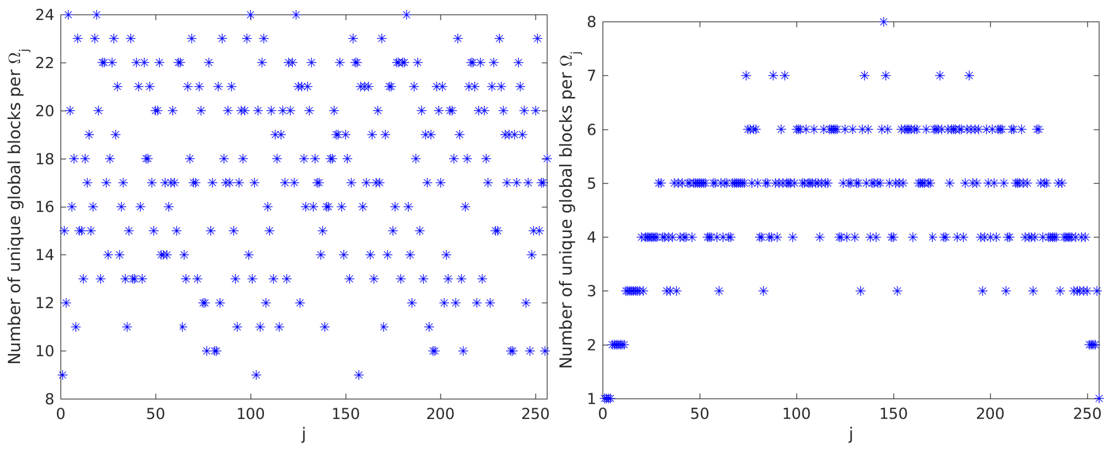

Figure 7.2.: Left: unordered global arrays. Right: ordered global arrays. Both cases consider $N = 6400$, $M = 256$, $n = 100$, $q = 4$. Every patch Ω_j has its own identifier j (x-axis). Y-axis shows how many global blocks of length 256 are accessed by the j-th patch.

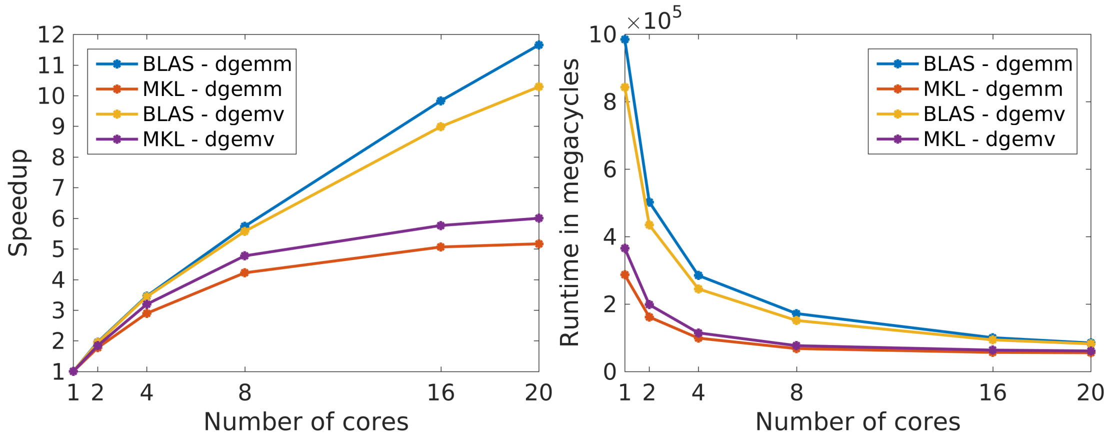

Figure 7.3.: *RBF-PUM* based parallelized simulation using *dgemv* and *dgemm* routines, invoked from *BLAS* and *Intel MKL* libraries. Left: strong scaling. Right: runtime.

using 20 cores), however, its sequential runtime is the worst. On the other hand, *MKL-dgemm* scales the worst (speedup of approximately 5.1 using 20 cores) but its sequential runtime is the best and so is the runtime when 20 cores are used. This behaviour can be reasoned by the fact that *Intel MKL* is sequentially more optimized than *BLAS*, since it for example uses *AVX* instructions. Also, *Intel MKL* is expected to behave better since it is developed for state of the art Intel architectures. Classical *BLAS* is a bullet-proof library, but not fully optimized with respect to nowadays computer architectures.

Based on that we decide to further analyze *MKL-dgemm* by presenting contention measurements and a trace of tasks.

Definition 7.1 *Let Γ_d be a time-domain bounded by the runtime of the whole algorithm and let $T^{(i)} = T^{(i)}(n) \in \Gamma_d$ be the runtime of a single task i as a function of number of threads n. Then* *contention $C = C(n)$ is given as,*

$$C(n) = \sum_{T^{(i)} \in \Gamma_d} \frac{T^{(i)}(n)}{T^{(i)}(1)}.$$

Using contention given in the definition above we study the effect of parallelism among three types of tasks: *scatterRHS*, *gatherRHS* and *blasMagic*. The reason to employ this measurement is simply to see if contention between several types of tasks is comparable: in this case the slowdown is a consequence of increasing demand for memory accesses with increasing number of threads, and not that a certain type of a task has negative characteristics when several threads try to execute it. Figure 7.4 confirms that the contention between the three most expensive tasks is comparable. However, its average size when using 20 threads is approximately 2.3. That also means the best possible speedup we could obtain is $\frac{n}{\max(C(n))} = \frac{20}{\max(C(20))} = \frac{20}{2.7} = 7.4$. We deduce that there could be more optimization done with respect to memory bandwidth, possibly by making a customized matrix-matrix product algorithm, or, finding a way to convert square-skinny MMP (see Section 5.4) to a square-square one.

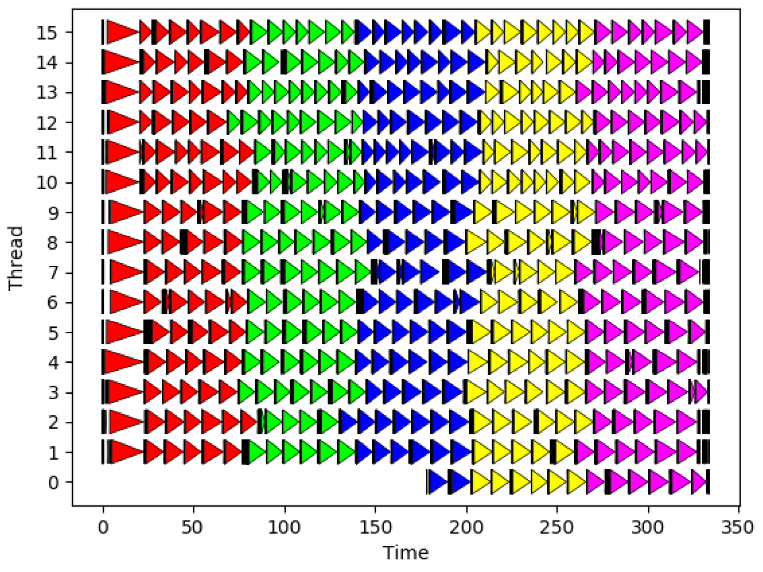

Figure 7.4.: A trace of tasks handled in 5 timesteps when running the parallelized simulation on 16 cores for $N = 16384$, $n = 100$, $M = 525$, $q = 4$, using 25 patch groups and global array blocksize of 64. The task submission is performed by thread 0 during the first ≈ 175 Megacycles. Note that one triangle respresents a single task. The tasks are colorized with respect to the timestep they belong to.

A trace of tasks gives us an overview of how tasks are executed with respect to the runtime. Figure 7.4 shows there are no critical delays between the executions of tasks – the majority of them are executed tightly one after another. The color of a node denotes a timestep in which a certain task is executed. We notice a mild overlap between consecutive timesteps. That is because we divided the global arrays into blocks, allowing the blocks that already obtained a final result in timestep k to continue obtaining the result for timestep $k + 1$. Our proposal with respect to that, described at the end of Section 5.2, is as such validated. We conclude that the provided task density is sufficient and that we can not drastically improve the speedup by improving the scheduling.

Finally, we proclaim the parallel algorithm based on *MKL-dgemm* as our best candidate to compete with the parallel implementation of *RBF-FD*.

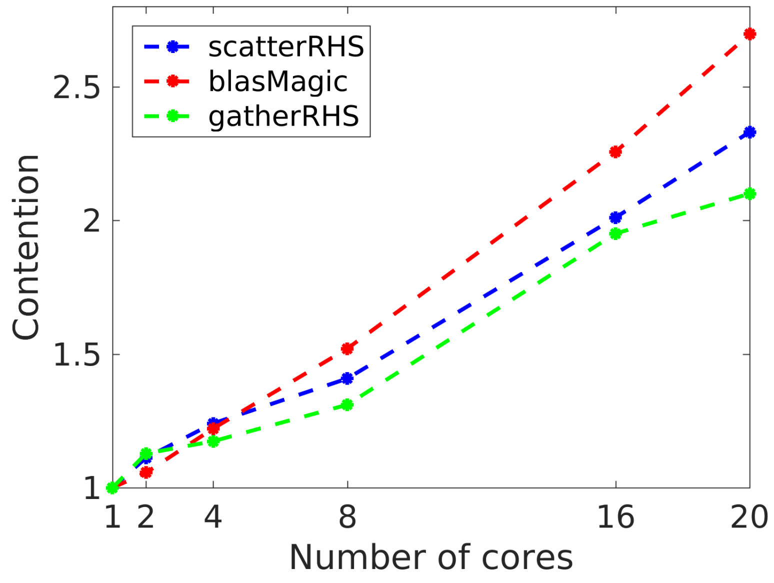

Figure 7.5.: Contention of three most important tasks measured for simulation parameters of $N = 155718$, $n = 100$, $q = 3.807$, $M = 6229$, $t_{\text{end}} = 15$ days, 150 patch groups and 422 global array blocks. Note that *MKL-dgemm* is used within *blasMagic* task.

7.3. Final results: RBF-PUM compared to RBF-FD

We compare the performance of our most promising *RBF-PUM* parallel implementation outlined in the subsection above and a parallel *RBF-FD* solver given in [22]. The key differences between main computational tasks in our approach and in approach from [22] are that the latter uses global sparse matrix-vector products, but we use the local dense matrix-vector products in combination with scattering and gathering data to global vectors.

Again, we consider a case of $N = 155718$ and $n = 100$ for both methods. The simulation is run for 100 timesteps. Within *RBF-PUM* we have $M = 6229$ patches with the overlap factor $q = 3.807$. Furthermore, the computational domain is split into 150 patch groups (≈ 42 patches per group) and 422 global array blocks (369 elements per global array block). The global sparse matrix within *RBF-FD* is split into blocks of size 2048×2048 elements.

It can be seen from Figure 7.6 that *RBF-PUM* scales slightly better than *RBF-FD*, no matter how many cores are used. Using 20 cores, our method has a speedup of approximately 5.2, while *RBF-FD* has a speedup of approximately 5.0. On the other hand, when comparing the runtimes (Figure 7.7), we see that *RBF-FD* solver is sequentially around 6-times faster than our solver. That can partially be reasoned by the fact that the patches are overlapping, i.e. if there are two patches overlapping, the local solution based on nodes which are inside the intersection will be computed twice (see Section 4.3.2). When using $q = 3.807$, as in our case, that means that on average every node is shared among 3.807 patches which means there is $q = 3.807$-times more work performed compared

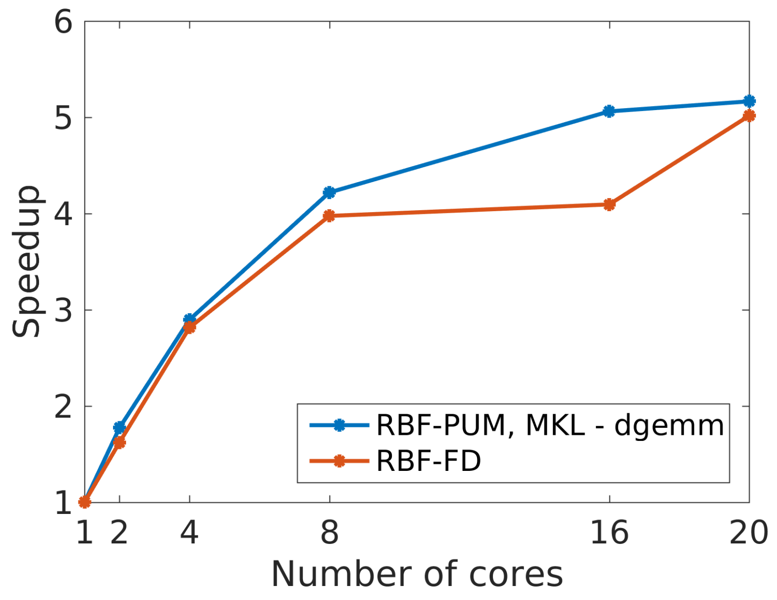

Figure 7.6.: Strong scaling, comparing *RBF-PUM* and *RBF-FD* based simulations ran for 100 timesteps, using $N = 155718$, $n = 100$.

to *RBF-FD*. By acknowledging that we see that our implementation is still $\frac{6}{3.807} = 1.58$-times slower, which is probably due to the given global sparse matrix-vector product being more optimized with respect to the memory bandwidth than our combination of scatter-local product-gather tasks. Furthermore, the contention reported in [22] is 1.52 for 16 threads, but our average contention for 16 threads (Figure 7.5) is slightly higher: 2.03, which just confirms that there is some space for improvements of our parallel implementation, especially since global sparse matrix-vector products are in general known to have many cache misses [22] and should as such be—from the very beginning—worse compared to dense products. However, on the other hand, it is emphasized in [22] that the actual matrix-vector product which is performed is much less sparse than it would be if 16 matrices describing the right-hand-side of the discrete shallow-water equations would not be combined into one sparse matrix.

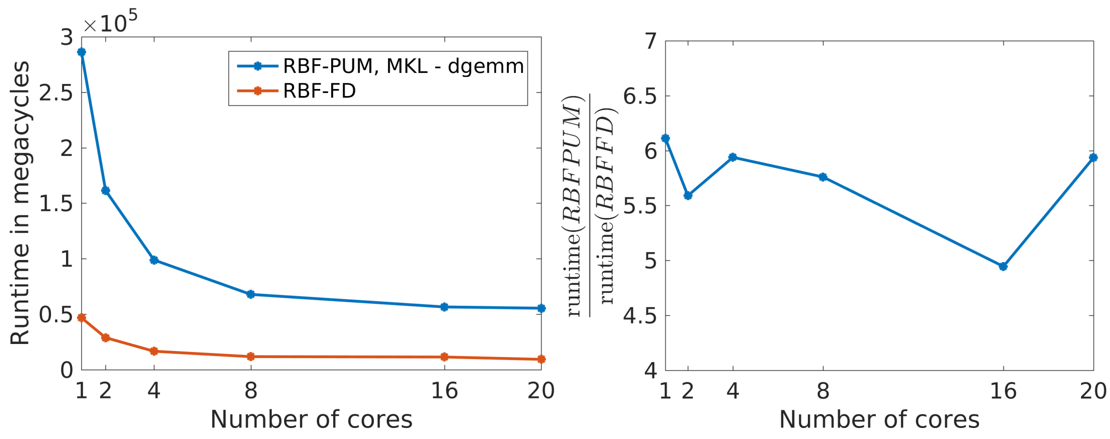

Figure 7.7.: Runtimes (left) and relative runtimes (right) of 100 timesteps simulation using *RBF-PUM* and *RBF-FD*, for $N = 155718$ and $n = 100$.

8. Conclusion

Based on Section 6.6 we deduce that the accuracy of *Radial basis functions partition of unity method* applied to the shallow-water equations is comparable to the accuracy of *Radial basis functions generated finite differences method*, when the initial conditions of *Rossby-Haurwitz wave test* and *Evolution of a highly nonlinear wave* are used. When *Flow over an isolated mountain* is considered, then we can not talk about methods being comparable anymore: *RBF-FD* is ≈ 22-times more accurate than *RBF-PUM* is when $N = 27556$ is used. This might well be because of the different hyperviscosity approaches used within the two methods. In order to draw the line for this inital case, more research on employing a better hyperviscosity approach within *RBF-PUM* has to be performed.

Section 7.3 shows that *RBF-PUM* can not compete with *RBF-FD* in terms of runtime, simply because of the observations outlined in Section 4.3.2. Although optimization on the memory bandwidth side could be performed in order to decrease the runtime of *RBF-PUM*, it is unlikely that this could provide a sufficient improvement considering the wind criteria which demands a large overlap of patches and as such introduces more work compared with *RBF-FD*.

We answer the questions stated in Introduction (Section 1).

- **Is *RBF-PUM* appropriate for approaching nonlinear hyperbolic problems in combination with explicit time-stepping schemes?**
  All of the indications presented in this thesis show that RBF-PUM *can be succesfully employed for approaching nonlinear hyperbolic problems, but is—as of the current research stage—not as accurate and fast as* RBF-FD *is.*

- **Are there any benefits delivered in comparison with similiar methods which were already applied to the shallow-water equations?**
  Considering initial condition Evolution of a highly nonlinear wave*, one of the benefits is that* RBF-PUM *is able to form a wave shape when used in the low-resolution setup. Comparable methods, such as* DG *and* DWD-SH *are not able to handle that.*

- **How does *RBF-PUM* in combination with explicit time-stepping scale when implemented as a parallel algorithm?**
  RBF-PUM *scales slightly better than* RBF-FD.

To conclude: regarding the numerical results, we now understand the impact of patch overlaps on the wind support of the scheme in a much better way and can therefore reason about the increased workload that is introduced in comparison to *RBF-FD*. These observations were—to the best of our knowledge—not given in any other publication. In addition, our parallel approach gives a good starting point for further investigations on parallelism within *RBF-PUM*.

9. Future work

One of the critical parts when it comes to the accuracy of the scheme based on *RBF-PUM* is the hyperviscosity approach A^{-1} described in Section 4.3.1. More research could be performed on the other approaches, for example approximating Δ^k using *RBF-PUM*. That would, however, require compactly supported partition-of-unity weight functions $w(\underline{x}) \in C^k$, which might be hard to obtain for higher k.

Although stabilization with respect to ill-conditioned interpolation matrices $A^{(j)}$ was not considered in this thesis, an idea leading to a stabilization using *RBF-QR* on the sphere (there is no such algorithm available yet) would be to employ a 2-dimensional *RBF-QR* [15] on the patches projected to planes tangential to their center. Despite this being a rough approximation since the curvature of the spherical caps is lost, it might still be acceptable for the cases when number of patches M tends towards infinity since this also means that the radius of a patch (spherical cap) tends towards zero, which then vastly reduces the curvature of a cap and by that reduces the error.

RBF-PUM runtime optimization could be made with a different parallel approach, namely using *SuperGlue* as a scheduler and one of the *GPGPU* frameworks for multiplication of matrix-vector products. That could greatly improve the speed of memory access, since it is known that *GPGPU* have better capabilities with respect to that. A drawback of this approach could on the other hand reflect in the large loading times that are needed in order to transfer the data between the memories.

Lastly, another possible optimization is an improvement of dense form of the matrix-matrix product introduced in Section 5.4. The current setup is such that a product DU consists of $D \in \mathbb{R}^{n\times n}$—where D is either one of the differentiation matrices or a hyperviscosity matrix—and $U \in \mathbb{R}^{n\times 4}$ which is a matrix consisting of temporal solutions u_1, u_2, u_3 and h. We could expand U to become $\hat{U} \in \mathbb{R}^{n\times n}$ by adding in additional $\frac{n-4}{4}$ temporal solutions as the matrix U columns. This would bring a big limitation on the size of patch groups, which would then have to be of size $k\frac{n}{4}$, $k = \{1, 2, ...\}$, but would on the other hand, as mentioned, form at least one product of two square matrices which would bring a possibility of better cache reusability and by that an improved runtime.

Bibliography

[1] Ivo Babuška and Jens Markus Melenk. The partition of unity method. *Internat. J. Numer. Methods Engrg.*, 40(4):727–758, 1997.

[2] Gregory E. Fasshauer. *Meshfree approximation methods with MATLAB*, volume 6 of *Interdisciplinary Mathematical Sciences*. World Scientific Publishing Co. Pte. Ltd., Hackensack, NJ, 2007. With 1 CD-ROM (Windows, Macintosh and UNIX).

[3] Jörg Fliege and Ulrike Maier. The distribution of points on the sphere and corresponding cubature formulae. *IMA J. Numer. Anal.*, 19(2):317–334, 1999.

[4] Natasha Flyer, Bengt Fornberg, Victor Bayona, and Gregory A. Barnett. On the role of polynomials in RBF-FD approximations: I. Interpolation and accuracy. *J. Comput. Phys.*, 321:21–38, 2016.

[5] Natasha Flyer, Erik Lehto, Sébastien Blaise, Grady B. Wright, and Amik St-Cyr. A guide to RBF-generated finite differences for nonlinear transport: shallow water simulations on a sphere. *J. Comput. Phys.*, 231(11):4078–4095, 2012.

[6] Natasha Flyer and Grady B. Wright. A radial basis function method for the shallow water equations on a sphere. *Proc. R. Soc. Lond. Ser. A Math. Phys. Eng. Sci.*, 465(2106):1949–1976, 2009.

[7] Bengt Fornberg. Calculation of weights in finite difference formulas. *SIAM Rev.*, 40(3):685–691 (electronic), 1998.

[8] Bengt Fornberg, Tobin A Driscoll, Grady Wright, and Richard Charles. Observations on the behavior of radial basis function approximations near boundaries. *Comput. Math. Appl.*, 43(3-5):473–490, 2002.

[9] Bengt Fornberg, Elisabeth Larsson, and Natasha Flyer. Stable computations with Gaussian radial basis functions. *SIAM J. Sci. Comput.*, 33(2):869–892, 2011.

[10] Bengt Fornberg and Erik Lehto. Stabilization of RBF-generated finite difference methods for convective PDEs. *J. Comput. Phys.*, 230(6):2270–2285, 2011.

[11] Bengt Fornberg and Grady Wright. Stable computation of multiquadric interpolants for all values of the shape parameter. *Comput. Math. Appl.*, 48(5-6):853–867, 2004.

[12] Joseph Galewsky, Richard K. Scott, and Lorenzo M. Polvani. An initial-value problem for testing numerical models of the global shallow-water equations. *Tellus*, 56A:429–440, 2004.

[13] Aiton Kevin. *A Radial Basis Function Partition of Unity Method for Transport on the Sphere*. Master Thesis, Boise State University, 2014.

[14] Jakub Kurzak and Jack Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. *Applied Parallel Computing. State of the Art in Scientific Computing: 8th International Workshop, PARA 2006, Umeå, Sweden, June 18-21, 2006, Revised Selected Papers*, pages 147–156, 2007.

[15] Elisabeth Larsson, Erik Lehto, Alfa Heryudono, and Bengt Fornberg. Stable computation of differentiation matrices and scattered node stencils based on Gaussian radial basis functions. *SIAM J. Sci. Comput.*, 35(4):A2096–A2119, 2013.

[16] Charles A. Micchelli. Interpolation of scattered data: distance matrices and conditionally positive definite functions. *Constr. Approx.*, 2(1):11–22, 1986.

[17] Christian Rieger and Barbara Zwicknagl. Sampling inequalities for infinitely smooth functions, with applications to interpolation and machine learning. *Adv. Comput. Math.*, 32(1):103–129, 2010.

[18] Ali Safdari-Vaighani, Alfa Heryudono, and Elisabeth Larsson. A radial basis function partition of unity collocation method for convection-diffusion equations arising in financial applications. *J. Sci. Comput.*, 64(2):341–367, 2015.

[19] Isaac J. Schoenberg. Metric spaces and completely monotone functions. *Ann. of Math. (2)*, 39(4):811–841, 1938.

[20] Ian H. Sloan and Robert S. Womersley. Extremal systems of points and numerical integration on the sphere. *Adv. Comput. Math.*, 21(1-2):107–125, 2004.

[21] Martin Tillenius. SuperGlue: a shared memory framework using data versioning for dependency-aware task-based parallelization. *SIAM J. Sci. Comput.*, 37(6):C617–C642, 2015.

[22] Martin Tillenius, Elisabeth Larsson, Erik Lehto, and Natasha Flyer. A scalable RBF-FD method for atmospheric flow. *J. Comput. Phys.*, 298:406–422, 2015.

[23] Bayona Victor, Moscoso Miguel, Carretero Manuel, and Kindelan Manuel. RBF-FD Formulas and Convergence Properties. *J. Comput. Phys.*, 229(22):8281–8295, November 2010.

[24] David L. Williamson, John B. Drake, James J. Hack, Rüdiger Jakob, and Paul N. Swarztrauber. A standard test set for numerical approximations to the shallow water equations in spherical geometry. *J. Comput. Phys.*, 102(1):211–224, 1992.

[25] Grady Wright. Exercise sheet: Problem 4, differentiation matrices. *Montestigliano Workshop*, 2014. `http://math.boisestate.edu/~wright/montestigliano/problem04.pdf`, retrieved on March 31, 2017.

[26] Grady Wright. Spherepts - A collection of Matlab functions for experimenting with different point sets on the sphere. *GitHub repository*, 2017. `https://github.com/gradywright/spherepts`, retrieved on March 31, 2017.

[27] Grady B. Wright and Bengt Fornberg. Stable computations with flat radial basis functions using vector-valued rational approximations. *J. Comput. Phys.*, 331:137–156, 2017.