ulm university universität **u**ulm

SISSA - Scuola Internazionale Superiore di Studi Avanzati
mathlab,
Mathematics Area

Fakultät für Mathematik und Wirtschafts-wissenschaften
Institut für
Numerische Mathematik

A multiprocess and multithreaded strategy for parallel proper orthogonal decomposition in a reduced order modeling framework

Master's thesis

By:
Florian Krötz
florian.kroetz@uni-ulm.de
884948

Reviewers :
Prof. Dr. Karsten Urban
Prof. Dr. Heiko Peuscher
Prof. Dr. Gianluigi Rozza

Supervisor:
Nicola Demo

2020

Version June 19, 2020

Acknowledgment

This is a Masters's thesis at Ulm university written at *Scuola Internazionale Superiore di Studi Avanzati* (SISSA) in Trieste at the mathLab department.

I want to thank Prof. Dr. Karsten Urban for providing contact to Prof. Dr. Gianluigi Rozza, which made it possible to write the master thesis in Italy.

And I want to thank Prof. Dr. GianIluigi Rozza for welcoming me at SISSA. I also want to thank him for moving me to a new office because of a nerve-wrecking sound that appeared after a few weeks at the first office.

A special thanks to my supervisor Nicola Demo for always taking time to listen and help me with any problem. He continuously supported me and gave me good advice in my process of working on my thesis. And thanks for teaching me how to play briscola after lunch.

Towards the end of my stay in Italy, I was confronted with challenges that came up with the coronavirus pandemic. I was disappointed that the coronavirus forced me to leave earlier as planned. But because I had to leave on short notice, I had to leave some of my things behind, which I am not particularly sad about since it gives me a reason to come back soon.

Abstract

The goal of this master thesis is to present a parallel approach of the Proper Orthogonal Decomposition (POD) and to integrate it into a reduced order modeling framework.

In the reduced-order modeling community, the POD is a popular snapshot-based method to obtain reduced basis functions. In most reduced-order molding frameworks, the computation of the POD is done in serial even though the snapshots are computed on distributed and parallel systems like HPC supercomputers. This leads to being the bottle-neck in many reduced order modeling frameworks, that can be solved by parallel implementation.

For this master thesis, we took a look at two different approaches to compute the POD in parallel. We compared a method using the singular value decomposition to a method solving the eigenvalue problem of the correlation matrix. The latter method has shown to be the best method in terms of computational cost and speedup. We were also able to increase the performance of the method by introducing a hybrid implementation combining a multithread with a multi-process paradigm. The proposed parallel implementation enables the exploitation of HPC supercomputers also for POD computation, and not only for the truth solutions.

The parallel algorithm is integrated into the reduced order modeling framework EZyRB, a framework for data-driven reduced order modeling, which has been exploited for the numerical results proposed in this work.

Contents

List of Figures

1 Introduction

In computational sciences and engineering, one often has to deal with very complex phenomena. In many cases, Partial Differential Equations (PDE) are used to model these complex problems.

Parametric Partial Differential Equations

PDEs are used to describe and solve problems in structural analysis, heat transfer, fluid flow, mass transport, or electromagnetic potential. Popular methods to solve PDEs are finite difference method (FDM), finite element method (FEM), finite volume method (FV), and spectral element method (SEM). Usually, these high order methods have a high computational cost and are mostly computed on distributed and parallel systems like HPC supercomputers. In many cases, the PDE depends on parameters, then we are talking about parametric PDEs. The problem with parametric PDEs is that we have to solve the PDE for every parameter of interest using a high order method. This leads to a high computational cost and a long evaluation time of the model. For the fast evaluation of parametric PDEs we can make use of Reduced Order Methods.

Reduced Order Methods

Reduced Order Methods (ROMs) are a category of methods and techniques used to reduce PDE's complexity. Models with reduced complexity can be evaluated more efficient than high order models. The fast evaluation of parametric PDEs can be of high interest in many fields of science and industry, when it comes to optimizing or controlling the parameter or using them in real-time systems. For example, ROMs are used in Computational Fluid Dynamic (CFD), in environmental sciences [15], in control problems over dynamical systems [13] and in problems with applications in finance.

Reduced basis (RB) methods are projection-based model order reduction techniques for reducing the computational complexity of solving parametric partial differential equation problems. RB methods project the problem from a high dimensional space onto a low dimensional reduced space. Common methods to compute these reduced spaces are the greedy algorithm and the Proper Orthogonal Decomposition.

The reduced problems created with RB methods have much less computational cost in comparison to full order methods. An HPC supercomputer is no longer needed. The reduced models can be computed on less powerful hardware like a laptop or even on smartphones or tablets.

Proper Orthogonal Decomposition

In this work, we will focus on the creation of a reduced basis using the Proper Orthogonal Decomposition (POD). The POD is also known as Karhunen–Loève decomposition (KL) or Principal component analysis (PCA). It is well known in the analysis of turbulent flow [9], but can also be used to create a reduced basis.

The basic idea of the POD is to extract the dominant information of a given data set using eigenvalue decomposition or singular value decommission. And then use the most dominant information to create the reduced space. In most reduced-order modeling frameworks, the computation of the POD is done in serial even though the snapshots are computed on distributed and parallel systems like HPC supercomputers. This leads to being the bottle-neck in many reduced order modeling frameworks. This bottle-neck can be resolved with a parallel implementation.

We present two different methods to compute the POD in a parallel way. One method is based on the singular value decomposition and was presented by Kunisch and Volkwein in 2007 [1]. The second method is based on solving the eigenvalue problem of the correlation matrix.

These two methods are implemented in parallel, and benchmarks of the parallel implementation are done on an HPC supercomputer.

Reduced Order Modeling Framework

Based on these benchmarks, we choose the best method and integrate it into the EZyRB (Easy Reduced Basis method) package [6]. EZyRB is an open-source python package for data-driven model order reduction. Due to the modular structure of EZyRB it can be used in computational pipeline with model order reduction for industrial and applied mathematics [16].

EZyRB is a serial software for data-driven model order reduction. Using it in a computational pipeline, it computes a reduced model after high order solvers compute the data for the reduction. Usually, the high order solvers are able to make use of an HPC supercomputer. With a parallel POD algorithm, EZyRB is also able to make use of these resources.

2 Reduced Basis Method

In this chapter, we give an overview on reduced basis (RB) methods. RB methods are used when we are interested in efficient solutions of the same parametric PDE for different parameters. This can be the case in multi-queries applications like optimization or in real-time applications.

The workflow of RB methods can be divided into two phases, the offline-phase and the online-phase.

1. In the offline-phase, we create a reduced model of the parametric PDE. To create this reduced model we need so called truth solutions. These truth solutions are very good approximations of the exact solutions. We assume that the approximation is sufficiently good so that we can call them truth solutions. We create the truth solutions using a discretization technique. Finite Element (FEM) and Finite Volume (FV) are popular discretization techniques to compute the truth solutions. The most commonly used software packages by the scientific community are FEniCS [11] for FEM and OpenFOAM [18] for FVM. These truth solutions have a high computational cost and they are usually calculated on a HPC supercomputer. The truth solutions are used to create the reduced model. In this chapter we will describe how to create a reduced model using the proper orthogonal decomposition.
2. In the online-phase, we solve the reduced model that we created in the offline-phase. This reduced problem requires much lower computational cost with respect to the previous phase. Solving the reduced problem can be done on less powerful hardware like a laptop, tablet, or smartphone.

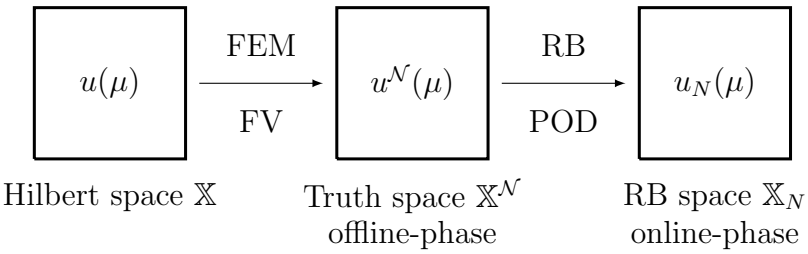

Figure 2.1: Sketch of the RB workflow. Starting from the exact solution of the PDE $u(\mu)$. Then the offline-phase, where we are creating snapshot solution $u^{\mathcal{N}}(\mu)$ in the truth space $\mathbb{X}^{\mathcal{N}}$ using FEM or FV. And finally, in the online-phase where we are computing reduced solutions $u_N(\mu)$ in the RB space $\mathbb{X}_N$.

Before we start, a view remarks to the notation. Everything associated with the exact solution has no indices. Everything associated with the truth space has a superscripted calligraphic $\mathcal{N}$ $\left(\cdot^{\mathcal{N}}\right)$. Everything associated with the RB space has a subscripted normal N $(\cdot_N)$.

We start with the weak formulation of a parametric PDE

$$\text{find } u(\mu) \in \mathbb{X}: \quad a(u(\mu), v; \mu) = f(v; \mu) \quad \forall v \in \mathbb{X}. \tag{2.1}$$

Where$\mathbb{X}$ is a Hilbert space in which we search for the solution, $\mu \in \mathbb{R}^n$ is the parameter vector containing all the parameters that describe our parametrized PDE, $a(\cdot, \cdot, \mu)$ is a parametric bilinear form containing the equation describing the physics, $f(\cdot, \mu)$ is a parametric linear form containing the source term on the right-hand side and $u(\mu) \in \mathbb{X}$ is the exact solution depending on μ.

In a weak formulation, an equation is no longer required to be well defined and has instead weak solutions only with respect to certain "test functions". This is equivalent to formulating the problem to require a solution in the sense of a distribution. In Section 5.2.1 we make an example of the weak formulation for the heat equation.

The exact solution is not available in most cases, so first we need to approximate the solution as good as possible. Because the Hilbert spaces are usually ∞-dimensional, we need a sufficient fine discretization. FEM and FV are popular discretization techniques to compute the solutions. We call solutions, which are computed by these methods truth solutions.

These methods discretise the ∞-dimensional space by choosing a finite number of basis functions $\{\varphi_i^{\mathcal{N}}\}_{i=1}^{\mathcal{N}} \in \mathbb{X}$. The basis functions $\varphi_i^{\mathcal{N}}$ span the truth space $\mathbb{X}^{\mathcal{N}} = \text{span}\{\varphi_1^{\mathcal{N}}, ..., \varphi_{\mathcal{N}}^{\mathcal{N}}\} \subset X$. The choice of these basis functions depends on the discretization technique. The truth solutions are denoted with $u^{\mathcal{N}}(\mu)$ where $\mathcal{N}$ stands for the degrees of freedom of the high fidelity solution. The truth solution can be represented as a linear combination of basis functions

$$u^{\mathcal{N}}(\mu) = \sum_{i=1}^{\mathcal{N}} u_i^{\mathcal{N}}(\mu) \cdot \varphi_i^{\mathcal{N}}, \tag{2.2}$$

where $u_i^{\mathcal{N}}(\mu)$ are coefficients that scale the basis functions in the linear combination. Using the discretization, the truth problem can now be written as

$$\text{find } u^{\mathcal{N}}(\mu) \in \mathbb{X}^{\mathcal{N}} : \quad a(u^{\mathcal{N}}(\mu), v; \mu) = f(v; \mu) \quad \forall v \in \mathbb{X}^{\mathcal{N}}. \tag{2.3}$$

Using the discrete basis functions, the truth problem (2.3) leads to the discrete system

$$\underline{A}^{\mathcal{N}}(\mu)\underline{u}^{\mathcal{N}}(\mu) = \underline{f}^{\mathcal{N}}(\mu),$$

with:

- The stiffness matrix $\mathbf{A}^{\mathcal{N}}(\mu) = [a(\varphi_i, \varphi_j, \mu)]_{i=1,...,\mathcal{N}}$
- The right hand side vector $\underline{f}^{\mathcal{N}}(\mu) = [f(\varphi_j, \mu)]_{j=1,...,\mathcal{N}}$
- The solution vector $\underline{u}^{\mathcal{N}}(\mu) = [u_i^{\mathcal{N}}(\mu)]_{i=1,...,\mathcal{N}}$ containing the coefficients $u_i^{\mathcal{N}}(\mu)$.

We define the discrete solution manifold

$$\mathbb{M}^{\mathcal{N}} = \{u^{\mathcal{N}}(\mu), \quad \mu \in \mathbb{P}\}.$$

as the set of all truth solution varying the parameter.

In many cases it is too expensive to compute a truth solution for all parameters we are interested in. The aim of RB methods is to approximate the solution manifold as good as possible but also efficiently.

The idea is to provide a projection to the manifold on low dimensional space using a small number of basis functions $\{\xi_i\}_{i=1}^{N}$.

A small number means that the number of basis functions is a lot smaller than the degrees of freedom of the high fidelity solution $N \ll \mathcal{N}$. The basis function spans the RB space $\mathbb{X}_N = \text{span}(\xi_1, ..., \xi_N) \subset \mathbb{X}^{\mathcal{N}}$. Compared to (2.2) for the reduced solution, we want to use N basis functions from the reduced basis instead of $\mathcal{N}$ basis functions from the truth basis. The reduced solutions are defined as a linear combination of reduced basis functions

$$u_N(\mu) = \sum_{i=1}^{N} u_{N,i}(\mu)\xi_i,$$

where $u_{N,i}(\mu)$ are the coefficients of the reduced system.

The reduced basis functions are build as a linear combination of truth basis functions

$$\xi_i = \sum_{k=1}^{\mathcal{N}} b_{k,i}\varphi_k^{\mathcal{N}}, \tag{2.4}$$

with the coefficients $b_{k,i}$. These coefficients are stored in the matrix $B \in \mathbb{R}^{\mathcal{N}\times N}$. The i-th column of B contains the coefficients for the i-th basis function ξ_i [8, Section 3.1]. A way to compute this basis, respectively the coefficients $b_{k,i}$, is the proper orthogonal decomposition (POD). The POD will be described in detail in Section 2.1.

Using the reduced basis functions, we can create the reduced system

$$\mathbf{A}_N(\mu)\underline{u}_N(\mu) = \underline{f}_N(\mu)$$

with:

- The reduced stiffness matrix $\mathbf{A}_N(\mu) = [a(\xi_i, \xi_j; \mu)]_{i,j=1,...,N}$
- The reduced right hand side vector $\underline{f}_N(\mu) = [f(\xi_j; \mu)]_{j=1,...,N}$
- The solution vector $\underline{u}_N(\mu) = [u_{N,i}(\mu)]_{i=1,...,N}$ containing the coefficients $u_{N,i}(\mu)$

To compute the reduced stiffness matrix $\mathbf{A}_N(\mu)$ we insert the reduced basis functions into the bilinear form $a(\xi_i, \xi_j, \mu)$. Since it is a bilinear form, we are able to write it as

$$a(\xi_i, \xi_j, \mu) = \sum_{k=1}^{\mathcal{N}}\sum_{l=1}^{\mathcal{N}} b_{i,k}b_{j,l}a(\varphi_i, \varphi_j; \mu).$$

Computing this for every given μ would be a very high computational effort. We assume

the bilinear form $a(\cdot,\cdot,\mu)$ is affine in the parameter. Which means we can decompose it in functions $\theta_a^q(\mu)$ containing the parameter μ and bilinear forms $a^q(\varphi_i,\varphi_j)$ independent from the parameter. We call this the affine decomposition

$$a(\xi_i,\xi_j,\mu)=\sum_{q=1}^{Q_a}\theta_a^q(\mu)\sum_{k,l=1}^{\mathcal{N}}b_{i,k}b_{j,l}a^q(\varphi_i,\varphi_j). \tag{2.5}$$

An example of such an affine decomposition is done in Section 5.2.1. The last sum in (2.5) is independent from μ and can be computed in the offline phase. It can be written in matrix form

$$\mathbf{A}_N^q=\mathbf{B}^T\mathbf{A}^{\mathcal{N},q}\mathbf{B},$$

where the matrices $\mathbf{A}^{\mathcal{N},q}$ are the parameter independent parts of the truth system. With this operation we project the coefficients of the truth system into the reduced space. The same is done for the right hand side part $\underline{f}_N^q=\mathbf{B}^T\underline{f}^{\mathcal{N},q}$.

$\mathbf{A}_N(\mu)$ and $\underline{f}_N(\mu)$ can be build efficiently in the online phase

$$\mathbf{A}_N(\mu)=\sum_{q=1}^{Q_a}\theta_a^q(\mu)\mathbf{A}_N^q,\qquad\qquad \underline{f}_N(\mu)=\sum_{q=1}^{Q_f}\theta_f^q(\mu)\underline{f}_N^q.$$

If there exists no affine decomposition, use Discrete Empirical Interpolation (DEIM) [8, Chapter 5]. More about DEIM can be found in [2].

The reduced solution should be approximately equal $u_N(\mu)\approx u^{\mathcal{N}}(\mu)$ to the truth solution. Of course, there will be an error. In this work, we will not go more in-depth on errors and error estimators.

The primary motivation of this work is to provide a parallel version of the POD. In the offline-phase, the high fidelity solutions are usually computed on a supercomputer. Despite POD algorithm usually has negligible computational cost, in many different distributed and parallel systems it may become the bottle-neck. The proposed parallel implementation enables the exploitation of HPC supercomputers for both, POD computation and the truth solutions.

2.1 Proper Orthogonal Decomposition

In this Section, we present the Proper Orthogonal Decomposition (POD), the numerical method adopted to extract the reduced basis in this thesis. POD is a widespread approach in RB community.

Theoretically, we could determine an optimal space by solving an optimization problem. In most cases, this is very hard or impossible. Therefore, strategies based on training sets are used. The POD is such a strategy.

We define the set of all training parameters $\mathbb{P}_{train} = \{\mu_1^{train}, ..., \mu_{n_{train}}^{train}\}$ with n_{train}, the number of training parameters and the space containing all solutions $\mathbb{X}_{train} := \text{span}\{u(\mu^{train}) : \mu^{train} \in \mathbb{P}_{train}\}$.

The POD space $\mathbb{X}_N$ is defined in [8, Section 3.2.1 (3.8)] as:

$$\mathbb{X}_N := \underset{\mathbb{X}_N \subset \mathbb{X}_{train}}{\arg\inf} \sqrt{\frac{1}{n_{train}} \sum_{\mu \in \mathbb{P}_{train}} \inf_{w_N \in \mathbb{X}_N} \|u^{\mathcal{N}}(\mu) - w_N\|_{\mathbb{X}}^2}. \tag{2.6}$$

The expression $\inf_{w_N \in \mathbb{X}_N} \|u^{\mathcal{N}}(\mu) - w_N\|_{\mathbb{X}}^2$ describes the error we make representing the truth solution $u^{\mathcal{N}}(\mu)$ with functions out of the POD space $w_N \in \mathbb{X}_N$. Then this error is summed up for all parameters in the training set. The POD space is defined as the space that minimizes the sum of errors.

We search the POD space $\mathbb{X}_N$ as an N-dimensional subspaces of $\mathbb{X}_{train}$. Because $\mathbb{X}_{train}$ is spanned by the truth solutions, the POD basis functions are defined as linear combinations of the truth solutions

$$\xi_i = \frac{1}{\sqrt{\lambda_i}} \sum_{n=1}^{n_{train}} v_{n,i} u^{\mathcal{N}}(\mu_n), \tag{2.7}$$

with the coefficients $v_{n,i}$. The functions ξ_i span the POD space $\mathbb{X}_N := \text{span}\{\xi_i : 1 \leq i \leq N\} \subset \mathbb{X}_{train}$

To find the functions ξ_i that are spanning a space $\mathbb{X}_N$ that fulfills the $\arg\inf_{\mathbb{X}_N \subset \mathbb{X}_{train}}$ requirement of (2.6), in [8, Section 3.2.1] and [10, Section 3.1] the symmetric and linear

operator

$$C(v) = \frac{1}{n_{train}} \sum_{m=1}^{n_{train}} (v, u^{\mathcal{N}}(\mu_m))_{\mathbb{X}} u^{\mathcal{N}}(\mu_m), \quad v \in \mathbb{X}_N.$$

is defined. Solving the eigenvalue problem of this operator

$$\big(C(\xi_n), u^{\mathcal{N}}(\mu_m)\big)_{\mathbb{X}} = \lambda_n (\xi_n, u^{\mathcal{N}}(\mu_m))_{\mathbb{X}}, \quad 1 \leq m \leq n_{train}, \tag{2.8}$$

gives us the eigenvalue-eigenfunction pairs (λ_n, ξ_n). These eigenfunctions span the space $\mathbb{X}_N$ that derive the optimality conditions for the optimization problem (2.6) [10, 17].

Because we are computing the POD basis functions solving an eigenvalue problem, these functions are orthogonal. This is a big advantage of the POD.

We further assume that eigenvalues are in decending order $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_{n_{train}}$. Then for the POD error, it applies that it is equal to the sum of the smallest eigenvalues [10]:

$$\epsilon_N^{POD} = \frac{1}{n_{train}} \sum_{\mu \in \Xi_{train}} \inf_{w_N \in \mathbb{X}_N} \|u^{\mathcal{N}}(\mu) - w_N\|_{\mathbb{X}}^2 = \sum_{n=N+1}^{n_{train}} \lambda_n. \tag{2.9}$$

We can truncate N smallest eigenvalues so that ϵ_l^{POD} is less than a given error boundary. That we can state the error done by the truncation is another bit advantage of the POD. The remaining eigenfunctions span the POD space $\mathbb{X}_N = \text{span}\{\xi_n : 1 \leq n \leq N\}$.

Computing the POD

To compute the POD we introduce the snapshot matrix $\mathcal{W}$, defined as:

$$\mathcal{W} = \begin{bmatrix} | & | & & | \\ \underline{u}^{\mathcal{N}}(\mu_1) & \underline{u}^{\mathcal{N}}(\mu_2) & \ldots & \underline{u}^{\mathcal{N}}(\mu_{n_{train}}) \\ | & | & & | \end{bmatrix}.$$

The snapshot matrix contains the solution vectors $\underline{u}^{\mathcal{N}}(\mu_i)$ one row for every $\mu \in \Xi_{train}$. The snapshot matrix has the dimension $\mathcal{W} \in \mathbb{R}^{\mathcal{N} \times n_{train}}$ with $\mathcal{N} \gg n_{train}$. $\mathcal{N}$ is the number of degrees of freedom and n_{train} is the number of snapshots.

We construct the correlation matrix $\underline{C}$ as the inner product of all parameter combinations $\mu \in \Xi_{train}$. Because $\mu \in \Xi_{train}$ and Ξ_{train} contains n_{train} elements, $\underline{C}$ has the dimension $n_{train} \times n_{train}$. The i-th column and the j-th row of the matrix $\underline{C}$ are defined as

$$\underline{C}_{i,j} = \frac{1}{n_{train}}(u(\mu_i), u(\mu_j))_{\mathbb{X}}. \tag{2.10}$$

We define $G = [(\varphi_k^{\mathcal{N}}, \varphi_l^{\mathcal{N}})_{\mathbb{X}}]_{1 \leq k,l \leq \mathcal{N}} \in \mathbb{R}^{\mathcal{N} \times \mathcal{N}}$ as the matrix containing the inner products of all basis functions. Using G and (2.2) we get the representation

$$\begin{aligned}\underline{C}_{i,j} &= \frac{1}{n_{train}} \sum_{k=1}^{\mathcal{N}} \sum_{l=1}^{\mathcal{N}} u_k(\mu_i) u_l(\mu_j) (\varphi_k^{\mathcal{N}}, \varphi_l^{\mathcal{N}})_{\mathbb{X}} \\ &= \frac{1}{n_{train}} \underline{u}^{\mathcal{N}}(\mu_i)^T G \underline{u}^{\mathcal{N}}(\mu_j).\end{aligned}$$

We stored $\underline{u}^{\mathcal{N}}(\mu_i)$ in the snapshot matrix $\mathcal{W}$, so we can compute the matrix $\underline{C}$ using a matrix-matrix product

$$\underline{C} = \frac{1}{n_{train}} \mathcal{W}^T G \mathcal{W}.$$

Solving the eigenvalue problem of $\underline{C}$

$$\underline{C} v_n = \lambda_n v_n, \quad 1 \leq n \leq n_{train},$$

is equivalent to the eigenvalue problem (2.8) [8, Section 3.2.1, Linear algebra box].

The POD basis functions ξ_i are given as linear combinations of the snapshots $u^{\mathcal{N}}(\mu_n)$

$$\xi_i = \frac{1}{\sqrt{\lambda_i}} \sum_{n=1}^{n_{train}} v_{n,i} u^{\mathcal{N}}(\mu_n) = \frac{1}{\sqrt{\lambda_i}} \sum_{n=1}^{n_{train}} \sum_{j=1}^{\mathcal{N}} v_{n,i} u_j^{\mathcal{N}}(\mu) \cdot \varphi_j^{\mathcal{N}}$$

where $v_{n,i}$ is the n-th entry of the i-th eigenvector [8, Section 3.2.1, Linear algebra box].

To represent the POD basis functions as in (2.4) we compute the matrix B. The coefficients are defined as $b_{i,k} = \frac{1}{\sqrt{\lambda_i}} v_{n,i} u_j^{\mathcal{N}}(\mu)$. We get the matrix representation

$$B = WVD^{-1/2}.$$

The matrix product WV is equivalent to the linear combination $v_{n,i}u_j^{\mathcal{N}}(\mu)$ for all N basis functions. The multiplication of the matrix $D^{-1/2}$ is the scaling factor $\frac{1}{\sqrt{\lambda_i}}$. This operation is cheap because D is a diagonal matrix with the eigenvalues on its diagonal. The matrix B contains the coefficients for the i-th POD basis functions ξ_i in its i-th column.

2.2 Singular Value Decomposition

The Singular Value Decomposition (SVD) of $\mathcal{W} \in \mathbb{R}^{\mathcal{N} \times n_{train}}$ is given through

$$\mathcal{W} = USV^T \tag{2.11}$$

with the left singular vectors $U \in \mathbb{R}^{\mathcal{N} \times n_{train}}$ and the right singular vectors $V \in \mathbb{R}^{n_{train} \times n_{train}}$, U and V are both orthonormal matrices.

When the scalar product of the space $\mathbb{X}$ in equation (2.10) is equal to the L^2 scalar product, then the eigenvalues of $\mathcal{W}^T\mathcal{W}$ are equal to the square of the singular vectors V of the snapshot matrix $\mathcal{W}$ [8, Section 3.2.1, Linear algebra box]. The POD modes then are the first N columns in U [1].

The connection between eigenvalues and singuar values is the following:

$$\mathcal{W}^T\mathcal{W} \overset{1.)}{=} (USV^T)^T USV^T \overset{2.)}{=} VSU^T USV^T \overset{3.)}{=} VS^2V^T. \tag{2.12}$$

In 1.), we replace the snapshot matrix $\mathcal{W}$ with its SVD. In 2.), we apply the transpose. This leads to U^TU, and because U is an orthonormal matrix, U^TU is equal to the identity matrix. In 3.), we cancel out U^TU because of the orthonormality. This leads to VS^2V^T, which is the eigendecomposition. It follows that the eigenvalues of $\mathcal{W}^T\mathcal{W}$ are equal to the square of the singular values of $\mathcal{W}$ and the eigenvectors of $\mathcal{W}^T\mathcal{W}$ are equal to the right singular vectors V of $\mathcal{W}$.

Because we are interested in the left singular vectors U, we rearrange (2.11). We can rearrange it, because we have computed the matrices U and S using the eigenvalue problem. We bring V^T on the other side by multiplying with V and S by multiplying with S^{-1}.

$$U = \mathcal{W}VS^{-1} \tag{2.13}$$

3 Parallel computation of the POD

3.1 Parallel POD using SVD

The method that is described in this Section was introduced in [1]. This approach aims to provide a parallel version using the SVD. The SVD of the snapshot matrix $\mathbb{W}$ is given through $\mathcal{W} = \mathcal{U}\mathcal{S}\mathcal{V}^T$, with the unitary $\mathcal{U}$ containing the left singular vectors, the unitary $\mathcal{V}^T$ containing the right singular vectors and the diagonal matrix $\mathcal{S}$ containing the singular values on its diagonal.

Instead of computing the SVD of the whole snapshot matrix, the snapshot matrix is partitioned, and the parts are scattered to the available processors. Then every processor computes the right singular vectors of its part. The singular vectors of all parts are then used to approximate the right singular vectors of the whole snapshot matrix. The left singular vectors u_i of $\mathcal{U}$, can be obtained by post-multiplying the snapshot matrix with the i-th column v_i of $\mathcal{V}$, since

$$\mathcal{W}v_i = \mathcal{U}\mathcal{S}\mathcal{V}^T v_i = s_{ii}u_i. \tag{3.1}$$

We now describe the method in more detail. We assume that we are just computing the N left singular vectors with the N largest singular values. N is less or equal to the number of snapshots $N \leq n_{train}$. The snapshot matrix is scattered to all processors. We denote n_p the number of processors. Therefore, the snapshot matrix $\mathcal{W} \in \mathbb{R}^{\mathcal{N}\times n_{train}}$ is partitioned in blocks with K rows, choosing K so that $n_p \times K = \mathcal{N}$. We obtain

$$\mathcal{W} = [W_1^T ... W_{n_p}^T]^T,$$

with $W_k \in \mathbb{R}^{N\times n_{train}}$, $k = 1, ..., n_p$.

Each processor computes the local right singular vectors V_k and its part of the correlation

matrix C_k.

$$\text{svd}(W_k) = [U, S, V_k] \tag{3.2}$$

$$C_k = W_k^T \cdot W_k \tag{3.3}$$

The N right singular vectors of V_k with the largest singular values and C_k are sent to the master process. This communication is cheap because C_k has the small dimension $n_{train} \times n_{train}$ and we only need to send the N first columns of V_k, so the dimension is $n_{train} \times N$.

The master process is now computing an approximation of the right singular values $\mathcal{V}$. Therefore, the matrix $\hat{\mathcal{V}}$ is constructed by writing the right singular vectors V_k column wise in a matrix

$$\hat{\mathcal{V}} = [V_1 ... V_{n_p}].$$

This matrix contains the right singular vectors of all parts of the snapshot matrix. To compute the approximation of the right singular vectors of the global snapshot matrix, the right singular vectors of $\hat{\mathcal{V}}$ are computed with

$$\text{svd}(\hat{\mathcal{V}}) = [\mathcal{T}, \mathcal{M}, \mathcal{V}].$$

The first N columns of $\mathcal{V}$ are a good approximation of the global singular vectors.

The master process also computes the global correlation matrix.

$$\mathcal{C} = \sum_{i=1}^{n_p} C_k. \tag{3.4}$$

Because the snapshot matrix was partitioned row wise, the global correlation matrix is the sum of all local correlation matrices C_k.

We have now obtained a good approximation of the first N global right singlar vectors. We can obtain the first N global left singlar vectors using (3.1). Therefore, we want to compute

$$\text{svd}(\mathcal{W}\mathcal{V}) = [\mathcal{U}, \mathcal{S}, \mathcal{Z}].$$

Computing svd($\mathcal{WV}$) globally is to expensive. We can make use of the correlation matrix to reduce the computational effort. As we have shown in (2.12) we can solve svd($\mathcal{WV}$) over the eigenvalue problem of the correlation matrix

$$(\mathcal{WV})^T(\mathcal{WV}) = \mathcal{V}^T\mathcal{W}^T\mathcal{WV} = \mathcal{V}^T\mathcal{CV} = \mathcal{Z}\mathcal{S}^2\mathcal{Z}.$$

We have already computed the correlation matrix $\mathcal{C}$ in (3.3) and (3.4).

Computing $\mathcal{V}^T C\mathcal{V}$ and solving the eigenvalue problem $\mathcal{V}^T C\mathcal{V} = \mathcal{Z}\mathcal{S}^2\mathcal{Z}$ is cheap because of the small dimension. After solving the eigenvalue problem we can compute $\mathcal{VZS}^{-1}$ on the master process so we just have to send a small $n_{train} \times N$ matrix.

Every process can now compute its part of the left singular vectors

$$\mathcal{U}_k = \mathcal{W}_k\mathcal{VZS}^{-1}.$$

A big disadvantage of this method is that we only compute N singular vectors but not the singular values, so we can't state anything about the error we make with the truncation.

3.2 Parallel POD by solving an eigenvalue problem

In this Section we describe the parallel aspects of computing the POD by solving the eigenvalue problem of the correlation matrix, as described in Section 2.1.
In Section 2.1 we saw that we need to solve the eigenvalue problem of the correlation matrix

$$\mathcal{W}^T\mathcal{W} = VDV^T, \tag{3.5}$$

to obtain the POD vectors

$$U = \mathcal{W}VD^{-\frac{1}{2}}. \tag{3.6}$$

The expensive operation in terms of computational effort for this method are the parts of the algorithm where we have to deal with the big dimension $\mathcal{N}$. These operations are:

- The computation of the correlation matrix $\mathcal{W}^T\mathcal{W}$. The snapshot matrix has the dimension $\mathcal{W} \in \mathbb{R}^{\mathcal{N}\times n_{train}}$. The computational effort is $O(\mathcal{N}^2 \cdot n_{train})$. The dimension of the truth solutions $\mathcal{N}$ has a quadratic influence. This makes this operation expensive.
- The multiplication of VS^{-1} on the snapshot matrix $\mathcal{W}$. After the truncation the matrix VS^{-1} has the dimension $\mathcal{W} \in \mathbb{R}^{n_{train}\times N}$. The computational effort is $O(\mathcal{N} \cdot n_{train} \cdot N)$. In this case the $\mathcal{N}$ has no quadratic influence but we still have to deal with this large dimension. n_{train} and N are proportionately small.

Luckily both operation are matrix products and can be parallelized.

The cheap operations in terms of computational effort for this method are the operations where we have not the big dimension $\mathcal{N}$. These operations are:

- Solving the eigenvalue problem of the correlation matrix, because of small dimension of $\mathcal{W}^T\mathcal{W} \in \mathbb{R}^{n_{train}\times n_{train}}$.
- VS^{-1} is cheap because of the small dimensions and S is a diagonal matrix. Since S is a diagonal matrix the inversion is just the multiplicative inverse of the diagonal elements.

We can call these operations cheap because they can be done on a single processor, and there is no time-consuming communication.

3.2.1 Parallel computation of the correlation matrix

For the parallel computation of the correlation matrix, we make use of the special dimension of the snapshot matrix. The snapshot matrix $\mathcal{W}$ has the dimension $\mathcal{N} \times n_{train}$.

We divide the matrix into blocks along the rows. We can now compute the correlation matrix for each block independent from the other blocks. The global correlation matrix can be computed by summing up the correlation matrices of the blocks. This is cheap because the local correlation matrices and the global correlation matrix have the small dimension $n_{train} \times n_{train}$.

$$C = W^T \cdot W = \begin{bmatrix} w_1^T & \dots & w_n^T \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} = w_1^T \cdot w_1 + \dots + w_n^T \cdot w_n = \sum_{i=1}^{n} w_i^T \cdot w_i \quad (3.7)$$

3.2.2 Solving the eigenvalue problem

After solving the eigenvalue problem (3.5) we compute $\hat{V} = VD^{-\frac{1}{2}}$. This is also very cheap because S is a diagonal matrix with the eigenvalues λ_i on its diagonal. We only need to scale the i-th row of V with $\frac{1}{\lambda_i}$. We use $\hat{V}$ to compute the POD vectors as in (3.6)

3.2.3 Parallel computation of U

Because W is already blocked, we only have to multiply $\hat{V}$ on each block of W from the right to compute U.

$$U = W \cdot VS^{-1} = W \cdot \hat{V} = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} \cdot \hat{V} = \begin{bmatrix} w_1 \cdot \hat{V} \\ \vdots \\ w_n \cdot \hat{V} \end{bmatrix} \quad (3.8)$$

4 Parallel implementation

In this chapter, we will explain the technical details of the parallel implementation. First, we will introduce the used technologies/techniques and libraries. Then we will take a detailed look at the parallelization.

4.1 Parallelization techniques

4.1.1 MPI - Message Passing Interface

MPI (Message Passing Interface) is a standard for a library interface for parallel programs. MPI is process-based, which means the interface provides a standard to send a message from one process to another process. The processes can be running on the same machine or on distributed systems like multi-computers or clusters.

If the processes are running on the same machine, MPI does not support shared memory. Sending a message from one process to another on the same machine, is a copy operation in the RAM, because the processes have no access to the virtual memory of the other processes.

If the processes are running on distributed systems, MPI sends the messages using the network. Common networking communications standards are InfiniBand used in high performance clusters, or ethernet and TCP/IP used for the internet.

Messages that can be sent with MPI are sequential arrays of standard data types like integer, floating-point numbers, or characters. Also, more complex data types can be defined. In this work, we send sequential arrays of the type floating-point number.

MPI provides many functions to send data. In this work, we will use the following functions:

- MPI_Bcast: This function broadcasts a sequential array from one process to all other processes. After applying this function, the content of the array is the same on every processor.

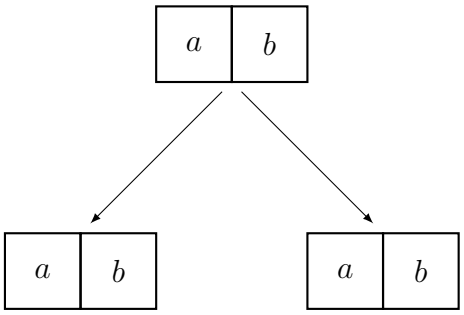

Figure 4.1: MPI_Bcast

- MPI_Reduce: This function takes a sequential array of the same size on every process, sends the data to one process, and applies a function on the data. Possible functions are for example: the minimum, the maximum or the sum. The function is applied to each element of the array, not on the array itself. In this work, we use the sum function.

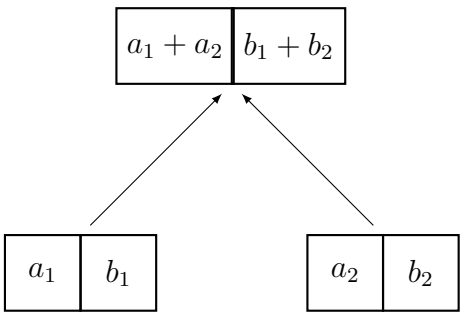

Figure 4.2: MPI_Reduce with MPI_sum

- MPI_Scatter: This function scatters a sequential array to all processors. Scatter, in this context, means the array is divided into sequential blocks, and every block is sent to another process.

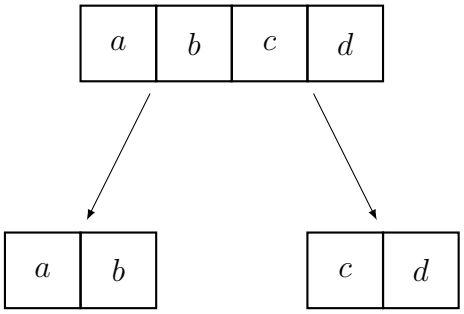

Figure 4.3: MPI_Scatter

- MPI_Gather: This function gathers sequential arrays from all processors. Gather in this context means that there are sequential arrays on every process, and these arrays are gathered to one big sequential array on one processor.

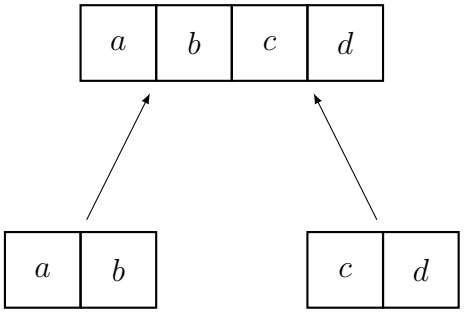

Figure 4.4: MPI_Gather

Scatter and Gather with specified location and size

MPI_Scatter and MPI_Gather divide the array into equally sized blocks. MPI provides functions where the block sizes can be specified. These functions are denoted with a v at the end, MPI_Scatterv and MPI_Gatherv. The array sizes are defined using two additional arrays that have to be passed to the functions. The fist array is called *sendcount* and contains the block sizes. The second array is called *displacement* and contains the starting positions of the blocks.

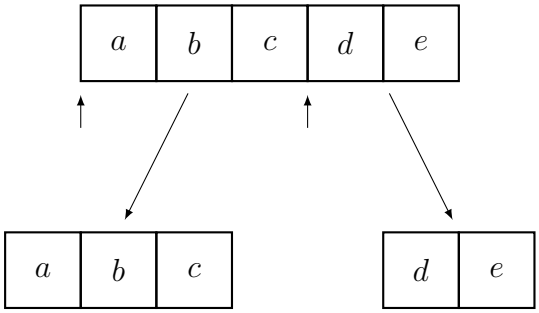

Figure 4.5: MPI_Scatterv sendcount = $[3, 2]$, displacement = $[0, 3]$

Non-blocking communication with MPI

The described functions are using blocking communication. Blocking communication means that the process is blocked until the communication is finished. During the communication, the computing power of the process is unused. To prevent this, MPI provides the most communication functions additionally, using non-blocking communication. These functions are marked with an I at the beginning. For example: MPI_Scatterv becomes MPI_Iscatterv.

4.1.2 OpenMP - Open Multi-Processing

OpenMP (Open Multi-Processing) is an application programming interface (API) for multi thread and shared memory programming. Compared to MPI OpenMP is multi thread. A process can have multiple threads running parallel on the same machine. The threads have access to the virtual memory of their related process. In this work we make use of OpenMP by using the multi thread property of the Eigen library.

4.1.3 Eigen

For the basic linear algebra we use Eigen [7]. Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.
Eigen provides the following features that are relevant for the POD:

- Data structures for dense vectors and matrices.
- Efficient implementations of the general dense matrix-matrix product which can make use of openMP.
- Efficient eigenvalue problem solvers.
- Efficient singular value decomposition solvers.

4.2 Parallel Algorithm

In this Section we discuss the parallel implementation of the algorithem described in Section 3.2.

First the snapshot matrix is partitioned and the partitions are scattered to all processors. n_p is the number of processors. In some cases the snapshot matrix $W \in \mathbb{R}^{\mathcal{N} \times n_{train}}$ is already distributed to the processors.

We partition snapshot matrix $\mathcal{W} = [w_0^T w_1^T ... w_{n_p}^T]^T$ in n_p blocks w_n. This partitioning is sketched in Figure 4.6. The number of rows per block $nr = \mathcal{N}/n_p$ is calculated by integer dividing the number of rows $\mathcal{N}$ through the number of processors n_p. It may be possible that this division generates a rest, which means that not all blocks have the same size. The remaining rows are obtained using the modulo operation $r = \mathcal{N}\%n_p$. To achieve blocks with the fairly same size, we make the first r blocks one row larger. The dimensions for the blocks w_n are $n \leq r : w_n \in \mathbb{R}^{nr+1 \times n_{train}}$ and $n > r : w_n \in \mathbb{R}^{nr \times n_{train}}$.

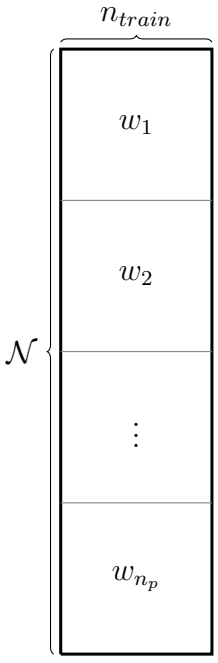

Figure 4.6: Partitioning of the snapshot matrix $\mathcal{W} \in \mathbb{R}^{\mathcal{W} \times n_{train}}$ with the parts w_n

Every block is sent so a processor. The n-th block w_n is sent to the n-th processor. This is done using the function MPI_Scatterv. Then every processor computes a local

correlation matrix $c_n = w_n^T w_n$.

The global correlation matrix is obtained by summing up all local correlations matrices. Summing up matrices is adding the corresponding elements of the matrices. The function MPI_Reduce provides this feature. The function collects data of a sequential array from each processor, adds the corresponding elements and writes the result in a sequential array on the master process. This gives us the global correlation matrix $C = \sum_{i=0}^{n_p} c_i$. We solve the eigenvalue problem of the correlation matrix on the master process $C = VS^2V$ using the SelfAdjointEigenSolver from Eigen.

Then the eigenvalues are truncated and the matrix

$$vs = VS^{-\frac{1}{2}}$$

is computed as described in Section 3.2. This matrix is sent to all processors using the function MPI_Bcast. Every processor multiplies the matrix vs on his local copy of the snapshot matrix to obtain the local POD vectors $u_n = w_n \cdot vs$.

The local POD vectors u_n are gathered to get the global POD vectors in the matrix $U = [u_0^T u_1^T ... u_{n_p}^T]^T$. This gathering is done by using the function MPI_Gatherv. A summary of the implementation is presented in Algorithm 1.

Algorithm 1 Pseudocode parallel POD

```
1:  function POD(W)
2:      MPI_Scatterv( W to w_n)
3:      Eigen matrix-matrix product c_n = w_n^T · w_n
4:      MPI_Reduce(c_n, MPI_sum) C = Σ_{i=0}^{n_p} c_i
5:      if masterProcess then
6:          Solve EVP C = VSV^T using Eigen
7:          Truncation
8:          vs = VS^{-1/2}
9:      end if
10:     MPI_Bcast( vs )
11:     Eigen matrix-matrix product u_n = w_n · vs
12:     MPI_Gatherv( u_n in U )
13:     Return U
14: end function
```

4.2.1 Non-blocking communication

In this Section we present an extension of the previous algorithm. We are sending a lot of data because $\mathcal{N}$ is very big. This leads to a long processor time. During the communication the processor is idle. We want to use this idle time by overlapping the communication and the calculations.

MPI provides for all communication functions additionally a non-blocking version. These non-blocking versions are marked with a capital I. MPI_Scatter becomes MPI_Iscatter and MPI_Gather becomes MPI_Igather. Non-blocking means that these functions start the communication in the background, so the process is able to continue working and is not blocked by the communication. The only thing we have to make sure, is that we do not use the communication buffer until the communication is finished.

In Algorithm 1 the operations in line 2 and 3 and in line 11 and 12 can be overlapped, using the non-blocking MPI_Iscatter MPI_Igather functions. We start with the lines 2 and 3.

We use the same partitioning of the snapshot matrix as in the Section before and divide every block additionally in sub-blocks. We overlap the sending of a matrix sub-block and the computation of the correlation matrix. In Figure 4.7 we sketch the partitioning and how the blocks are distributed to the processors. The snapshot matrix is stored row major, so it is no problem to block it again row wise. We call these new blocks sub-blocks. A sub-block is defined as a compact set of rows. The sub-block size bs is defined as the number of rows of a sub-block.

A block is blocked in the sub-blocks $w_n = [{w_n^{(1)}}^T, {w_n^{(2)}}^T, ..., {w_n^{(nofbs)}}^T]^T$, where $nofbs$ is the number of sub-blocks.

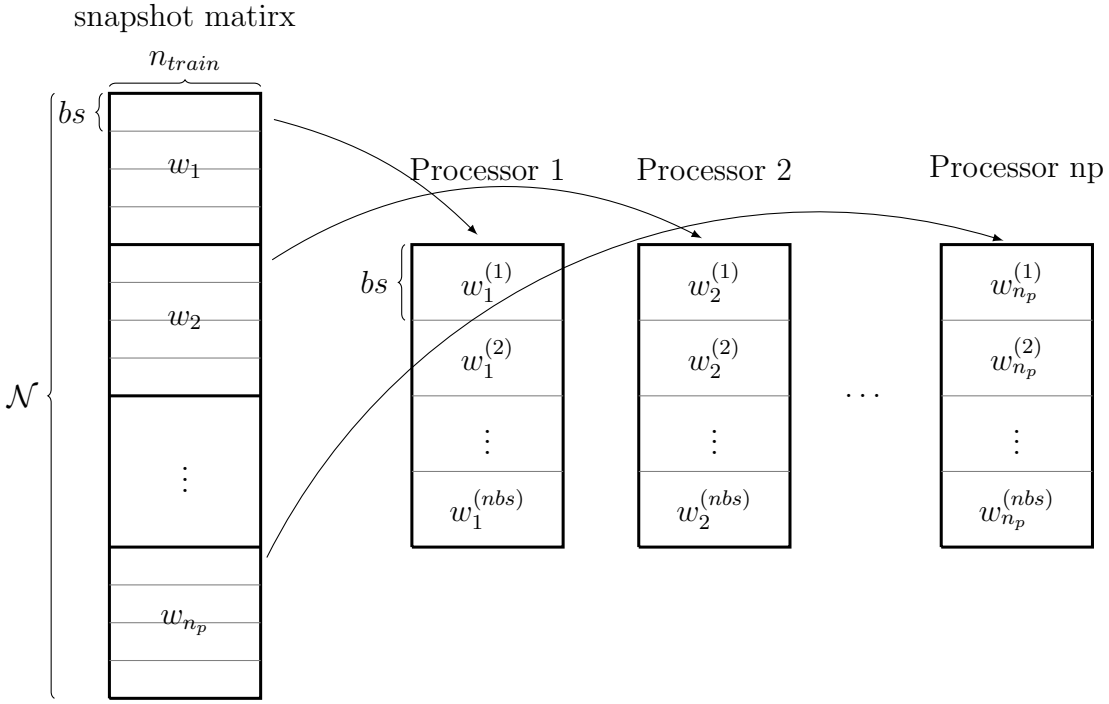

Figure 4.7: Partitioning of the snapshot matrix for non-blocking communication

The overlapping using non-blocking communication for the lines 2 and 3 of Algorithm 1 is presented in Algorithm 2.

We begin by sending the first sub-block of every block to the corresponding processor of the block. We do this using MPI_Iscatterv. It does not matter if the first sub-blocks are sent in a blocking or a non-blocking way, becaus we can't start computing until the first communication is finished. After the first sub-blocks are sent to the processors, the overlapping part starts. The next sub-blocks are sent in a non-blocking way (see Algorithm 2 line 3). Next we compute correlation matrix of the previous block. We are now communicating and sending at the same time. This is done for all sub-blocks. When all blocks are sent, the last block is computed. This last computation is not overlapped, because the communication is already finished (see Algorithm 2 line 8).

The MPI_Wait functions in line 4 and 7 of Algorithm 2 make sure that we do not access memory, if the communication is not yet finished.

Algorithm 2 Pseudocode overlapping communication

1: MPI_IScatterV ($w_{\cdot}^{(1)}$)
2: **for** $i = 2 : nbs$ **do**
3: MPI_IScatterV ($w_{\cdot}^{(i)}$)
4: MPI_Wait(i-1)
5: Eigen matrix-matrix product $c_n = c_n + {w_n^{(i-1)}}^T \cdot w_n^{(i-1)}$
6: **end for**
7: MPI_Wait(nbs)
8: Eigen matrix-matrix product $c_n = c_n + {w_n^{(nbs)}}^T \cdot w_n^{(nbs)}$

We can use the same overlapping principle from Algorithm 2 for the overlapping of lines 11 and 12 in Algorithm 1. But in this case it is much simpler. All the required data for the computation is already available. We can compute a block, start to send this block and continue with the same procedure for the next block. We do not have to wait for the communication to be finished while computing.

4.2.2 Hybrid approach

In Section 5.1 we will see that we spend a lot of time by waiting for the communication to be finished, especially increasing the number of processes because the many global communications can overload the distributed architecture. We want to reduce the waiting time by reducing the number of communications. Especially in a pure MPI implementation, every communication on the same note is a copy operation in the RAM.

The idea of the hybrid implementation is to use one MPI process for every processor in the cluster and make this MPI process able to use all cores of the processor using OpenMP. With this approach, we make use of the shared memory property of OpenMP and get rid of the MPI communications inside a single node. We only use MPI for the communication between the different processors where communication is necessary anyway.

For the hybrid implementation we use the code described in Algorithm 1, which brings the advantage that we do not need to change the code. We use the Eigen library for the matrix-matrix multiplications, and Eigen supports OpenMP. We have to make sure that we compile the program with OpenMP enabled.

5 Numerical Results

In this Section the methods and algorithms introduced in the previous chapters will be evaluated and compared by using benchmarks. We will see an example using the best algorithm in an RB software.

The numerical results and the implementation can be found on `GitLab`:
`https://gitlab.com/Flousen/parallelpod`

5.1 Benchmarks

The cluster

The Benchmarks are done on the Sissa HPC cluster called Ulysses. It was inaugurated on September 24 in 2014. The cluster can operate with 100 teraflops and provides 34 million computing hours a year [12]. For the benchmarks in this Section, we use four nodes. Every node has two processors with ten cores per processor and 160 gigabytes of ram.

Speedup

In the benchmarks, we compare the speedup of the different algorithms. The speedup is the factor of how much faster the algorithm is using multiple processors compared to the serial version. The optimum would be an algorithm where the factor is equal to the number of processors. In the following plots this optimum is marked with a blue line. We measure the speedup by dividing the time the algorithm took with n processors through time the algorithm took with serial version, since no concrete differences occur from the serial version.

Benchmarks

The computations are done with a random double matrix of the size $10^7 \times 300$ and using all the 300 POD modes which mean no truncation. 10^7 is a realistic value for $\mathcal{N}$ the degrees of freedom as well 300 is a realistic value for n_{train} the number of snapshots [14].

In the benchmarks we bind MPI processes to processors. So calling the program with two processes, we are using two cores, calling it with three processes we are using three cores and so on.

5.1.1 Comparison of the two algorithms

First, we compare the algorithm using the SVD described in Section 3.1 with the algorithm using the eigenvalue problem (EVP) described in Section 3.2.

In Figure 5.1, we compare the two algorithms in case the Snapshot matrix is not yet distributed to the processors. We simulate, the use case when all the snapshots are centralized into a single master process. We can see that the speedup of the EVP algorithm reaches a maximum of 7 at around 20 processors and is no longer growing. That means, at this point, adding more processors does not increase the speedup.

The SVD algorithm performs worse than the EVP algorithm. We see nearly no speedup. For 80 processors, the speedup is 1.94.

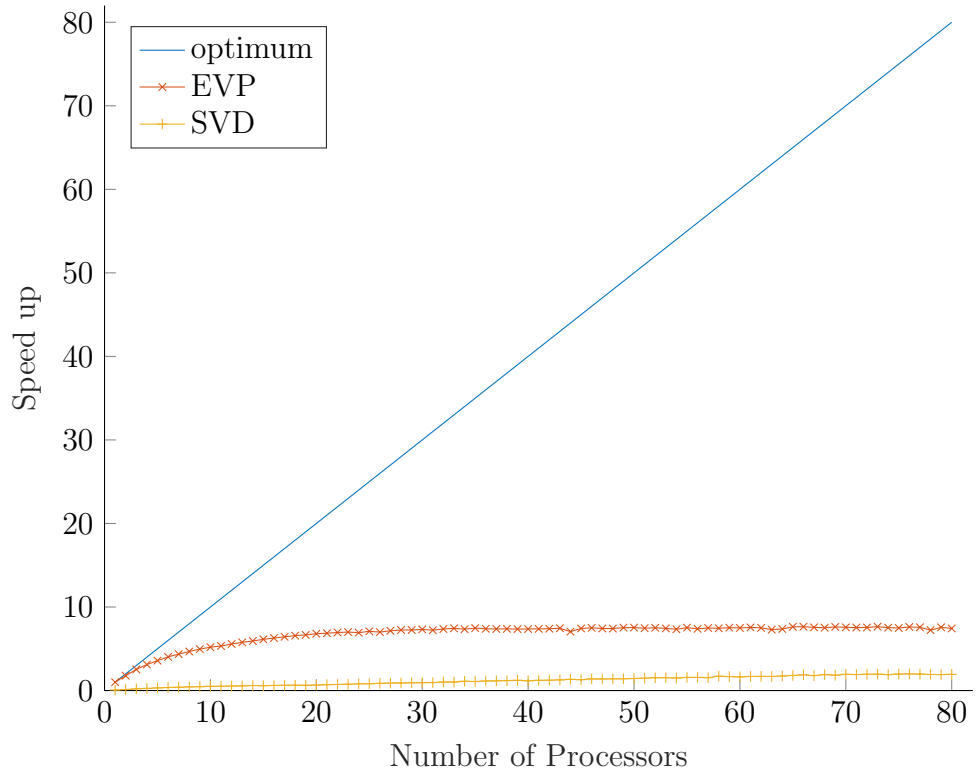

Figure 5.1: Speedup with communication

Figure 5.2 shows the case, that the snapshot matrix is already distributed to the processors, as usually happens in a decomposed domain.

In this case, we see both algorithms scale linearly. That means adding more processors increases the speedup for both processors. Both do not reach an optimal gradient of 1. The EVP algorithm has with 0.7 a higher growth rate than the SVD algorithm with 0.5

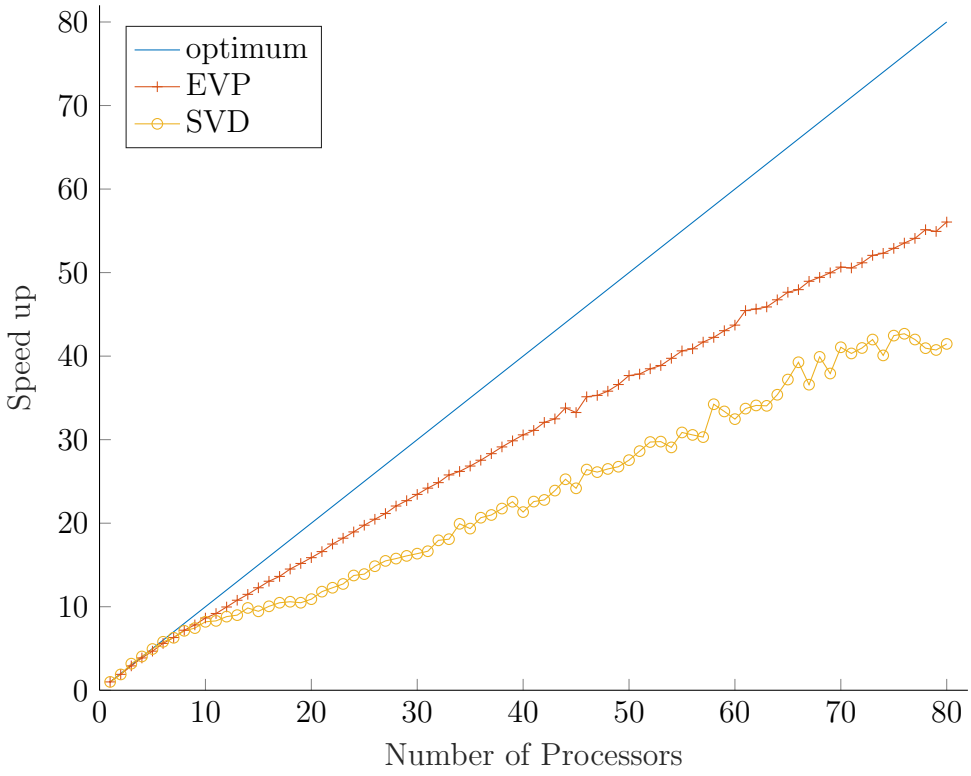

Figure 5.2: Speedup without initial distribution of the snapshots.

5.1.2 Communication block size

In case the snapshot matrix is not distributed to the processors, we reach a very bad speedup. We try to speed up the computation in this case by overlapping the communication and the computation of the correlation matrix, as described in Section 4.2.1.

Best block size

In Subsection 4.2.1 we defined a block as a consecutive sequence of rows. The minimum block size is one row.

We try different block sizes to find the optimal block size. In Table 5.1, one can see the overall time for different block sizes. We measured time for block sizes 10 to 500 in steps of 10 using 80 cores. We achieved the best overall time for the blocksizes 330.

bs	time	bs	time	bs	time	bs	time
10	9.16431	140	2.30277	270	2.22414	**400**	2.21817
20	4.42226	150	2.28341	280	2.28847	410	2.25889
30	3.41624	160	2.28626	290	2.22041	420	2.22602
40	2.96124	170	2.27469	**300**	2.25380	430	2.24157
50	2.71812	180	2.31770	310	2.28236	440	2.22704
60	2.68969	190	2.26263	320	2.25686	460	2.30125
70	2.58505	**200**	2.23699	**330**	2.18519	450	2.27380
90	2.49853	220	2.26803	340	2.27657	470	2.27369
80	2.51069	210	2.24825	350	2.26934	480	2.22723
100	2.43835	230	2.32931	360	2.22802	490	2.26090
110	2.40874	240	2.25289	370	2.21477	500	2.25510
120	2.44562	250	2.22183	380	2.19930		
130	2.38514	260	2.21478	390	2.21896		

Table 5.1: Overall time for different block sizes using 80 processors.

We selected a view block sizes from Table 5.1 and conducted benchmarks, to see how they perform in relation to speedup. The selected block sizes are marked red in Table 5.1. We selected 40 because it is the first block size where the time is in the magnitude of two seconds. We selected 330 because it is the fastest block size. We selected 100, 200, 300, and 400 to see how speedup is developing.

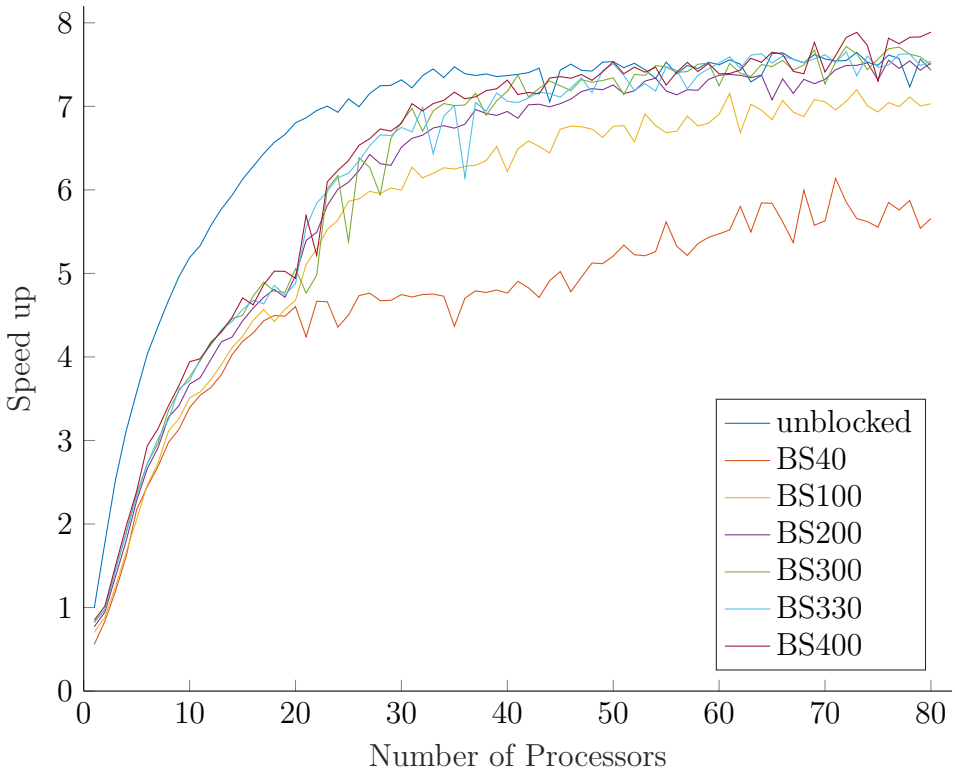

Figure 5.3: Speedup for different block sizes

Two things stand out in Figure 5.3:

1. All block sizes behave similarly up to 20 processors.
2. None of the block sizes is faster than the algorithm with blocking communication.

This shows that we don't gain any speed up with the approach using non blocking communication. Therefore, we investigate communications time respectively the waiting time.

Communication time

We measure the waiting time in Algorithm 2 (line 4). We measure it for a block size $bs = 300$. By measuring this we are measuring the waiting time we are not using for computations. We measured the waiting time for every block. In the appendix A.2 one can find the output of the measurement. We printed out the minimum and the maximum waiting time for every process. The mean minimum waiting time is 0.0062

seconds and the mean maximum waiting time is 0.0263 seconds. The maximum waiting time is in the magnitude of 10 longer than the minimum waiting time.

We explain the behavior by not knowing the structure of the cluster by assuming the true number of communications is congesting the cluster architecture. Even if the nodes are connected with a fast networking communications standard like InfiniBand, the communication will be slow if the data has traverse one or more nodes to get to its destination.

Up to 20 processes this has no effect because we are working on one single node. This is the reason why in Figure 5.3 all block sizes behave similar up to 20 processes.

For more than 20 processes we need communication with two or more nodes. We use the MPI_Igather and MPI_Iscatter functions; this leads to a slow down for all processes if the communication to one method is slow. That is because the process that scatters the data is waiting to make sure that we are not working on a communication buffer.

Possible solutions to this problem could be:

- Find a strategy for the specific arrangement of the nodes, involving the global communications.
- Make sure MPI_Igather and MPI_Iscatter are not blocking faster nodes. For example replace gather/scatter with a master worker pattern that uses a MPI_Isend and MPI_Irevive for every block sent to any node. This would prevent that slow communication to some nodes slows down other nodes.

For our propose the overlapping of the communication and computations brought no improvements.

5.1.3 Hybrid implementation

As we saw in the Section above, we spend a lot of time waiting for the communication to be finished. Using MPI, every communication between processes on the same node is copying data in the RAM. We can avoid these copying operations using a hybrid implementation. Hybrid means we are using MPI for the communication between the nodes and openMP for parallelization inside the node. Using OpenMP, we do not need copying operations inside the nodes because openMP is multithread with shared memory.

On the Ulysses cluster, every node has two processors with ten cores. We reach the best results using one MPI process per processor, that means two MPI processes per node. With this configuration every MPI process is running on its processor and can make use of all ten cores using OpenMP.

In Figure 5.4 you can see that the hybrid implementation is faster than the pure MPI implementation with a growing number of processors. The hybrid implementation does not reach a speedup maximum with 80 cores and still grows.

In Figure 5.4 you can see less dense measuring points for the speedup. This is because of the hybrid structure of the program. We are creating benchmarks by increasing the number of MPI processes. In the hybrid implementation, every MPI processes can make use of ten cores. This is also the reason why the measuring points are in steps of ten processors for the hybrid implementation.

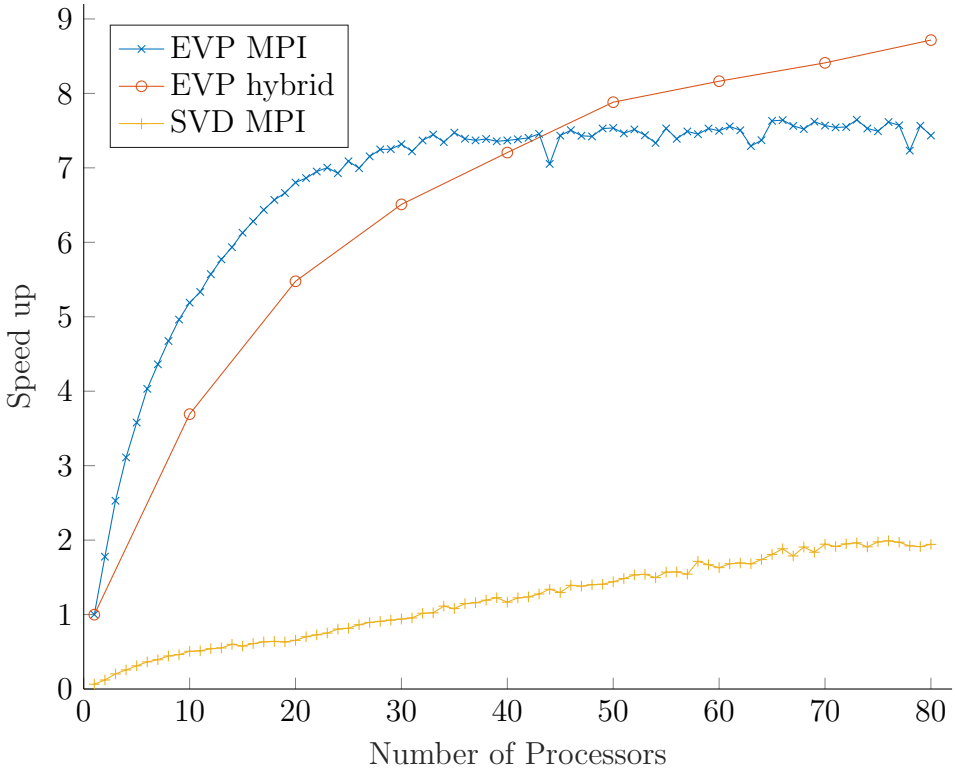

Figure 5.4: Speedup hybrid implementation MPI and OpenMP

5.2 Integration in EZyRB

`EZyRB` is an open-source python package for data-driven model order reduction. `EZyRB` provides three algorithms to compute the POD:

1. Singular value decomposition from NumPy
2. A randomized SVD, implemented using NumPy
3. Solving the eigenvalue problem of the correlation matrix, implemented using NumPy

NumPy provides fundamental linear algebra functions to python. The implemented POD algorithms are only using NumPy and cannot make use of a cluster with several nodes.

Python provides the possibility to write modules in C/C++. We wrote such a module and included the hybrid POD implementation. The source code of this module can be found in the appendix A.1 and on GitHub `https://github.com/Flousen/EZyRB`.

The MPI functionality is provided to `EZyRB` by the python package MPI4PY (MPI for Python) [3][4][5]. This package also provides c headerfiles that allow us to pass the MPI communicators from python to the C module. This makes the integration of the hybrid implementation into `EZyRB` very simple.

5.2.1 Problem Heat Conduction

In this Section, we describe a simple parametric PDE. The example is taken from [8, Section 2.3.1].

We consider a simple heat conduction problem with two parameters on a two-dimensional domain. The domain is defined as a rectangle. In the middle of the rectangle, there is a circle. This circle has a different conductivity than the rest of the rectangle. We call the circle Ω_1 and the rest of the rectangle Ω_2 (see Figure 5.5). The conductivity of Ω_1 is defined by the first parameter $\kappa = \mu_0$. The conductivity of Ω_2 is set to one $\kappa = 1$. The boundary of the domain $\partial\Omega$ is split in three parts. The two side edges of the rectangle Γ_{side}, the top edge Γ_{top}, and the bottom edge Γ_{base}.

The scalar field variable $u(\mu)$ is the temperature that satisfies Poisson's equation in Ω. With the boundary conditions:

- Homogeneous Neumann conditions on the side boundaries Γ_{side}. With this condition we have no heat flux on the side edges. This modeling a insulation of the sides.
- Homogeneous Dirichlet condition on the top edge boundary Γ_{top}. This is modeling a constant temperature of zero on the edge.
- Parametrized Neumann conditions on the bottom edge boundary Γ_{base}. The heat flux over the bottom edge is given by the second parameter μ_1.

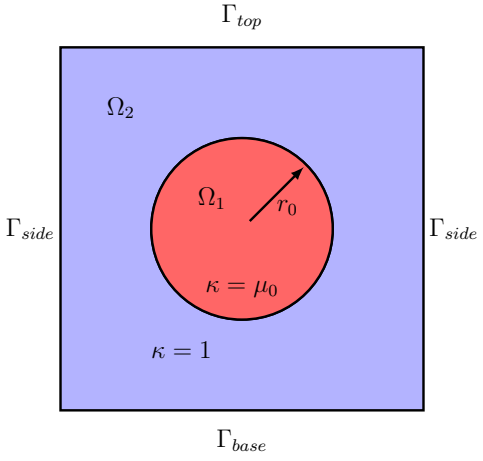

Figure 5.5: Sketch of the domain with different thermal conductivity coefficients.

There are two parameters in the problem. The conductivity μ_0 of Ω_1 and the temperature of the base edge μ_1.
The parameter space is defined by $\mathbb{P} = [\mu_0^{min}, \mu_0^{max}] \times [\mu_1^{min}, \mu_1^{max}]$. The parameter vector ist defined by $\mu = (\mu_0, \mu_1)^T \in \mathbb{P}$.

The strong formulation of the parameterized problem is given by:
for a given parameter $\mu \in \mathbb{P}$, find $u(x, \mu)$ such that

$$\begin{cases} -\nabla \cdot (\kappa(x,\mu)\nabla u(x,\mu)) = 0 & \text{in } \Omega \\ u(x,\mu) = 0 & \text{on } \Gamma_{top} \\ \kappa(\mu)\nabla u(x,\mu) \cdot \mathbf{n} = 0 & \text{on } \Gamma_{side} \\ \kappa(\mu)\nabla u(x,\mu) \cdot \mathbf{n} = \mu_1 & \text{on } \Gamma_{base} \end{cases} \tag{5.1}$$

where

- $\mathbf{n}$ denotes the outer normal to the boundaries Γ_{side} and Γ_{base},
- the conductivity $\kappa(x, \mu)$ is defined as follows:

$$\kappa(x,\mu) = \begin{cases} \mu_0, & x \in \Omega_1 \\ 1, & x \in \Omega_2 \end{cases} \tag{5.2}$$

The functional space is defined as $\mathbb{X} = \left\{ v \in H^1(\Omega) : v|_{\Gamma_{top}} = 0 \right\}$. The weak formulation is obtained by multiplying a test function $v \in \mathbb{X}$ on the function and then integrate over the domain Ω

$$\begin{aligned} -\nabla \cdot (\kappa(x,\mu)\nabla u(x,\mu))v(x) &= 0 \\ -\int_\Omega \nabla \cdot (\kappa(x,\mu)\nabla u(x,\mu))v(x)d\boldsymbol{x} &= 0 \end{aligned}$$

We integrate by part

$$\begin{aligned} &\int_{\Gamma_{top}} \kappa(x,\mu)\nabla u(x,\mu)v(x)d\boldsymbol{x} + \int_{\Gamma_{side}} \kappa(x,\mu)\nabla u(x,\mu)v(x)d\boldsymbol{x} \\ &+ \int_{\Gamma_{base}} \kappa(x,\mu)\nabla u(x,\mu)v(x)d\boldsymbol{x} + \int_\Omega \kappa(x,\mu)\nabla u(x,\mu)\nabla v(x)d\boldsymbol{x} = 0. \end{aligned}$$

The boundary integral over Γ_{top} disappears because we choose the test functions $v \in \mathbb{X}$,

all functions in $\mathbb{X}$ are zero on $v|_{\Gamma_{top}} = 0$. The boundary integral over Γ_{side} disappears because of the Neumann boundery condition $\kappa(\mu)\nabla u(x,\mu) \cdot \mathbf{n} = 0$. The boundary integral over Γ_{base} can be written as $\mu_1 \int_{\Gamma_{base}} v\mathrm{d}s$ on the right hand side.

The corresponding weak formulation reads: for a given parameter $\mu \in \mathbb{P}$,

$$\text{find } u \in \mathbb{X}: \quad a(u, v; \mu) = f(v; \mu) \quad \forall v \in \mathbb{X}$$

where

- the parametrized bilinear form $a(\cdot, \cdot; \mu)$ is defined by:

$$a(u, v; \mu) = \int_\Omega \kappa(\mu)\nabla u(x,\mu) \cdot \nabla v(x)\ d\boldsymbol{x},$$

- the affine decompositions are defined by:

$$\begin{aligned} a(u, v; \mu) &= \int_\Omega \kappa(\mu)\nabla u(x,\mu) \cdot \nabla v(x)\ d\boldsymbol{x} \\ &= \mu_0 \int_{\Omega_1} \nabla u(x,\mu) \cdot \nabla v(x)\ d\boldsymbol{x} + \int_{\Omega_2} \nabla u(x,\mu) \cdot \nabla v(x)\ d\boldsymbol{x}, \end{aligned}$$

- the parametrized linear form $f(v; \mu)$ is defined by:

$$f(v; \mu) = \mu_1 \int_{\Gamma_{base}} v\mathrm{d}s.$$

5.2.2 EZyRB benchmarks

To show the functionality of our algorithm we used it on the data from `tutorial 1` from `EZyRB` [6]. In the example the snapshots of the corresponding parameters and the mesh informations are included. For that tutorial eight snapshot of the example described above in Section 5.2.1 were created using `FEnics`. The parameter space $\mathbb{P}$ was sampled, in a range $\mathbb{P} = [0, 10] \times [-1, 1]$, with the following eight parameter combinations

$$\Xi_{train} = \left\{ \begin{pmatrix} 0.5 \\ -0.2 \end{pmatrix}, \begin{pmatrix} 8.6 \\ 0.1 \end{pmatrix}, \begin{pmatrix} 5.3 \\ 0.8 \end{pmatrix}, \begin{pmatrix} 9.4 \\ 0.1 \end{pmatrix}, \begin{pmatrix} 7.3 \\ -0.8 \end{pmatrix}, \begin{pmatrix} 0.2 \\ 0.8 \end{pmatrix}, \begin{pmatrix} 3.5 \\ -0.5 \end{pmatrix}, \begin{pmatrix} 0.3 \\ 0.6 \end{pmatrix} \right\}.$$

Figure 5.6 shows plots of the snapshots. The plots show the mesh information and the heat distribution.

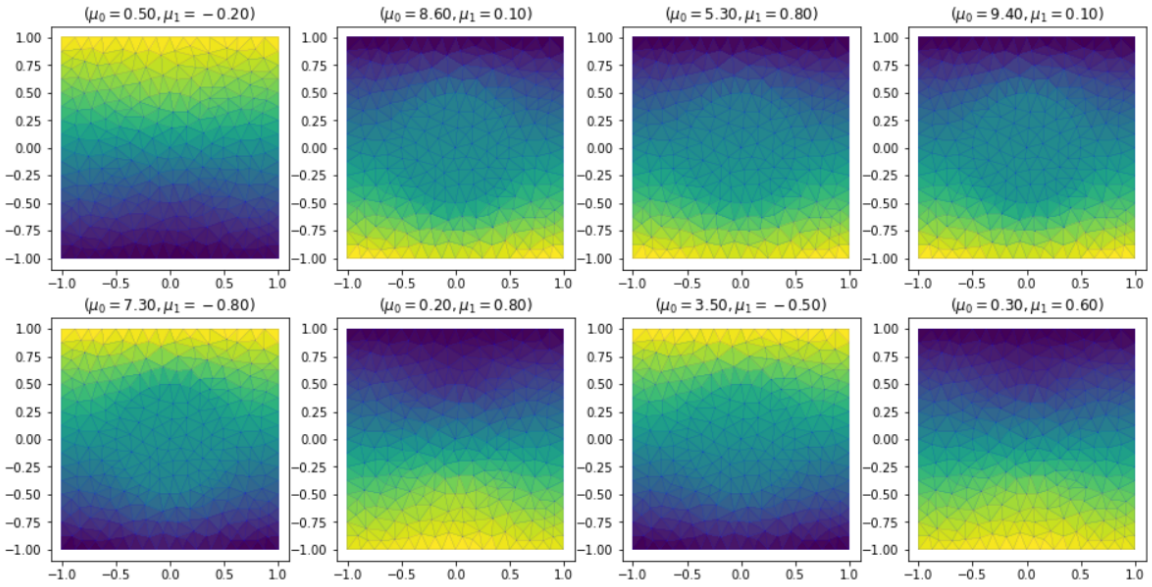

Figure 5.6: Heat distribution for the differen snapshots.

We created a reduced model from the snapshots using our algorithm. In Figure 5.7 the value of the singular values respectively the square root of the eigenvalues is plotted. We can see that the singular values shrink a lot between the first two modes. The following values are also shrinking but with a smaller magnitude than the first ones.

This behavior can be observed in practice. The first singular values have the biggest

magnitude and decrease fast. The modes with the singular values that fall no longer significantly can be truncated as discussed in Section 2.1.

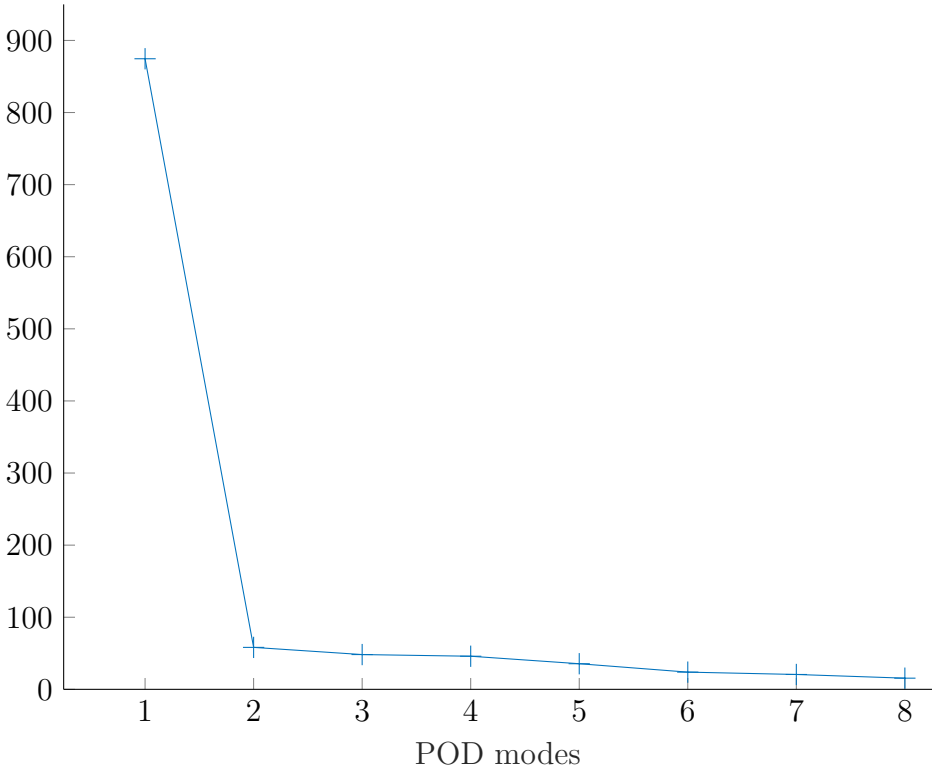

Figure 5.7: Singular values of the corresponding POD modes

For this simple example we didn't do the truncation. Figure 5.8 shows a reconstucted solution with the parameter $\bar{\mu} = (\bar{\mu}_1, \bar{\mu}_2)^T = (8, 1)^T$.

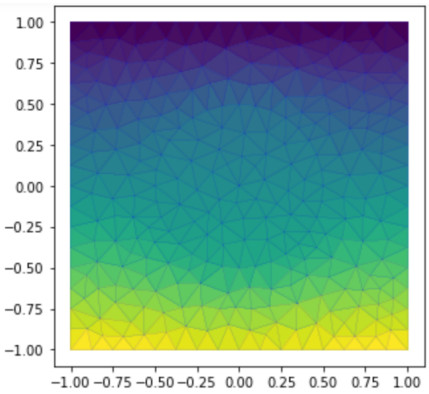

Figure 5.8: A solution computed with the reduced model created with `EZyRB`

Benchmark

With the data from tutorial 1 from `EZyRB` [6] we showed that the algorithm works properly, but it is not suitable to evaluate the performance.

To evaluate the performance we used a snapshot matrix with the same dimension as in Section 5.1 and measured the time the POD class needs to compute the POD modes. This benchmark shows the performance we loose by using the python layer.

Due to issues with the cluster we had to do this benchmark on a different cluster. The new cluster has processors with 16 cores. In Figure 5.9 we can see that the measuring points increase in steps of 16 processors and not in steps of 10 like the benchmarks in Section 5.1. We can still see that the implementation performed similar to the other benchmark in Figure 5.4 from Section 5.1.3.

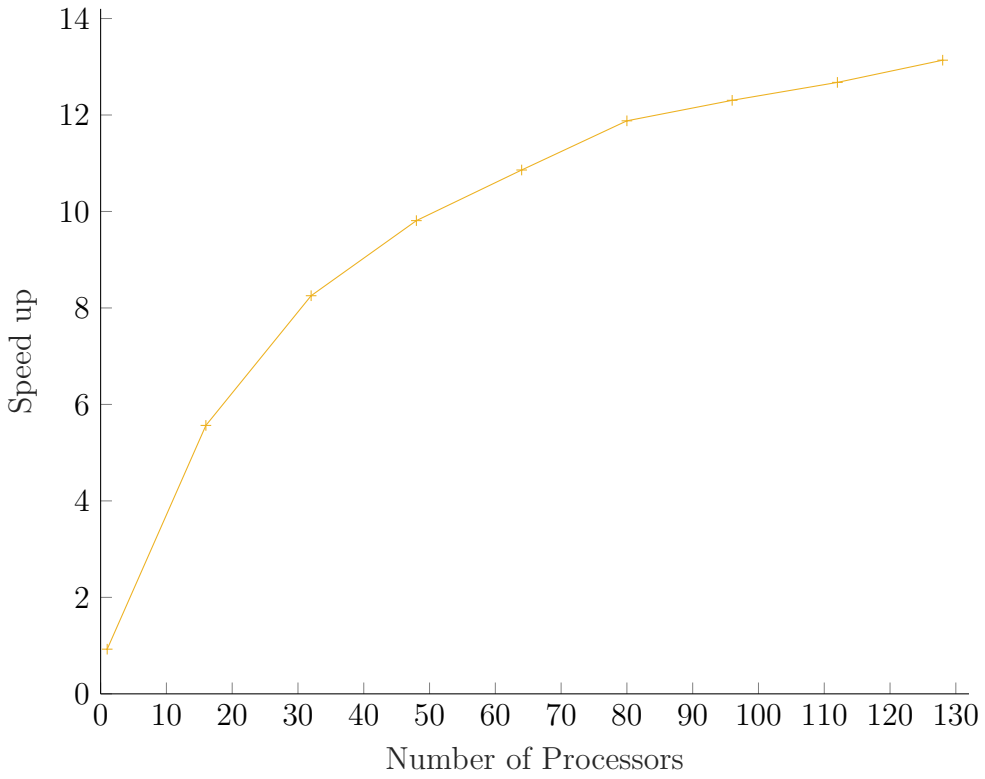

Figure 5.9: Benchmark integrated in the reduced order modeling framework `EZyRB`

5.3 Conclusion

We presented two algorithms to compute the POD in parallel. A parallel algorithm using the singular value decomposition (SVD) and a parallel algorithm solving the eigenvalue problem of the correlation matrix (EVP). Both algorithms performed good in case the snapshot matrix is already distributed to the different processors. We saw that the EVP algorithm performs best, due to its limited number of operations.

For the case that the snapshot matrix is not distributed to the processors the algorithm got improved. We proposed a hybrid implementation to overcome the number of communications.

Finally we integrated the hybrid implementation into the reduced order modeling framework `EZyRB`. We tested the new algorithm using `EZyRB` and also benchmarked the algorithm in the Python package. With the parallel algorithm `EZyRB` can now make use af high perfomance cluster for the POD modes extraction.

In future works one could try to speed up the computation of the correlation matrix for the case that the snapshot matrix is not distributed. Scattering the snapshot matrix to the processors is an expensive operation and slows down the overall POD computation a lot. Instead of scattering the snapshot matrix to the processors, one could try to use Graphics Processing Units (GPU) for the computation of the correlation matrix. GPUs are well suited these kine of operations.

A Appendix

A.1 POD module in C for Python

The MPI wrapper is taken from [4]:
https://bitbucket.org/mpi4py/mpi4py/src/
73129d2c792291a735c47bca51684ae1d524a4aa/demo/wrap-c/?at=master

We extended a numpy wrapper and included the POD algorithm.

```
1
2  #define MPICH_SKIP_MPICXX 1
3  #define OMPI_SKIP_MPICXX  1
4  #include <Eigen/Dense>
5  #include <mpi4py/mpi4py.h>
6  #include "arrayobject.h"
7
8  using namespace Eigen;
9  typedef Matrix<double, -1, -1, RowMajor> Mat;
10 /* - Hybrid POD algorithem ------------------------------------------------ */
11
12 static void
13 master(MPI_Comm comm, PyArrayObject *wmatrix, PyArrayObject *umatrix) {
14
15   if (comm == MPI_COMM_NULL) {
16     printf("You passed MPI_COMM_NULL !!!\n");
17     return;
18   }
19   // MPI: Who am i and how many
20   int size; MPI_Comm_size(comm, &size);
21   int rank; MPI_Comm_rank(comm, &rank);
22
23   // Pointer to allocated Snapsotmatrix
24   double * W = (double *)wmatrix->data;
25   // Pinter to allocated POD vectors
26   double * U = (double *)umatrix->data;
27
28   // get dimention from numpy object
29   int m,n;
30   m = PyArray_DIM(wmatrix, 0);
31   n = PyArray_DIM(wmatrix, 1);
32
33   // allocate correlation matrix
34   Mat CORR(n,n);
35
36   // brodcast dimentions to all processors
37   MPI_Bcast(&m, 1, MPI_INT, 0, comm);
38   MPI_Bcast(&n, 1, MPI_INT, 0, comm);
39
40   // calculate block sizes
41   int nofrows = m / size;
```

```
42    int remainder = m % size;
43    if (rank < remainder) ++nofrows;
44
45    int counts[size]; int displs[size];
46
47    int offset = 0;
48    for (int i = 0; i < size; ++i) {
49      displs[i] = offset;
50      counts[i] = m / size;
51      if (i < remainder) ++counts[i];
52
53      counts[i] *= n;
54      offset += counts[i];
55    }
56
57
58    // initiate Self Adjoint Eigen Problem Solver
59    Eigen::SelfAdjointEigenSolver<Mat> es;
60
61    // allocate local matrices
62    Mat w(nofrows, n);
63    Mat sv(n,n);
64
65    // Scatter snapshots
66    MPI_Scatterv( W , counts , displs , MPI_DOUBLE ,
67    w.data(), nofrows*n , MPI_DOUBLE ,
68    0 , comm  );
69
70    // compute local correaltion matrix
71    Mat corr(n,n);
72    corr.noalias() = w.transpose() * w;
73
74    // compute global correlation matrix
75    MPI_Reduce(corr.data(), CORR.data(), n*n, MPI_DOUBLE,
76    MPI_SUM, 0, comm);
77
78
79    // solve eigenlvlaue problem
80    es.compute(CORR);
81    // work = V * S^-1
82    sv = es.eigenvectors().rowwise().reverse().leftCols(n);
83    for(int i=0; i<sv.cols(); i++){
84      sv.col(i) *= 1.0/sqrt(es.eigenvalues().reverse().head(n)[i]);
85    }
86
87    // brodcast sv matrix
88    MPI_Bcast(sv.data(), n*n, MPI_DOUBLE, 0, comm);
89
90    // compute local pod vectors
91    Mat u;
92    u.noalias() = w * sv;
93
94    // gather pod vectors
95    MPI_Gatherv( u.data() , nofrows*n , MPI_DOUBLE ,
96    U, counts ,  displs , MPI_DOUBLE ,
97    0 , comm  );
98
99    return;
100 }
101
102 static void
103 worker(MPI_Comm comm) {
104
105   if (comm == MPI_COMM_NULL) {
106     printf("You passed MPI_COMM_NULL !!!\n");
107     return;
108   }
109
110   // MPI: Who am i and how many
111   int size; MPI_Comm_size(comm, &size);
112   int rank; MPI_Comm_rank(comm, &rank);
```

```
113
114   // brodcast dimentions to all processors
115   int m,n;
116   MPI_Bcast(&m, 1, MPI_INT, 0, comm);
117   MPI_Bcast(&n, 1, MPI_INT, 0, comm);
118
119   // calculate block sizes
120   int nofrows = m / size;
121   int remainder = m % size;
122   if (rank < remainder) ++nofrows;
123
124   int counts[size]; int displs[size];
125
126   // allocate local matrices
127   Mat w(nofrows, n);
128   Mat sv(n,n);
129
130   // recive local snapshots
131   MPI_Scatterv( NULL , counts , displs , MPI_DOUBLE ,
132   w.data(), nofrows*n , MPI_DOUBLE ,
133   0 , comm );
134
135   // compute local correaltion matrix
136   Mat corr(n,n);
137   corr.noalias() = w.transpose() * w;
138
139   // compute global correlation matrix
140   MPI_Reduce(corr.data(), NULL, n*n, MPI_DOUBLE,
141   MPI_SUM, 0, comm);
142
143   // recive sv
144   MPI_Bcast(sv.data(), n*n, MPI_DOUBLE, 0, comm);
145
146   // compute local correlation matrix
147   Mat u;
148   u.noalias() = w * sv;
149
150   //send local pod vecotrs
151   MPI_Gatherv( u.data() , nofrows*n , MPI_DOUBLE ,
152   NULL , counts , displs , MPI_DOUBLE ,
153   0 , comm );
154
155   return;
156 }
157
158 /* - python wrapper functions ---------------------------------------------- */
159
160 // Master Function called by python
161 static PyObject *
162 parapod_master(PyObject *self, PyObject *args)
163 {
164   PyObject *py_comm;
165   MPI_Comm *comm_p;
166   PyArrayObject *wmatrix;
167   PyArrayObject *umatrix;
168
169   if (!PyArg_ParseTuple(args, "OOO:master", &py_comm, &wmatrix, &umatrix)) return NULL;
170
171   comm_p = PyMPIComm_Get(py_comm);
172   if (comm_p == NULL) return NULL;
173
174   master(*comm_p, wmatrix, umatrix);
175
176   Py_INCREF(Py_None);
177   return Py_None;
178 }
179
180 // Worker Function called by python
181 static PyObject *
182 parapod_worker(PyObject *self, PyObject *args)
183 {
```

```
184   PyObject *py_comm;
185   MPI_Comm *comm_p;
186
187   if (!PyArg_ParseTuple(args, "O:worker", &py_comm)) return NULL;
188
189   comm_p = PyMPIComm_Get(py_comm);
190   if (comm_p == NULL) return NULL;
191
192   worker(*comm_p);
193
194   Py_INCREF(Py_None);
195   return Py_None;
196 }
197
198
199 static struct PyMethodDef parapod_methods[] = {
200   {"master",    (PyCFunction)parapod_master,    METH_VARARGS, NULL},
201   {"worker",    (PyCFunction)parapod_worker,    METH_VARARGS, NULL},
202   {NULL,        NULL,                           0,            NULL} /* sentinel */
203 };
204
205 #if PY_MAJOR_VERSION < 3
206 /* --- Python 2 --- */
207
208 PyMODINIT_FUNC initparapod_module(void)
209 {
210   PyObject *m = NULL;
211
212   /* Initialize mpi4py C-API */
213   //if (import_mpi4py() < 0) goto bad;
214   if (import_mpi4py() < 0) return;
215
216   /* Module initialization  */
217   m = Py_InitModule("parapod", parapod_methods);
218   if (m == NULL) goto bad;
219
220   return;
221
222   //bad:
223   //return;
224 }
225
226 #else
227 /* --- Python 3 --- */
228
229 static struct PyModuleDef parapod_module = {
230   PyModuleDef_HEAD_INIT,
231   "parapod", /* m_name */
232   NULL,          /* m_doc */
233   -1,            /* m_size */
234   parapod_methods    /* m_methods */,
235   NULL,          /* m_reload */
236   NULL,          /* m_traverse */
237   NULL,          /* m_clear */
238   NULL           /* m_free */
239 };
240
241 PyMODINIT_FUNC
242 PyInit_parapod(void)
243 {
244   PyObject *m = NULL;
245
246   /* Initialize mpi4py's C-API */
247   if (import_mpi4py() < 0) return NULL;
248
249   /* Module initialization  */
250   m = PyModule_Create(&parapod_module);
251   if (m == NULL) return NULL;
252
253   return m;
254 }
```

```
255
256 #endif
```

A.2 Waiting time BS 330

```
1  nodes=80 dofs=1000000 snapshots=300 bs=330
2  rank = 12  min = 0.0043131 max = 0.0249866
3  rank = 13  min = 0.0046517 max = 0.0242660
4  rank = 14  min = 0.0044078 max = 0.0240446
5  rank = 15  min = 0.0047344 max = 0.0241300
6  rank = 3   min = 0.0018230 max = 0.0247925
7  rank = 16  min = 0.0047779 max = 0.0245413
8  rank = 0   min = 0.0077286 max = 0.0135336
9  rank = 2   min = 0.0028240 max = 0.0250917
10 rank = 1   min = 0.0013872 max = 0.0214948
11 rank = 4   min = 0.0031843 max = 0.0247981
12 rank = 5   min = 0.0009631 max = 0.0252511
13 rank = 6   min = 0.0028348 max = 0.0271796
14 rank = 7   min = 0.0022160 max = 0.0245563
15 rank = 8   min = 0.0042824 max = 0.0255985
16 rank = 9   min = 0.0025902 max = 0.0258349
17 rank = 10  min = 0.0042816 max = 0.0229909
18 rank = 11  min = 0.0041316 max = 0.0235578
19 rank = 17  min = 0.0047368 max = 0.0243006
20 rank = 18  min = 0.0046265 max = 0.0246917
21 rank = 19  min = 0.0046437 max = 0.0271941
22 rank = 30  min = 0.0076763 max = 0.0314391
23 rank = 31  min = 0.0044388 max = 0.0293830
24 rank = 32  min = 0.0047194 max = 0.0298569
25 rank = 33  min = 0.0081659 max = 0.0307738
26 rank = 34  min = 0.0043481 max = 0.0297940
27 rank = 35  min = 0.0094767 max = 0.0307792
28 rank = 23  min = 0.0022570 max = 0.0297334
29 rank = 20  min = 0.0018981 max = 0.0291365
30 rank = 21  min = 0.0007308 max = 0.0275428
31 rank = 22  min = 0.0021548 max = 0.0276729
32 rank = 24  min = 0.0015231 max = 0.0311961
33 rank = 25  min = 0.0076888 max = 0.0296880
34 rank = 26  min = 0.0040453 max = 0.0311608
35 rank = 27  min = 0.0068116 max = 0.0312140
36 rank = 28  min = 0.0086282 max = 0.0316162
37 rank = 29  min = 0.0094516 max = 0.0294205
38 rank = 43  min = 0.0130634 max = 0.0232435
39 rank = 40  min = 0.0133841 max = 0.0241305
40 rank = 41  min = 0.0130425 max = 0.0254509
41 rank = 42  min = 0.0130218 max = 0.0256491
42 rank = 44  min = 0.0099885 max = 0.0290970
43 rank = 45  min = 0.0108420 max = 0.0290336
44 rank = 46  min = 0.0109732 max = 0.0285007
45 rank = 47  min = 0.0099189 max = 0.0284578
46 rank = 48  min = 0.0117451 max = 0.0279839
47 rank = 49  min = 0.0104234 max = 0.0286017
48 rank = 50  min = 0.0143602 max = 0.0229421
49 rank = 51  min = 0.0119498 max = 0.0275804
50 rank = 52  min = 0.0143473 max = 0.0228536
51 rank = 53  min = 0.0126253 max = 0.0269693
52 rank = 54  min = 0.0135484 max = 0.0262195
53 rank = 63  min = 0.0137158 max = 0.0233905
54 rank = 55  min = 0.0146116 max = 0.0235405
55 rank = 60  min = 0.0136492 max = 0.0233761
56 rank = 58  min = 0.0133206 max = 0.0265307
57 rank = 56  min = 0.0135418 max = 0.0278898
58 rank = 59  min = 0.0136631 max = 0.0282376
59 rank = 57  min = 0.0135731 max = 0.0291565
60 rank = 61  min = 0.0136313 max = 0.0282906
```

```
rank = 62  min = 0.0134978 max = 0.0237248
rank = 64  min = 0.0143792 max = 0.0231585
rank = 65  min = 0.0138563 max = 0.0283055
rank = 66  min = 0.0148908 max = 0.0268243
rank = 67  min = 0.0145527 max = 0.0234873
rank = 68  min = 0.0142011 max = 0.0237200
rank = 69  min = 0.0144420 max = 0.0284821
rank = 70  min = 0.0144735 max = 0.0238491
rank = 71  min = 0.0141707 max = 0.0241045
rank = 72  min = 0.0143723 max = 0.0287977
rank = 73  min = 0.0142900 max = 0.0239163
rank = 74  min = 0.0145044 max = 0.0242005
rank = 75  min = 0.0143538 max = 0.0241516
rank = 79  min = 0.0136074 max = 0.0283309
rank = 76  min = 0.0133768 max = 0.0282639
rank = 77  min = 0.0136647 max = 0.0295025
rank = 78  min = 0.0135516 max = 0.0255812
rank = 36  min = 0.0127170 max = 0.0241173
rank = 37  min = 0.0110443 max = 0.0255328
rank = 38  min = 0.0128222 max = 0.0243029
rank = 39  min = 0.0128500 max = 0.0239090
```

Bibliography

[1] Christopher Beattie et al. "A Domain Decomposition Approach to POD". In: Jan. 2007, pp. 6750 –6756. DOI: 10.1109/CDC.2006.377642.

[2] Saifon Chaturantabut and Danny Sorensen. "Nonlinear Model Reduction via Discrete Empirical Interpolation". In: *SIAM J. Scientific Computing* 32 (Jan. 2010), pp. 2737–2764. DOI: 10.1137/090766498.

[3] Lisandro D. Dalcin et al. "Parallel distributed computing using Python". In: *Advances in Water Resources* 34.9 (2011). New Computational Methods and Software Tools, pp. 1124 –1139. ISSN: 0309-1708. DOI: https://doi.org/10.1016/j.advwatres.2011.04.013. URL: http://www.sciencedirect.com/science/article/pii/S0309170811000777.

[4] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. "MPI for Python". In: *Journal of Parallel and Distributed Computing* 65.9 (2005), pp. 1108 –1115. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2005.03.010. URL: http://www.sciencedirect.com/science/article/pii/S0743731505000560.

[5] Lisandro Dalcín et al. "MPI for Python: Performance improvements and MPI-2 extensions". In: *Journal of Parallel and Distributed Computing* 68.5 (2008), pp. 655 –662. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2007.09.005. URL: http://www.sciencedirect.com/science/article/pii/S0743731507001712.

[6] Nicola Demo, Marco Tezzele, and Gianluigi Rozza. "EZyRB: Easy Reduced Basis method". In: *The Journal of Open Source Software* 3.24 (2018), p. 661. DOI: https://doi.org/10.21105/joss.00661.

[7] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[8] Jan Hesthaven, Gianluigi Rozza, and Benjamin Stamm. *Certified Reduced Basis Methods for Parametrized Partial Differential Equations*. Jan. 2016. ISBN: 978-3-319-22470-1. DOI: 10.1007/978-3-319-22470-1.

[9] Philip Holmes, John L. Lumley, and Gal Berkooz. *Turbulence, Coherent Structures, Dynamical Systems and Symmetry*. Cambridge Monographs on Mechanics. Cambridge University Press, 1996. DOI: 10.1017/CBO9780511622700.

[10] Karl Kunisch and Stefan Volkwein. "Galerkin Proper Orthogonal Decomposition Methods for a General Equation in Fluid Dynamics". In: *SIAM J. Numerical Analysis* 40 (July 2002), pp. 492–515. DOI: 10.1137/S0036142900382612.

[11] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Ed. by Anders Logg, Kent-Andre Mardal, and Garth N. Wells. Springer, 2012. ISBN: 978-3-642-23098-1. DOI: 10.1007/978-3-642-23099-8.

[12] Davide Ludovisi. "The new supercomputer is called Ulysses". In: (2014). URL: https://www.sissa.it/announcement/new-supercomputer-called-ulysses.

[13] Alfio Quarteroni, Gianluigi Rozza, and Annalisa Quaini. "Reduced basis methods for optimal control of advection-diffusion problems". In: (2007).

[14] Maria Strazzullo et al. *Model Reduction For Parametrized Optimal Control Problems in Environmental Marine Sciences and Engineering*. 2017. arXiv: 1710.01640 [math.NA].

[15] Maria Strazzullo et al. "Model Reduction for Parametrized Optimal Control Problems in Environmental Marine Sciences and Engineering". In: *SIAM Journal on Scientific Computing* 40.4 (2018), B1055–B1079. DOI: 10.1137/17M1150591.

[16] Marco Tezzele et al. *An integrated data-driven computational pipeline with model order reduction for industrial and applied mathematics*. 2018. eprint: arXiv:1810.12364.

[17] Stefan Volkwein. "Optimal Control of a Phase-Field Model Using Proper Orthogonal Decomposition". In: *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik* 81 (Feb. 2001), pp. 83 –97. DOI: 10.1002/1521-4001(200102)81:2<83::AID-ZAMM83>3.0.CO;2-R.

[18] H.G. Weller et al. "A Tensorial Approach to Computational Continuum Mechanics Using Object Orientated Techniques". In: *Computers in Physics* 12 (Nov. 1998), pp. 620–631. DOI: 10.1063/1.168744.

Name: Florian Krötz Matrikelnummer: 884948

Declaration

I declare that I have written the work independently and have not used any other sources and aids than those cited.

Ulm, June 19, 2020, 

Florian Krötz