High Performance Computing for the Discontinuous Galerkin methods

A thesis submitted for the degree of Doctor of Philosophy

Author:
Farukh Mukhamedov

Supervisor:
Matthias Maischak

Department of Mathematics
Brunel University London

2018

Declaration of Autorship

I undersigned, Farukh Mukhamedov, declare that this thesis titled

High Performance Computing for the Discontinuous Galerkin methods,

and the work presented in it is my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at the Brunel University London.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ______________________________

Date: ______________________________

ABSTRACT

Discontinuous Galerkin methods form a class of numerical methods to find a solution of partial differential equations by combining features of finite element and finite volume methods. Methods are defined using a weak form of a particular model problem, allowing for discontinuities in the discrete trial and test spaces. Using a discontinuous discrete space mesh provides proper flexibility and a compact discretisation pattern, allowing a multi-domain and multiphysics simulation.

Discontinuous Galerkin methods with a higher approximation polynomial order, the so-called p-version, performs better in terms of convergence rate, compared with the low order h-version with smaller element sizes and bigger mesh. However, the condition number of the Galerkin system grows subsequently. This causes surge in the amount of required storage, computational complexity and in the time required for computation. We use the following three approaches to keep the advantages and eliminate the disadvantages.

The first approach will be a specific choice of basis functions which we call **C1** polynomials. These ensure that the majority of integrals over the edge of the mesh elements disappears. This reduces the total number of non-zero elements in the resulting system. This decreases the computational complexity without loss in precision. This approach does not affect the number of iterations required by chosen Conjugate Gradients method when compared to the other choice of basis functions. It actually decreases the total number of algebraic operations performed.

The second approach is the introduction of suitable preconditioners. In our case, the Additive two-layer Schwarz method, developed in [4], for the iterative Conjugate Gradients method is considered. This directly affects the spectral condition number of the system matrix and decreases the number of iterations required for the computation. This approach, however, increases the total number of algebraic operations and might require more operational time.

To tackle the rise in the number of algebraic operations, we introduced a modified Additive two-layer non-overlapping Schwarz method with a Multigrid process. This using a fixed low-order approximation polynomial degree on a coarse grid. We show that this approach is spectrally equivalent to the first preconditioner, and requires less time for computation.

The third approach is a development of an efficient mathematical framework for distributed data structure. This allows a high performance, massively parallel, implementation of the discontinuous Galerkin method. We demonstrate that it is possible to exploit properties of the system matrix and **C1** polynomials as basis functions to optimize the parallel structures. The previously mentioned parallel data structure allows us to parallelize at the same time both the matrix-vector multiplication routines for the Conjugate Gradients method, as well as the preconditioner routines on the solver level. This minimizes the transfer ratio amongst the distributed system. Finally, we combined all three approaches and created a framework, which allowed us to successfully implement all of the above.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Solutions of partial differential equations (PDEs) are required in a wide area of applications from engineering to finance. Analytical methods of solutions for PDEs are often complicated, complex to implement. Exact solutions are not always available for testing purposes, and moreover, sometimes exact solutions are not required to study real-world problems. Numerical methods provide a way to find approximate solutions of required accuracy of the PDEs. Numerical methods are approximate in the sense that solutions of PDEs depend on the size of the discretization of the domain coordinates in continuous space setting, although the numerical solutions depend on a finite number of degrees of freedom.

Three well-known classes of the finite discretization methods are finite difference methods (FDM), finite volume methods (FVM) and finite element methods (FEM). FVM and FEM are especially well suited for computations in complicated domains consisting of irregular or unstructured meshes. A mesh divides the domain of validity into a finite number of elements. On each element, the discretization aims to reduce each variable of the PDE such that it depends only on a finite number of degrees of freedom. In many cases FEMs might appear to be more difficult to understand, however they offer more flexibility and superior accuracy in domains with complex boundaries and boundary conditions. Furthermore, the mathematical theory of FEMs is well-developed.

The most widely used cost-effective implementation of FEM, in terms of machine computation and high performance computing, is the (space) discontinuous Galerkin (dG) FEM. In this method a piecewise linear discretization needs to be found on each element and the limit or trace values approaching the nodes (or faces) from the element left or right of a node (or face) are not assumed to be continuous. Hence this discretization is continuous in each element, but discontinuous across elements. While more degrees of freedom are required in the discontinuous discretization, it (generally) offers more flexibility and more accuracy.

Both continuous Galerkin (cG) and dG FEMs have been well studied, and according to the **MathSciNet** there has been approximately 4500 and 2700 works, respectively, published in the last fifty years. Most of the work on a dG FEM, around 1200 of them are for so-called the h method, 300 so-called the p method, and "just" under a hundred for the *hp* dG FEM. The high performance and parallelisation techniques for *hp* dG FEM are studied in only ten projects, with only four of those for higher polynomial degree dG FEM. In [11],[43],[36],[47] and [8] authors use conventional parallelization approach applied on a linear Galerkin system, without accounting for discontinuity, choice of basis functions or actual analysis of the computational implementation. Additionally in those works, authors don't give any analysis on higher order dG FEMs, concentrating mainly on mesh structure and size analysis.

The main aim of this research is analysing the algorithmic implications of using the dG FEMs for higher order polynomials on a distributed memory machine with a relatively high communication cost. In this work, we are going to analyse the computational complexity of the existing method of Conjugate Gradients applied to a linear problem. We also analyse the two-level additive Schwarz method and show that the Multigrid method can be employed as the coarse level approximate local solver for successful preconditioning. We consider and present a special choice of polynomials, which, when used as the local basis functions, allows to successfully reduce computational complexity for higher polynomial degree dG FEM, without impacting convergence.

This work is organised in the following order. In the first chapter, Section **1.2** gives the definition of the Galerkin FEMs. In Section **1.3** we introduce the model problem for the Poison Equation, along with all the necessary mathematical tools and notation for the Symmetric Interior Penalty Galerkin (SIPG) and Local dG (LDG) FEMs. Additionally we are going to show how the LDG FEM can be reduced to any of the currently developed symmetric dG FEMs.

In the next chapter, Chapter **2**, we analyse the SIPG and LDG, provide the associated variational formulation, and bilinear and penalty forms which arise from the formulation. In Section **2.2** we introduce a basis for the chosen finite dimensional Sobolev spaces. We analyse the choice of the Antiderivative of Legendre polynomials (further **ADLP**) and C1 (b)-spline polynomials (further **C1**) as the global and local basis functions. We provide analysis of the global and local basis functions for different spatial dimensions, and highlight the properties of the chosen basis functions, and discuss the impact of such a choice on the overall computational complexity. In Section **2.3** we show the matrix setup and matrix formulation of the model problem. We are performing the complexity analysis, for the SIPG and LDG methods, and show how different approach to implementation can heavily impact the computational complexity of both the system matrix itself, and the solver. We analyse the matrix setup for integrals, which arise from the model problem's primal formulation, such as the volume integral, edge restricted integral and the computation of the integrals for lifting operators. In Section **2.4** we show the computation of the right hand side of the model problem, and give an estimate of the computational complexity. Section **2.5** is dedicated to the global matrix formulation. Error estimates in approximated solutions, obtained from different solvers, are given in Section **2.6**. Section **2.7** describes the Lanczos process. In this work we use fact that Lanczos process is built-in in the chosen solver. This allows computation of the spectral condition number for larger problems.

Chapter **3** is dedicated to the in-depth analysis of the iterative solvers, that solve the system of equations, given by the SIPG and LDG FEMs. In Section **3.1** we analyse the

Conjugate Gradient method as a solver, as well as analyse the complexity and the implications of the parallel implementation. We provide the parallelization techniques, best fit for higher order polynomial degree basis functions, and explain how the distribution of the data for computation is stored. We also analyse the effect of the basis function choice on the most cost-demanding operation in CG method - matrix-vector multiplication. We also provide an operation count analysis for the cases with Antiderivative of Legendre polynomial and C1 (b)-spline basis functions. Additionally we give the iteration number estimate in terms of the size of the chosen discretisation mesh and polynomial degree for the different choice of basis functions. Preconditioning of the CG method is briefly shown in the same section. In Section **3.2** we apply the Additive two-level non-overlapping Schwarz method as a preconditioner for the Conjugate Gradient method. In that section we show the complexity of the method, implications on a data-structure for parallel implementation and the specific ways of avoiding the surge in complexity arising from the preconditioner's use of the inverse of the system matrix. Further, in that section we describe in detail how the parallel implementation is constructed. We also discuss, how the parallel split can be optimised with proper analysis of the mesh elements, instead of a direct split of the system matrix. In Section **3.3** we describe the Multigrid method, and show that it can be used as the coarse level solver for the Additive two-level non-overlapping Schwarz method. Later in section **3.4** we analyse the abstract convergence of the Multigrid method and compare it with direct inverse solver for Additive Schwarz method. We will also prove the convergence of the developed method. In Section **3.5** we analyse the total complexity, and give a time estimate for the methods from the first to the last step.

Chapter **4** is dedicated to the numerical results for serial and parallel executions of the implementation. In Sections **4.1**,**4.2** we define test problems in two and three spatial dimensions. Parallel solvers are described in section **4.3**. To set up a common ground for repeatability of the problems, and to be able to compare them with one another, we describe the hardware and software specification in Section **4.4**. This allows to scale the results obtained on machines with different hardware and software specifications. To validate the results of the test problems, in section **4.5** we present error estimation techniques, as well as the error in the approximated solution. Condition number estimates are presented in the same section. We also analyse the speed-up effect in Section **4.6**, gained by the use of a specific choice of basis functions. We show, that C1-basis functions provide significant reduction of time spent on solving the test problems, while keeping the error and iteration number at the same order as for the ADLP basis functions. Additionally we present a comparison of the different solvers described in this work. This is done in Section **4.7**. In Section **4.8** we present and discuss the effect of parallelization for 2d and 3d test problems.

Chapter **5** completes this work with a conclusion. We provide a detailed discussion of the results of this work. We also suggest ideas for further research in that chapter.

1.2 Definition of the problem

Continuous and discontinuous Galerkin finite element discretization usually contains the following steps [12]:

I. *Derive the weak formulation:* Each side of the partial differential equation is multiplied by its own arbitrary test function, integrated over the domain of validity entirely or as a sum of integrals over all elements, and integrated by parts.

II. *Form the discretized weak formulation system:* The variables are expanded in the domain or in each element in a series in terms of a finite number of basis functions. Each basis function has compact support over neighbouring elements (for continuous finite elements) or within each element (for discontinuous finite elements). These expansions are then substituted into the weak formulation, and a test function is chosen alternately to coincide with a basis function. The resulting system is a linear or non-linear algebraic system.

III. *Evaluation of integrals in a local coordinate system:* A local or reference coordinate system is used to evaluate the integrals. Global matrices and vectors are assembled in the assembly routine.

IV. *Solve the algebraic system:* The resulting algebraic system is solved generally iteratively. In the case of non-linear algebraic system, such an iterative solution method is chosen, which essentially solves a linear system at each iteration step. For the linear algebraic system, which we are going to concentrate on in this work, a broad development was accomplished in recent decades to create methods that compute effective and accurate solutions.

Compact support means that the test functions only take non-zero values over one or a few neighbouring elements. In the continuous setting, the basis functions are taken to be zero at the edge of their domain of influence, while in the discontinuous case the basis function is (generally) non-zero within the element boundary and zero elsewhere.

In this work, we are going to present a framework, required to approximate the solution of linear algebraic systems, defined on a distributed memory machine with communication costs, dominating the execution time.

Finally, the careful definition of function spaces for the test and basis functions is common practice in finite element methods. This is often perceived to be very complicated, especially in the case of higher polynomial degrees in each variable on each element.

1.3 The Discontinuous Galerkin Finite Elements method

In the last ten years there has been a vigorous development in the study of the discontinuous Galerkin methods, and solvers for both linear and non-linear problems. Nine different discontinuous Galerkin methods were introduced and examined from the invention of the method. We start by introducing the discontinuous Galerkin method. For the sake of simplicity we choose the Poisson Equation. We note that the Poisson equation can be extended for different real life problems by introducing different non-zero factors. The model problem is then given by:

$$\begin{aligned} -\Delta u &= f \quad \text{in} \quad \Omega \\ u &= 0 \quad \text{on} \quad \partial\Omega, \end{aligned} \tag{1.1}$$

where f is a given function in suitable space $L^2(\Omega)$. Here we need to define our domain. Let $\Omega \subset \mathbb{R}^d, \quad d \geq 2$ be a polygonal domain, in some dimension d. Ω is an open bounded connected subset, such that $\bar{\Omega}$ is the union of a finite number of polyhedra. We also denote the boundary of the domain Ω by $\partial\Omega$.

We rewrite our problem as a first-order problem. This allows us to introduce the flux and average formulations. Additionally, it clears out the impact which choice of a domain partitioning and discontinuity have on the Galerkin method. The first-order formulation allows reduction of the problem to all existing Discontinuous Galerkin Methods [7].

$$\begin{aligned} \varrho &= \nabla u \quad \text{in} \quad \Omega \\ -\nabla \cdot \varrho &= f \quad \text{in} \quad \Omega \\ u &= 0 \quad \text{on} \quad \partial\Omega. \end{aligned} \tag{1.2}$$

We will now show how to derive the discontinuous Galerkin formulation of the (1.2). Only the discontinuous Galerkin method will later be investigated.

To be able to perform the next step of the Galerkin FEM, we need to multiply the resulting system (1.2) with the test function in the suitable spaces and integrate the result. To be able to do that, we first need to consider a finite decomposition

$$\bar{\Omega} = \bigcup_{K_i \in \mathscr{T}_h} K_i, \tag{1.3}$$

where:

1. each K is a polyhedron with non-empty interior:$\overset{o}{K} \neq \emptyset$
2. $\overset{o}{K}_1 \cap \overset{o}{K}_2 = \emptyset$ for each distinct $K_1, K_2 \in \mathscr{T}_h$
3. if $e = K_1 \cap K_2 \neq \emptyset$ (K_1 and K_2 are distinct elements of $\mathscr{T}_h$) then e is a common face, side, edge, or vertex of K_1 and K_2
4. $\operatorname{diam}(K) \lesssim h$ for each $K \in \mathscr{T}_h$.

Here we use the notation $x \lesssim y$ to state that there exists a generic constant $C > 0$, so that $x \leq Cy$. We note, that C is not the same for different occurrences. The nature of the constants in use is mentioned prior to occurrence.

For simplicity we use an arbitrary suitable invertible affine mapping T_K, such that each element K of $\mathscr{T}_h$ can be obtained from reference polyhedron $\hat{K}$ (or K^{ref}), where $K = T_K(\hat{K})$. Next, we define the spaces $L^2(\Omega)$ and $H^1(\Omega)$ as suitable infinite-dimensional spaces in Ω. To define integration on the edges (faces) of the elements, we reserve Γ as a so-called "skeleton" of the domain Ω - set of all edges e (faces in higher dimensions) on elements K of the domain decomposition $\mathscr{T}_h$, and Γ^0 as an "interior skeleton" without edges on $\partial\Omega$ as $\Gamma^0 := \Gamma \backslash \partial\Omega$. In short Γ is the set of **all** edges, Γ^0 is the set of only **internal** edges.

Additionally the employed common space decomposition techniques are triangulation and/or rectangulation (further, for both - **partitioning**)

$$\mathcal{T}_h = \{K\} \tag{1.4}$$

of the domain Ω, where we assumed that elements K are to be shape-regular without extreme acute angles. For simplicity of the notation, we will not restrict ourselves to equilateral shapes of the elements, for two main cases:

d-simplex The reference polyhedron $\hat{K}$ (or K^{ref}) is the unit d-simplex, e.g. $\triangle$**ABC** with the coordinates $A(0;0), B(0;1), C(1;0)$, for $d = 2$; and the tetrahedron $ABCD$ with the coordinates $A(0;0;0)$, $B(0;1;0)$, $C(1;0;0)$, $D(0;0;1)$, for $d = 3$ etc.

d-cube The reference polyhedron $\hat{K}$ (or K^{ref}) is the d-cube $[-1,1]^d$.

A second aspect to be noted, is a determination of a finite-dimensional space V_{hp}. Such a necessity arises for a suitable approximation of the chosen spaces $L^2(\Omega)$ and $H^1(\Omega)$ in Ω.To be more precise, we use

$$\mathcal{P}_p, p \geq 0, \tag{1.5}$$

to denote the space of polynomials of degree at most p in the variables $x_1, \ldots, x_d$. We use

$$\mathcal{Q}_p, p \geq 0, \tag{1.6}$$

to denote the space of polynomials that are of degree at most p with respect to **each** variable $x_1, \ldots, x_d$.

To employ hierarchical polynomials as the basis functions, the following inclusion should hold [44]

$$\mathcal{P}_p \subset \mathcal{Q}_p \subset \mathcal{P}_{d \cdot p}$$

To handle the mixed problems, where we often have to deal with vector-valued functions, we introduce the space of vector polynomials:

$$\mathcal{D}_p := (\mathcal{P}_{p-1})^d \oplus x\mathcal{P}_{p-1}, \quad p \geq 1,$$

where $x \in \mathbb{R}^d$ is the independent variable of interest. The space of vector polynomials $\mathcal{D}_p$ was first introduced by Raviart and Thomas in [45] and normally denoted as RT_p. Here, the next inclusion holds [44]

$$(\mathcal{P}_{p-1})^d \subset \mathcal{D}_p \subset (\mathcal{P}_p)^d$$

Finally, we set the space of the rectangular space decomposition:

$$V_{hp} := \{v_h \in L^2(\Omega) \quad : v\,|_K \circ T_K \in \mathcal{Q}_p \quad \forall K \in \mathcal{T}_h\}. \tag{1.7}$$

We also introduce the space of triangular finite elements

$$V_{hp} := \{v_h \in L^2(\Omega) \quad : v\,|_{\overset{o}{K}} \in \mathcal{P}_p \quad \forall K \in \mathcal{T}_h\}, \tag{1.8}$$

for the sake of completeness, but we will not use it later.

For the case of vector-valued functions, we introduce the space which covers both decomposition cases for the chosen reference elements [7]:

$$W_{hp} := \{q_h \in [L^2(\Omega)]^d \quad : q_h \mid_K \in [\mathrm{RT}_p]^d \quad \forall K \in \mathscr{T}_h\}\}, \tag{1.9}$$

with an appropriate choice of the space of polynomial functions for the rectangular decomposition. Here we can immediately see the following relation:

$$\nabla_h V_{hp} \subset W_{hp} \quad W_{hp} = [V_{hp}]^d. \tag{1.10}$$

The term ∇_h in (1.10) is applied to all functions in V_{hp} and for the test functions $v \in V_{hp}$ and $q \in W_{hp}$ is defined to be such, that functions $\nabla_h v$ and $\nabla_h \cdot q$ are the functions whose restrictions to each element $K \in \mathscr{T}_h$ are equal to ∇v and $\nabla \cdot q$, respectively.

Now, we multiply by the test functions $v \in V_{hp}$ and $q \in W_{hp}$. Integrating over all elements $K \in \mathscr{T}_h$ of domain Ω partitioning, we arrive at a weak formulation of the problem:

$$\begin{aligned} \int_{K\in\mathscr{T}_h} \varrho \cdot q dx &= -\int_{K\in\mathscr{T}_h} u\nabla \cdot q dx + \int_{\partial K\in\mathscr{T}_h} u n_K \cdot q ds, \\ \int_{K\in\mathscr{T}_h} \varrho \cdot \nabla v dx &= \int_{K\in\mathscr{T}_h} f v dx + \int_{\partial K\in\mathscr{T}_h} \varrho \cdot n_K v ds \end{aligned}$$

where n_K is the outward normal unit vector to ∂K.

Now, the general formulation is:

Find $u_h \in V_{hp}$ and $\varrho_h \in W_{hp}$, such that, $\forall K \in \mathscr{T}_h$:

$$\begin{aligned} \int_K \varrho_h \cdot q dx &= -\int_K u_h \nabla \cdot q dx + \int_{\partial K} \hat{u}_K n_K \cdot q ds, \quad q \in W_{hp} \\ \int_K \varrho_h \cdot \nabla v dx &= \int_K f v dx + \int_{\partial K} \hat{\varrho}_K \cdot n_K v ds, \quad v \in V_{hp} \end{aligned} \tag{1.11}$$

where the numerical fluxes, $\hat{\varrho}_K$ is an approximation to $\varrho = \nabla u$ and $\hat{u}_K$ is an approximation to u on the boundary of K. Numerical flux is introduced by defining an appropriate functional setting. We use $H^l(\mathscr{T}_h)$ to denote the space of functions on Ω whose restriction to each element K belongs to the Sobolev space $H^l(K)$, i.e. $H^l(\mathscr{T}_h) = \{u \in L^2(\Omega) : u \mid_K \in H^l(K)\}\}$. Then, the finite-dimensional space V_{hp} is the subset of $H^l(\mathscr{T}_h)$, for any l, $l = 1/2$ for dG. And W_{hp} is the subset of $[H^l(\mathscr{T}_h)]^d$, for any l and $d \geq 2$. Traces of a function in $H^1(\mathscr{T}_h)$ belong to $T(\Gamma) := \prod_{K\in\mathscr{T}_h} L^2(\partial K)$, where

$$\Gamma = \bigcup_{K\in\mathscr{T}_h} \partial K \tag{1.12}$$

is the union of the edges of the elements K of $\mathscr{T}_h$. Functions in $T(\Gamma)$ are thus vector-valued on

$$\Gamma^0 := \Gamma \setminus \partial\Omega \tag{1.13}$$

and single-valued on $\partial\Omega$. The space $L^2(\Gamma)$ is then the subspace of $T(\Gamma)$ consisting of functions for which the values are the same on all internal edges. Then, numerical flux

$\hat{u} = (\hat{u}_K)_{K\in\mathcal{T}_h}$ is scalar valued and $\hat{\varrho} = (\hat{\varrho}_K)_{K\in\mathcal{T}_h}$ is vector valued. Both numerical fluxes are linear functions and according to Arnold et al. [7] only their normal components contribute to the DG method. In other words numerical flux $\hat{u}_K$ is a trace of a function u on element K.

We can now sum over all elements $K \in \mathcal{T}_h$ in (1.11):

$$\begin{aligned} \int_\Omega \varrho_h \cdot q dx &= -\int_\Omega u_h \nabla_h \cdot q dx + \sum_{K\in\mathcal{T}_h} \int_{\partial K} \hat{u}_K n_K \cdot q ds, \quad q \in W_{hp} \qquad (1.14) \\ \int_\Omega \varrho_h \cdot \nabla_h v dx &= \int_\Omega f v dx + \sum_{K\in\mathcal{T}_h} \int_{\partial K} \hat{\varrho}_K \cdot n_K v ds, \quad v \in V_{hp}. \end{aligned}$$

The aforementioned system (1.14) uses the explicit sum on the right hand side, which can be simplified with the use of the jump and average functions. As one of the common components of any DG method we define the jump and the average of the test functions. To denote them we first introduce the sets of interior and boundary faces respectively $\mathcal{F}_h^I$ and $\mathcal{F}_h^B$ [5] and set $\mathcal{F}_h = \mathcal{F}_h^I \bigcup \mathcal{F}_h^B$. We then have:

For a mixed formulation of a model problem, we have scalar and vector valued test functions $v \in T(\Gamma)$ and $q \in [T(\Gamma)]^d$. For the scalar functions we define the weighted average $\{v\}_\sigma$ (just $\{v\}$ for whenever $\sigma = 1/2$), and the jump $[\![v]\!]$ of v on Γ^0 across the edge (face) e shared by two elements K_1 and K_2 with unit normal vectors n_1 and n_2 ($n_1, n_2 \in \mathbb{R}^d$) on that edge pointing exterior to K_1 and K_2, respectively. Then for $v_i := v \mid_{\partial K_i}$

$$\begin{aligned} \{v\}_\sigma &= \sigma_1 v_1 + (1-\sigma_2) v_2, \quad \{\cdot\} : T(\Gamma) \to \mathbb{R} \\ [\![v]\!] &= v_1 n_1 + v_2 n_2, \quad [\![\cdot]\!] : T(\Gamma) \to \mathbb{R}^d, \end{aligned}$$

on $e \in \mathcal{F}_h^I$. And by applying the same scheme towards $q \in [T(\Gamma)]^d$

$$\begin{aligned} \{q\}_\sigma &= \sigma_1 q_1 + (1-\sigma_2) q_2, \quad \{\cdot\} : [T(\Gamma)]^d \to \mathbb{R}^d, \\ [\![q]\!] &= q_1 \cdot n_1 + q_2 \cdot n_2, \quad [\![\cdot]\!] : [T(\Gamma)]^d \to \mathbb{R}, \end{aligned}$$

on $e \in \mathcal{F}_h^I$. Note that on the boundary faces $e \in F_h^B$:

$$\{q\}_\sigma|_{\partial\Omega} = q, \quad \{\cdot\} : [T(\Gamma)]^d \to \mathbb{R}^d,$$

and

$$[\![v]\!]|_{\partial\Omega} = v \cdot n, \quad [\![\cdot]\!] : T(\Gamma) \to \mathbb{R}^d.$$

The mapping of the jump and average operator gives:

$$\begin{aligned} &\{\cdot\}_\sigma : T(\Gamma) \to L^2(\Gamma^0), \quad [\![\cdot]\!] : T(\Gamma) \to [L^2(\Gamma)]^d \\ &\{\cdot\}_\sigma : [T(\Gamma)]^d \to [L^2(\Gamma)]^d, \quad [\![\cdot]\!] : [T(\Gamma)]^d \to L^2(\Gamma^0) \end{aligned}$$

Now, as defined by Arnold et. al. [7], we introduce the identity:

$$\sum_{K\in\mathcal{T}_h} \int_{\partial K} v_K q_K ds = \int_\Gamma [\![v]\!] \cdot \{q\} ds + \int_{\Gamma^0} \{v\} [\![q]\!] ds \qquad (1.15)$$

Applying (1.15) to (1.14) yields:

$$\begin{aligned}\int_\Omega \varrho_h \cdot q dx &= -\int_\Omega u_h \nabla_h \cdot q dx + \int_\Gamma [\![\hat{u}]\!] \cdot \{q\} ds + \int_{\Gamma^0} \{\hat{u}\}[\![q]\!] ds, \quad q \in W_{hp} \qquad (1.16)\\ \int_\Omega \varrho_h \cdot \nabla_h v dx &- \int_\Gamma [\![\hat{\varrho}]\!] \cdot \{v\} ds - \int_{\Gamma^0} \{\hat{\varrho}\}[\![v]\!] ds = \int_\Omega f v dx + \sum_{K \in \mathscr{T}_h} \int_{\partial K} \hat{\varrho}_K \cdot n_K v ds, \quad v \in V_{hp},\end{aligned}$$

Now to write ϱ_h only in terms of u_h, we should use (1.15) and restrict the functions to their traces. We obtain, $\forall \tau \in [H^1(\mathscr{T}_h)]^2$ and $v \in H^1(\mathscr{T}_h)$, the integration by parts formula:

$$\int_\Omega \nabla_h \cdot \tau v dx = \int_\Omega \tau \cdot \nabla_h v dx - \int_\Gamma \{\tau\}[\![v]\!] ds - \int_{\Gamma^0} [\![\tau]\!]\{v\} ds \qquad (1.17)$$

Choosing such $v = u_h$ in (1.17), then substituting the resulting right hand side into (1.16), and, for every $\tau \in W_{hp}$:

$$\int_\Omega \varrho_h \cdot \tau dx = \int_\Omega \nabla_h u_h \cdot \tau dx + \int_\Gamma [\![\hat{u} - u_h]\!] \cdot \{\tau\} ds + \int_{\Gamma^0} \{\hat{u} - u_h\}[\![\tau]\!] ds \qquad (1.18)$$

We can now define the lifting operators, to simplify the numerical flux and jump and average notation. We are giving the definitions of lifting operators for the scalar- and vector valued test functions, $v \in T(\Gamma)$ and $q \in [T(\Gamma)]^d$ respectively. Recalling (1.10) we use the defition by [7] as:
the right lifting operator:

$$r : [L^2(\Gamma)]^d \to [W_{hp}]^d, \quad \int_\Omega r(q) \cdot \tau dx = -\int_\Gamma q \cdot \{\tau\} ds, \quad \forall \tau \in [W_{hp}]^d, \qquad (1.19)$$

the left lifting operator:

$$l : L^2(\Gamma^0) \to [W_{hp}]^d, \quad \int_\Omega l(v) \cdot \tau dx = -\int_{\Gamma^0} v \cdot [\![\tau]\!] ds, \quad \forall \tau \in [W_{hp}]^d. \qquad (1.20)$$

Noting, that $\nabla_h V_{hp} \subset W_{hp}$, we can substitute (1.19,1.20) into the (1.18):

$$\varrho_h = \varrho_h(u_h) := \nabla_h u_h - r([\![\hat{u}(u_h) - u_h]\!]) - l(\{\hat{u}(u_h) - u_h\}) \qquad (1.21)$$

Choosing $\tau = \nabla_h v$ in (1.18), we rewrite (1.16) as:

$$B_h(u_h, v) = \int_\Omega f v dx, \quad \forall v \in V_{hp}, \qquad (1.22)$$

where:

$$\begin{aligned}B_h(u_h, v) \; &:= \; \int_\Omega \nabla_h u_h \cdot \nabla_h v dx + \int_\Gamma [\![\hat{u} - u_h]\!] \cdot \{\nabla_h v\} - \{\hat{\varrho}\} \cdot [\![v]\!] ds\\ &+ \; \int_{\Gamma^0} \{\hat{u} - u_h\}[\![\nabla_h v]\!] - [\![\hat{\varrho}]\!]\{v\} ds. \qquad (1.23)\end{aligned}$$

For any functions $u_h, v \in H^2(\mathscr{T}_h)$, the setting above defines $B_h(u_h, v)$, with $\hat{u} = \hat{u}(u_h)$ and

$\hat{\varrho} = \hat{\varrho}(u_h, \varrho_h(u_h))$, where the mapping $u_h \mapsto \varrho_h(u_h)$ is given by (1.21). The form $B_h : H^2(\mathscr{T}_h) \times H^2(\mathscr{T}_h) \to \mathbb{R}$ is bilinear [7] and is symmetric for the numerical fluxes presented later in the table 1.2. We call (1.22) the primal formulation of the method. The bilinear form $B_h(\cdot,\cdot)$ the primal form.

If $(u_h, \varrho_h) \in V_{hp} \times W_{hp}$ solves (1.2), then u_h solves (1.22).

At this point, the defined primal formulation can be used to yield the Interior Penalty and Local Discontinuous Galerkin FEMs.

We start with the Interior Penalty method. The method was originally proposed as a primal formulation [6], with:

$$\begin{aligned} B_h(u_h, v) &= \int_\Omega \nabla_h u_h \cdot \nabla_h v dx - \int_\Gamma ([\![u_h]\!] \cdot \{\nabla_h v\} + \{\nabla_h u_h\} \cdot [\![v]\!]) ds \\ &+ \alpha_j(u_h, v), \end{aligned} \tag{1.24}$$

where

$$\alpha_j(u_h, v) = \int_\Gamma \alpha [\![u_h]\!] \cdot [\![v]\!] ds \tag{1.25}$$

is the interior penalty, stabilization term with the penalty weighting function $\alpha : \Gamma \to \mathbb{R}$, given by

$$\alpha = \gamma_e p_e^2 h_e^{-1}$$

on each $e \in \mathscr{F}_h$, with $\gamma_e > 1$. The numerical fluxes, are chosen as

$$\begin{aligned} \hat{u} &= \{u_h\} \quad \text{on} \quad \Gamma^0, \\ \hat{u} &= 0 \quad \text{on} \quad \partial\Omega, \\ \hat{\varrho} &= \{\{\nabla_h u_h\} - \alpha_j([\![u_h]\!]) \quad \text{on} \quad \Gamma,. \end{aligned} \tag{1.26}$$

With this choice for the numerical fluxes (1.21) yields:

$$\varrho_h = \nabla_h u_h + r([\![u_h]\!]) \tag{1.27}$$

and:

$$\begin{aligned} \int_\Gamma \{\hat{\varrho}\} \cdot [\![v]\!] ds &= \int_\Gamma \{\nabla_h u_h\} \cdot [\![v]\!] ds \\ &- \int_\Gamma \alpha_j([\![u_h]\!]) \cdot [\![v]\!] ds. \end{aligned} \tag{1.28}$$

And (1.24) follows by substituting (1.28) into (1.23).

Note that the alternative formulation can be derived through the use of the lifting operator, $r_e : [L^1(e)]^2 \to W_{hp}$, given by:

$$\int_\Omega r_e(\phi) \cdot \tau dx = -\int_e \phi \cdot \{\tau\} ds, \quad \forall \tau \in W_{hp}, \phi \in [L^1(e)]^d \tag{1.29}$$

And defining $\alpha_r(\phi) := -\eta_e \{r_e(\phi)\}$ on e. $r_e(\phi)$ vanishes outside of the union of one or two elements of the partition containing e as a common, and that $r(\phi) = \sum_{e \in \mathscr{F}_h} r_e(\phi)$, for all

$\phi \in [L^1(e)]^2$. Keeping the choice for the numerical flux $\hat{u}$ similar to the one in (1.26), but using the α_r instead of α_j in the $\hat{\varrho}$, yields the method of Bassi et al [9], with the primal:

$$\begin{aligned} B_h(u_h, v) &= \int_\Omega \nabla_h u_h \cdot \nabla_h v dx - \int_\Gamma ([\![u_h]\!] \cdot \{\nabla_h v\} + \{\nabla_h u_h\} \cdot [\![v]\!]) ds \\ &+ \alpha_r(u_h, v), \end{aligned} \tag{1.30}$$

with stabilization term α_r modified as:

$$\begin{aligned} \alpha_r(u_h, v) &= \int_\Gamma \alpha_r(u_h) \cdot [\![v]\!] ds \\ &= \sum_{e \in \mathscr{F}_h} \int_\Omega \eta_e r_e([\![u_h]\!]) \cdot r_e([\![v]\!]) ds \end{aligned} \tag{1.31}$$

At this point we should note, that both stabilization terms α_r and α_j are used in most contemporary discontinuous Galerkin FEMs.

Now we present the definition of the Local Discontinuous Galerkin FEM, first introduced in [21]. The choice of the numerical fluxes is given

$$\begin{aligned} \hat{u} &= \{u_h\} - \beta \cdot [\![u_h]\!] \quad \text{on} \quad \Gamma^0 \\ \hat{u} &= 0 \quad \text{on} \quad \partial\Omega \end{aligned} \tag{1.32}$$

and

$$\begin{aligned} \hat{\varrho} = \{\varrho_h\} + \beta [\![\varrho_h]\!] - \alpha_j([\![u_h]\!]) \quad \text{on} \quad \Gamma^0 \\ \hat{\varrho} = \{\varrho_h\} - \alpha_j([\![u_h]\!]) \quad \text{on} \quad \partial\Omega, \end{aligned}$$

where $\beta \in [L^2(\Gamma^0)]^d$ is a vector-valued function, constant on each edge(face). From the scalar flux choice (1.32) we have $\{\hat{u} - u_h\} = -\beta \cdot [\![u_h]\!]$ on Γ^0, and $[\![\hat{u} - u_h]\!] = -[\![u_h]\!]$ on Γ, so that (1.18) yields

$$\varrho_h = \nabla_h u_h + \tau, \tag{1.33}$$

with $\tau := r([\![u_h]\!]) + l(\beta \cdot [\![u_h]\!]) \in W_{hp}$. Then the vector flux choice (1.33) gives

$$\hat{\varrho} = \{\nabla_h u_h\} + \{\tau\} + \beta [\![\nabla_h u_h]\!] + \beta [\![\tau]\!] - \alpha_j([\![u_h]\!])$$

Using the right and left lifting operators (1.19,1.20), we get

$$\begin{aligned} \int_\Gamma \{\hat{\varrho}\} \cdot [\![v]\!] ds &= \int_\Gamma \{\nabla_h u_h\} \cdot [\![v]\!] ds + \int_{\Gamma^0} [\![\nabla_h u_h]\!] \beta \cdot [\![v]\!] ds \\ &- \int_\Omega [r([\![v]\!]) + (\beta \cdot [\![v]\!])] \cdot \tau dx - \alpha_j(u_h, v) \end{aligned}$$

Substituting into (1.23), we obtain the following bilinear form for the LDG method:

$$\begin{aligned} B_h(u_h, v) &= \int_\Omega \nabla u_h \cdot \nabla v dx - \int_\Gamma ([\![u_h]\!] \cdot \{\nabla v\} + \{\nabla u_h\} \cdot [\![v]\!]) ds \\ &+ \int_{\Gamma^0} (\beta \cdot [\![u_h]\!][\![\nabla v]\!] + [\![\nabla u_h]\!]\beta \cdot [\![v]\!]) ds \\ &+ \int_\Omega \Big[r([\![u_h]\!]) + l(\beta \cdot [\![u_h]\!]) \Big] \cdot \Big[r([\![v]\!]) + l(\beta \cdot [\![v]\!]) \Big] dx \\ &+ \alpha_j(u_h, v) \end{aligned}$$

We would use the Local discontinuous Galerkin method, as our main choice, as this method can be reduced to any other Discontinuous Galerkin FEM. Prior to demonstrating this we give the definitions of the used tools.

The inner product $(\cdot,\cdot) : V_{hp} \times V_{hp} \to \mathbb{R}$ is defined in the following way:

$$\begin{aligned} (w, v)_\Omega &:= \int_\Omega w \cdot v dx \\ (w, v) &\equiv (w, v)_\Gamma := \int_\Gamma w \cdot v ds \\ \langle w, v \rangle &\equiv (w, v)_{\Gamma^0} := \int_{\Gamma^0} w \cdot v ds \\ \forall w, v \in V_{hp} \end{aligned}$$

We define a bilinear form $a(\cdot,\cdot) : V_{hp} \times V_{hp} \to \mathbb{R}$:

$$a(w, v)_\Omega := \int_\Omega \nabla_h w \cdot \nabla_h v dx$$

Additionally, we are going to define the norm on Ω in terms of an inner product, then the $L^2(\Omega)$ norm is

$$||v||^2_{L^2(\Omega)} := (v, v)_{L^2(\Omega)} = \int_\Omega v \cdot v ds.$$

This also allows to define the $H^1(\Omega)$ norm as follows

$$||v||^2_{H^1(\Omega)} := ||v||^2_{L^2(\Omega)} + ||\nabla v||^2_{L^2(\Omega)}.$$

At last we note that the bilinear form $B_h(\cdot,\cdot)$ is endowed with the mesh dependent norm $||\cdot||_{DG}$ defined by:

$$||v||^2_{DG} := ||\nabla_h v||^2_{L^2(\Omega)} + \sum_{e \in \mathscr{F}_h} ||\alpha^{1/2} [\![v]\!]||^2_{L^2(e)}$$

Now, we show the primal forms of other discontinuous Galerkins in the following Table 1.1, by Arnold [7]

Table 1.1: Primal forms for the DG methods

Method	$B_h(w,v)$
Bassi-Rebay	$a(w,v)-(\{\nabla_h w\},[\![v]\!])-([\![w]\!],\{\nabla_h v\})+(r([\![w]\!]),r([\![v]\!]))_\Omega$
Brezzi et al.	$a(w,v)-(\{\nabla_h w\},[\![v]\!])-([\![w]\!],\{\nabla_h v\})+(r([\![w]\!]),r([\![v]\!])_\Omega+\alpha_r(w,v)$
LDG	$a(w,v)-(\{\nabla_h w\},[\![v]\!])-([\![w]\!],\{\nabla_h v\})$
	$+\langle\beta\cdot[\![w]\!],[\![\nabla v]\!]\rangle+\langle[\![\nabla w]\!],\beta\cdot[\![v]\!]\rangle$
	$+(r([\![w]\!]),r([\![v]\!]))_\Omega+(r([\![w]\!]),l(\beta\cdot[\![v]\!]))_\Omega$
	$+(l(\beta\cdot[\![w]\!]),r([\![v]\!]))_\Omega+(l(\beta\cdot[\![w]\!]),l(\beta[\![v]\!]))_\Omega+\alpha_j(w,v)$
IP	$a(w,v)-(\{\nabla_h w\},[\![v]\!])-([\![w]\!],\{\nabla_h v\})+\alpha_j(w,v)$
Bassi et al.	$a(w,v)-(\{\nabla_h w\},[\![v]\!])-([\![w]\!],\{\nabla_h v\})+\alpha_r(w,v)$
Baumann-Oden	$a(w,v)-(\{\nabla_h w\},[\![v]\!])+([\![w]\!],\{\nabla_h v\})$
NIPG	$a(w,v)-(\{\nabla_h w\},[\![v]\!])+([\![w]\!],\{\nabla_h v\})+\alpha_j(w,v)$
Babuška-Zlámal	$a(w,v)+\alpha_j(w,v)$
Brezzi et al.	$a(w,v)+\alpha_r(w,v)$

with the following choice of the numerical fluxes shown in table 1.2

Table 1.2: Numerical fluxes for the DG methods

Method	$\hat{u}_K$	$\hat{\varrho}_K$
Bassi-Rebay	$\{u_h\}$	$\{\varrho_h\}$
Brezzi er al.	$\{u_h\}$	$\{\varrho_h\}-\alpha_r([\![u_h]\!])$
LDG	$\{u_h\}-\beta\cdot[\![u_h]\!]$	$\{\varrho_h\}+\beta[\![\varrho_h]\!]-\alpha_j([\![u_h]\!])$
IP	$\{u_h\}$	$\{\nabla_h u_h\}-\alpha_j([\![u_h]\!])$
Bassi et al.	$\{u_h\}$	$\{\nabla_h u_h\}-\alpha_r([\![u_h]\!])$
Baumann-Oden	$\{u_h\}+n_K\cdot[\![u_h]\!]$	$\{\nabla_h u_h\}$
NIPG	$\{u_h\}+n_K\cdot[\![u_h]\!]$	$\{\nabla_h u_h\}-\alpha_j([\![u_h]\!])$
Babuška-Zlámal	$(u_h\mid_K)\mid_{\partial K}$	$-\alpha_j([\![u_h]\!])$
Brezzi et al.	$(u_h\mid_K)\mid_{\partial K}$	$-\alpha_r([\![u_h]\!])$

In this work, we are going to concentrate on a Symmetric Interior Penalty method (**IP**), as one of the well-studied and computationally stable, easy to implement, and has only a "classical" bilinear form, jump and average operators and the penalty term. Another method which will be covered in this work is a Local Discontinuous Galerkin method (**LDG**), which has all of the aforementioned forms of the IP method, but also contains additional β penalty terms and employs the lifting operators, which adds special interest for the code implementation, and allows the use of additional splines. Moreover, the LDG method, possesses bilinear forms which can be found in any other existing discontinuous Galerkin FEMs. Implementing the LDG method would allow implementation and testing for all existing dG methods.

Chapter 2

Model problem

2.1 Discontinuous Galerkin Method

2.1.1 Variational formulation

In order to examine and analyse the mentioned method we first give the definition of the Local Discountinuous Galerkin method, given in the table 1.1. We start by recalling definition of the space 1.8 for the rectangular space decomposition:

$$V_{hp} := \{v_h \in L^2(\Omega) \quad : v\,|_K \circ T_K \in \mathcal{Q}_p \quad \forall K \in \mathcal{T}_h\}.$$

Then the the Local Discontinuous Galerkin method is:
Find $u_h \in V_{hp}$, such that

$$B_h(u_h, v) = F_h(v), \tag{2.1}$$

for an arbitrary test function $v \in V_{hp}$.

The bilinear form

$$B_h(\cdot,\cdot) : V_{hp} \times V_{hp} \to \mathbb{R},$$

is defined as

$$\begin{aligned} B_h(u_h, v) &= \int_\Omega \nabla u_h \cdot \nabla v dx - \int_\Gamma ([\![u_h]\!] \cdot \{\nabla v\} + \{\nabla u_h\} \cdot [\![v]\!]) ds \qquad (2.2)\\ &+ \int_{\Gamma^0} (\beta \cdot [\![u_h]\!][\![\nabla v]\!] + [\![\nabla u_h]\!]\beta \cdot [\![v]\!]) ds \\ &+ \theta \int_\Omega \Big[r([\![u_h]\!]) + l(\beta \cdot [\![u_h]\!]) \Big] \cdot \Big[r([\![v]\!]) + l(\beta \cdot [\![v]\!]) \Big] dx \\ &+ \alpha^j(u_h, v). \end{aligned}$$

The bilinear form is trivially positive definite [7]. The right hand side $F_h(\cdot) : V_{hp} \to \mathbb{R}$, is given as

$$F_h(v) = \int_\Omega f v dx$$

Bilinear form (2.2) can be compacted to Symmetric Interior Penalty method (**SIPG**), by setting $\theta = 0$ and $\beta = 0$ [6]. In order to obtain the Local Discontinuous Galerkin method

(**LDG**), we set [21] $\theta = 1$ and allow $\beta \in \mathbb{R}^d$ be a uniformly bounded vector, with special case for null vector β where we have a superconvergent LDG method. Additionally, we set α_j as in the works [41],[5], as

$$\alpha_j := \gamma N_{pd_K}^2 h_K^{-1},$$

where h_K and N_{pd_K} denote diameter and polynomial degree of element $K \in \mathcal{T}_h$, and $\gamma \geq 1$ is an arbitrary chosen scalar value. For simplicity, we restrict ourselves to uniform values for diameter $h = \max(h_K)$ and polynomial degree $N_{pd} = \max(N_{pd_K})$, uniform partition.

Using agreed notation, for the sake of simplicity, we can perform split of the primal bilinear form (2.2) into shorter forms. This allows to analyse each part of the primal form separately.

$$a(u_h, v) = (\nabla u_h, \nabla v)_\Omega \tag{2.3}$$
$$a_{uv}(u_h, v) = ([\![u_h]\!], \{\nabla v\})_\Gamma \tag{2.4}$$
$$a_{vu}(u_h, v) = (\{\nabla u_h\}, [\![v]\!])_\Gamma \tag{2.5}$$
$$a_{bu}(u_h, v) = (\beta \cdot [\![u_h]\!], [\![\nabla v]\!])_{\Gamma^0} \tag{2.6}$$
$$a_{ub}(u_h, v) = ([\![\nabla u_h]\!], \beta \cdot [\![v]\!])_{\Gamma^0} \tag{2.7}$$
$$a_s(u_h, v) = \alpha^j(u_h, v) = (\alpha \cdot [\![u_h]\!], [\![v]\!])_\Gamma. \tag{2.8}$$

Forms (2.4,2.5) are the same, and in matrix form represented as an original and a transpose of one. We note the same for forms (2.6,2.7). And lifting operators, after multiplication:

$$a_{rr}(u_h, v) = (r([\![u_h]\!]), r([\![v]\!]))_\Omega \tag{2.9}$$
$$a_{rl}(u_h, v) = (r([\![u_h]\!]), l(\beta \cdot [\![v]\!]))_\Omega \tag{2.10}$$
$$a_{lr}(u_h, v) = (l(\beta \cdot [\![u_h]\!]), r([\![v]\!]))_\Omega \tag{2.11}$$
$$a_{ll}(u_h, v) = (l(\beta \cdot [\![u_h]\!]), l(\beta [\![v]\!]))_\Omega. \tag{2.12}$$

Forms (2.10,2.11) are the same, and in matrix form represented as an original and a transpose of one, as in the previous case for the forms arising from the edge interaction.

To introduce the matrix formulation of the problem, we will first define how the functions $v \in V_{hp}$ can be represented as a linear combination of basis functions, introduce the notation for the basis functions, and perform an analysis for the choice of the basis functions. Later in the chapter, in Section **2.3**, we provide the details of the implementation of each of these terms.

2.2 Choice of the basis functions

We first recall the definition of the finite partitioning $\mathcal{T}_h$ 1.3 of the domain Ω, for **total number of elements** N_{mesh} :

$$\mathcal{T}_h = \{K_i, \quad i = 1, \ldots, N_{mesh}\}, \tag{2.13}$$

where each rectangular (d cube for higher dimensions) element

$$K_i, \quad i = 1, \dots, N_{mesh},$$

can be obtained applying invertible affine mapping $T_{K_i}(\cdot)$ to the reference element K^{ref}

$$K_i = T_{K_i}(K^{ref}), \quad i = 1, \dots, N_{mesh}.$$

We define N_{loc} to be the **total number of basis functions** on each element $K_i, \quad i = 1, \dots, N_{mesh}$.

Now, introducing arbitrary chosen basis functions $\phi_i \in V_{hp}, i = 1, \dots, N_{mesh} \cdot N_{loc}$

$$c^u = (c_i^u)_{N_{mesh} \cdot N_{loc}}, \tag{2.14}$$

for

$$u_h(x) = \sum_{i=1}^{N_{mesh} \cdot N_{loc}} c_i^u \phi_i(x), \quad u_h \in V_{hp}, \quad x \in \Omega \tag{2.15}$$

over all elements, one could set a matrix system notation for a problem.

Notation in the form $(\cdot)_m$ represents a vector with length m. Similar notation is used for matrices $(\cdot)_m^n$ to explicitly indicate matrix with m columns and n rows.

And to represent a test function $v(x) \in V_{hp}$, we can use linear combination of basis functions $\phi_i(x) \in V_{hp}, \quad i = 1, \dots, N_{mesh} \cdot N_{loc}$, such that:

$$v(x) = \sum_{i=1}^{N_{mesh} \cdot N_{loc}} c_i^v \phi_i(x),$$

for a vector $c^v = (c_i^v), \quad i = 1, \dots, N_{mesh} \cdot N_{loc}$.

Before defining actual basis function, we should determine properties and restrictions for such functions. One should note that chosen basis functions should imply the following property:

- $\phi_i(x)$ only has a non-zero value on one element and vanishes on all others,

We are going to use N_{pd} as a maximal **polynomial degree of a function**. We also note, that our basis functions on the d cube are the product of one-dimensional functions, and because to compute the values of the one-dimensional hierarchical polynomial function of given degree N_{pd} on a chosen geometry with the dimension d, we need to compute all $N_{pd} + 1$ values of lower order functions of the same kind in each variable, yielding total as

$$N_{loc} = (N_{pd} + 1)^d, \tag{2.16}$$

this creates a **relation for the number of basis functions N_{loc} on the element K_i and polynomial degree N_{pd} of the hierarchical basis function for spatial dimension d.**

We are going to define reference basis functions for reference element K^{ref}

$$\phi_k^{ref}(t), \quad k = 1, \dots, N_{loc},$$

where t is a set of coordinates on a reference element for the chosen rectangular partitioning. For the reference element we have $t \in (-1,1)^d$.

To obtain the local basis function for current element $K_i, \quad i = 1,\dots,N_{mesh}$ of the employed decomposition we use invertible affine mapping

$$x = F_{K_i}(t),$$

which allows the following:

$$\phi_{i,k}^{loc}(x) = \begin{cases} \phi_k^{ref}(F_{K_i}^{-1}(x)), \quad x \in K_i \\ 0, \quad x \notin K_i \end{cases}$$

where $i = 1,\dots,N_{mesh}$ and $k = 1,\dots,N_e$.

We note that chosen domain $\Omega \subset \mathbb{R}^d$. For $\phi : \Omega \to \mathbb{R}$ and $\bar{\phi} : K^{ref} \to \mathbb{R}$ we have

$$\phi(x) = \bar{\phi}(F^{-1}(x)) = \bar{\phi} \circ F^{-1}(x),$$

which also yields:

$$x = F(t) \Rightarrow dx = \left|\frac{\partial F}{\partial t}\right| dt.$$

We also perform an analysis of the chosen mapping. For the reference element $K^{ref} \subset \mathbb{R}^d$, transformation defined as

$$F := \begin{cases} K^{ref} \Rightarrow \Omega \\ t \to x \end{cases}$$

Now to evaluate the integral of $u, v \in \Omega$ using $\bar{u}, \bar{v} \in K^{ref}$ we transform to the reference element:

$$\int_\Omega uv dx = \int_\Omega u(F^{-1}(x))v(F^{-1}(x))dx = \int_{K^{ref}} \bar{u}\bar{v} \left|\frac{\partial F}{\partial t}\right| dt$$

And to represent a gradient $\nabla_x u(x)$ on the reference element with $\nabla_t \bar{u}(t)$ we do the following:

$$\nabla_x u = \nabla_x \bar{u}(F^{-1}(x)) = ((\partial_{x_k} F^{-1}(x))\nabla_t \bar{u} \circ F^{-1})_k = (\sum_{i=1}^{d} \frac{\partial t_i}{\partial x_k} \partial_{t_i} \bar{u} \circ F^{-1})_k,$$

for each spatial coordinate $x_k, \quad k = 1,\dots,d$. Therefore for the integral estimation we have:

$$\begin{aligned}
&\int_\Omega \nabla_x u \cdot \nabla_x v dx \\
=& \int_\Omega \nabla_x \bar{u}(F^{-1}(x)) \cdot \nabla_x \bar{v}(F^{-1}(x)) dx \\
=& \int_\Omega \left(\sum_{k=1}^d \left[(\partial_{x_k} F^{-1}(x)) \nabla_t \bar{u} \circ F^{-1} \right] \cdot \left[(\partial_{x_k} F^{-1}(x)) \nabla_t \bar{v} \circ F^{-1} \right] \right) dx \\
=& \int_{K^{ref}} \left(\sum_{k=1}^d \left[(\partial_{x_k} F^{-1}(x)) \circ F \nabla_t \bar{u} \right] \cdot \left[(\partial_{x_k} F^{-1}(x)) \circ F \nabla_t \bar{v} \right] \right) \left| \frac{\partial F}{\partial t} \right| dt \\
=& \int_{K^{ref}} \left(\sum_{k=1}^d \left[\sum_{i=1}^d \frac{\partial t_i}{\partial x_k} \partial_{t_i} \bar{u} \right] \cdot \left[\sum_{j=1}^d \frac{\partial t_j}{\partial x_k} \partial_{t_j} \bar{v} \right] \right) \left| \frac{\partial F}{\partial t} \right| dt \\
=& \int_{K^{ref}} \sum_{i=1}^d \sum_{j=1}^d \left(\sum_{k=1}^d \frac{\partial t_i}{\partial x_k} \frac{\partial t_j}{\partial x_k} \right) \partial_{t_i} \bar{u} \partial_{t_j} \bar{v} \left| \frac{\partial F}{\partial t} \right| dt
\end{aligned}$$

And due to $t = F^{-1} \circ F(t)$, for identity matrix I

$$I = \nabla_t t = \nabla_t F^{-1} \circ F(t) = ((\nabla_x F^{-1}) \circ F(t)) \nabla_t F(t). \tag{2.17}$$

This means

$$I = \left(\frac{\partial t_i}{\partial t_j} \right)_{i,j} = \left(\frac{\partial t_i(x(t))}{\partial t_j} \right)_{i,j} = \left(\sum_{k=1}^d \frac{\partial t_i}{\partial x_k} \frac{\partial x_k}{\partial t_j} \right)_{i,j} = \left(\frac{\partial t_i}{\partial x_k} \right)_{i,k} \left(\frac{\partial x_k}{\partial t_j} \right)_{k,j},$$

And this leads to

$$\left(\frac{\partial t_i}{\partial x_k} \right)_{i,k} = \left[\left(\frac{\partial x_k}{\partial t_j} \right)_{k,j} \right]^{-1}.$$

For the sake of completeness we also demonstrate the Jacobian matrix computation. The following formulae can be implemented easily and do not depend on a choice of basis functions. The Jacobian in two dimensions is

$$\begin{pmatrix} \frac{\partial t_1}{\partial x_1} & \frac{\partial t_1}{\partial x_2} \\ \frac{\partial t_2}{\partial x_1} & \frac{\partial t_2}{\partial x_2} \end{pmatrix} = \begin{pmatrix} \frac{\partial x_1}{\partial t_1} & \frac{\partial x_1}{\partial t_2} \\ \frac{\partial x_2}{\partial t_1} & \frac{\partial x_2}{\partial t_2} \end{pmatrix}^{-1} = \frac{1}{\left| \frac{\partial F}{\partial t} \right|} \begin{pmatrix} \frac{\partial x_2}{\partial t_2} & -\frac{\partial x_1}{\partial t_2} \\ -\frac{\partial x_2}{\partial t_1} & \frac{\partial x_1}{\partial t_1} \end{pmatrix} \tag{2.18}$$

with

$$\left| \frac{\partial F}{\partial t} \right| = \frac{\partial x_1}{\partial t_1} \frac{\partial x_2}{\partial t_2} - \frac{\partial x_1}{\partial t_2} \frac{\partial x_2}{\partial t_1}.$$

In three dimensions,

$$\begin{aligned}
\frac{\partial F}{\partial t} &= \frac{\partial x}{\partial t_1} \left(\frac{\partial x}{\partial t_2} \times \frac{\partial x}{\partial t_3} \right) \\
&= \frac{\partial x_1}{\partial t_1} \left(\frac{\partial x_2}{\partial t_2} \frac{\partial x_3}{\partial t_3} - \frac{\partial x_3}{\partial t_2} \frac{\partial x_2}{\partial t_3} \right) - \frac{\partial x_2}{\partial t_1} \left(\frac{\partial x_1}{\partial t_2} \frac{\partial x_3}{\partial t_3} - \frac{\partial x_3}{\partial t_2} \frac{\partial x_1}{\partial t_3} \right) + \frac{\partial x_3}{\partial t_1} \left(\frac{\partial x_1}{\partial t_2} \frac{\partial x_2}{\partial t_3} - \frac{\partial x_2}{\partial t_2} \frac{\partial x_1}{\partial t_3} \right)
\end{aligned}$$

and

$$\left(\frac{\partial t_i}{\partial x_k}\right)_{i,k} = \frac{1}{\left|\frac{\partial F}{\partial t}\right|}\begin{pmatrix} \frac{\partial x_2}{\partial t_2}\frac{\partial x_3}{\partial t_3} - \frac{\partial x_3}{\partial t_2}\frac{\partial x_2}{\partial t_3} & -\frac{\partial x_1}{\partial t_2}\frac{\partial x_3}{\partial t_3} + \frac{\partial x_3}{\partial t_2}\frac{\partial x_1}{\partial t_3} & \frac{\partial x_1}{\partial t_2}\frac{\partial x_2}{\partial t_3} - \frac{\partial x_2}{\partial t_2}\frac{\partial x_1}{\partial t_3} \\ -\frac{\partial x_2}{\partial t_1}\frac{\partial x_3}{\partial t_3} + \frac{\partial x_3}{\partial t_1}\frac{\partial x_2}{\partial t_3} & \frac{\partial x_1}{\partial t_1}\frac{\partial x_3}{\partial t_3} - \frac{\partial x_3}{\partial t_1}\frac{\partial x_1}{\partial t_3} & -\frac{\partial x_1}{\partial t_1}\frac{\partial x_2}{\partial t_3} + \frac{\partial x_2}{\partial t_1}\frac{\partial x_1}{\partial t_3} \\ \frac{\partial x_2}{\partial t_1}\frac{\partial x_3}{\partial t_2} - \frac{\partial x_3}{\partial t_1}\frac{\partial x_2}{\partial t_2} & -\frac{\partial x_1}{\partial t_1}\frac{\partial x_3}{\partial t_2} + \frac{\partial x_3}{\partial t_1}\frac{\partial x_1}{\partial t_2} & \frac{\partial x_1}{\partial t_1}\frac{\partial x_2}{\partial t_2} - \frac{\partial x_2}{\partial t_1}\frac{\partial x_1}{\partial t_2} \end{pmatrix} \tag{2.19}$$

Here one should note that the global basis function for an approximation and local basis function on an element are bounded using the mapping between them on a neighbouring elements.

For the edge-based integral in our primal form, we can use different basis functions, which have the necessary properties for the edge-based computations, e.g. have zero values on all neighbouring edges, except one.

2.2.1 One dimensional functions

We now define one dimensional functions, which can be used as a basis.

2.2.1.1 Antiderivative of Legendre polynomials

Antiderivative of Legendre polynomials (**ADLP**) [40, p19], can be used as hierarchical recursive basis functions, with arbitrary polynomial degree N_{pd}. They are based on a Legendre polynomials, which are hierarchical functions. Legendre polynomials are defined in the following way:

$$\begin{aligned} L_0(x) &= 1, \\ L_1(x) &= x, \\ L_n(x) &= \frac{2n-1}{n} x L_{n-1}(x) - \frac{n-1}{n} L_{n-2}(x). \end{aligned}$$

The first derivatives are given in recursive form:

$$\begin{aligned} L_0'(x) &= 0, \\ L_1'(x) &= 1, \\ (1-x^2)L_n'(x) &= -nxL_n(x) + nL_{n-1}(x), \quad n \geq 1. \end{aligned}$$

The **ADLP** are defined in the following way:

$$\begin{aligned} \mathscr{L}_0(x) &= \frac{1}{2}(1-x), \\ \mathscr{L}_1(x) &= \frac{1}{2}(1+x), \\ \mathscr{L}_n(x) &= \frac{1}{2n-1}(L_n(x) - L_{n-2}(x)) = \int_{-1}^{x} L_{n-1}(y)dy, \quad n \geq 2. \end{aligned} \tag{2.20}$$

The important and simplifying property on the reference element $\square := [-1,1]^2$ can be achieved:

$$\mathscr{L}_n(\pm 1) = 0, \quad n \geq 2$$

Also, there holds:

$$\begin{aligned}\mathscr{L}_0' &= -\frac{1}{2},\\ \mathscr{L}_1' &= \frac{1}{2},\\ \mathscr{L}_n' &= L_{n-1}, \quad n \geq 2.\end{aligned}$$

2.2.1.2 C1 (b-spline) polynomials

Another hierarchical recursive set of basis functions the C1 (b-spline) polynomials [40, p20], which we can employ, for an arbitrary polynomial degree $N_{pd} \geq 3$. This choice restricts us to rectangular partitions. Main advantage and subsequent importance of the C1 basis functions is discussed in Section **2.3**. An effect of the C1 basis function on a solver is discussed further in Section **3.1**. C1 basis functions are based on an Anti-derivative of a Legendre polynomial:

$$\begin{aligned}C_0(x) &= \frac{1}{4}(1-x)^2(2+x),\\ C_1(x) &= \frac{1}{4}(1+x)^2(2-x),\\ C_2(x) &= \frac{1}{4}(1+x)(1-x)^2,\\ C_3(x) &= \frac{1}{4}(1+x)^2(x-1),\\ C_n(x) &= (1-x^2)\mathscr{L}_{n-2}(x), \quad n \geq 4\end{aligned} \tag{2.21}$$

The desired properties, are:

$$\begin{gathered}C_0(-1)=1, C_0(1)=0, C_0'(-1)=0, C_0'(1)=0,\\ C_1(-1)=0, C_1(1)=1, C_1'(-1)=0, C_1'(1)=0,\\ C_2(-1)=0, C_2(1)=0, C_2'(-1)=1, C_2'(1)=0,\\ C_3(-1)=0, C_3(1)=0, C_3'(-1)=0, C_3'(1)=1,\\ C_n(\pm1)=C_n'(\pm1)=0, \quad n \geq 4\end{gathered} \tag{2.22}$$

The derivatives of the C1 polynomials are:

$$\begin{aligned}C_0'(x) &= -\frac{3}{4}(1-x^2),\\ C_1'(x) &= \frac{3}{4}(1-x^2),\\ C_2'(x) &= \frac{1}{4}(1-x)(-1-3x),\\ C_3'(x) &= \frac{1}{4}(1+x)(3x-1),\\ C_n'(x) &= -2x\mathscr{L}_{n-2}(x)+(1-x^2)L_{n-3}(x), \quad n \geq 4\end{aligned}$$

2.2.1.3 C1 (b-spline) polynomials of the second kind

Alternatively, C1 basis function setting can be devised as:

$$\begin{aligned}
\bar{C}_0(x) &= \frac{1}{4}(1-x)^2(2+x),\\
\bar{C}_1(x) &= \frac{1}{4}(1+x)^2(2-x),\\
\bar{C}_2(x) &= \frac{1}{4}(1+x)(1-x)^2,\\
\bar{C}_3(x) &= \frac{1}{4}(1+x)^2(x-1),\\
\bar{C}_n(x) &= (1-x^2)^2 L_{n-4}(x), \quad n \geq 4,
\end{aligned}$$

with the derivative defined as:

$$\begin{aligned}
\bar{C}_0'(x) &= -\frac{3}{4}(1-x^2),\\
\bar{C}_1'(x) &= \frac{3}{4}(1-x^2),\\
\bar{C}_2'(x) &= \frac{1}{4}(1-x)(-1-3x),\\
\bar{C}_3'(x) &= \frac{1}{4}(1+x)(3x-1),\\
\bar{C}_4'(x) &= 4x(x^2-1)\\
\bar{C}_n'(x) &= 4x(x^2-1)L_{n-4}(x) + (1-x^2)^2 L_{n-4}'(x)\\
&= 4x(x^2-1)L_{n-4}(x)\\
&+ (1-x^2)\big[-(n-4)xL_{n-4}(x) + (n-4)L_{n-5}(x)\big], \quad n \geq 5.
\end{aligned}$$

We note that this kind of C1 basis functions is not employed in this work, but is given for the sake of completeness.

2.2.2 Two dimensional functions

2.2.2.1 Raviart-Thomas elements

Calculation of vector-valued global basis functions requires normal direction $\mathbf{n}$, and for two vector-valued functions $\psi_{i,k}^{loc}$ and $\psi_{j,l}^{loc}$ on the joint edge $e_k = e_l$ for corresponding neighbouring elements K_i and K_j ([39]):

$$\psi_{i,k}^{loc}(e_k) \cdot \mathbf{n} = \psi_{j,l}^{loc}(e_l) \cdot \mathbf{n}.$$

To solve this problem, one can set weighting relation:

$$w(i,k) \cdot \psi_{i,k}^{loc}(e_k) \cdot \mathbf{n} = w(j,l) \cdot \psi_{j,l}^{loc}(e_l) \cdot \mathbf{n}$$

setting $w(i,k) := 1$ for cases where element K_i does not have a neighbour on edge e_k. Otherwise, if one sets $w(i,k) := 1$ for element's K_i edge e_k and for the joint edge e_l of element K_j, if $j > i$ one gets:

$$w(j,l) = \frac{\psi_{i,k}^{loc}(e_k)\cdot \mathbf{n}}{\psi_{j,l}^{loc}(e_l)\cdot \mathbf{n}}$$

2.2.2.2 Extension to higher dimension

In order to use the one-dimensional polynomials $f(x)$ on a two-dimensional mesh, we can extend them using their tensor-product:

$$\phi(x_1,x_2) := f(x_1)\cdot f(x_2),$$

In this case, global basis functions would look like:

$$\phi(x_1,x_2) := \mathscr{L}_n(x_1)\cdot \mathscr{L}_n(x_2),$$

with $n = 1,\ldots,N_{pd}$, for the anti-derivative of Legendre polynomials, and

$$\phi(x_1,x_2) := C_n(x_1)\cdot C_n(x_2),$$

with $n = 1,\ldots,N_{pd}$, for the C1 polynomials. In order to restrict local basis functions' contributions on a matrix, one can introduce the penalty factor (not to be confused with DG penalty term α) for the basis functions on reference elements. The penalty factor for different shapes of the element, is chosen such that it yields zero values on the boundaries. For the reference rectangle $\square := [-1,1]^2$, we apply the factors $(1-x_1^2)(1-x_2^2)$, and get the following form of the basis function on the reference element:

$$(1-x_1^2)(1-x_2^2)\phi(x_1,x_2)$$

which ensures that the basis function disappears on the edges of the rectangle, guaranteeing that we have a function with non-zero values only inside the element. This extension reduces number of non-zero basis functions per element down to N_{pd} . Important and desirable property of the C1 polynomials is the fact, that its derivative would be zero on all but one edges of the element. This can be shown, using the mapping:

$$\begin{aligned}\phi^{loc}(x) &= \phi^{ref}(F^{-1}(t))\\ n_e\cdot\nabla_x\phi^{loc}(x) &= n_e\cdot(J^T\nabla_t\phi^{ref}(t)),\end{aligned}$$

where n_e is a normal to the edge (face) of the reference element.

To give the details of the previous formula first note that

$$n_e\cdot(J\nabla_t\phi) = n_e^T J^T\nabla_t\phi = (Jn_e)^T\nabla_t\phi = (Jn_e)\cdot\nabla_t\phi$$

Knowing, that J is a Jacobian matrix, in 2-D:

$$\begin{pmatrix}\frac{\partial t_1}{\partial x_1} & \frac{\partial t_1}{\partial x_2}\\ \frac{\partial t_2}{\partial x_1} & \frac{\partial t_2}{\partial x_2}\end{pmatrix} = \begin{pmatrix}\frac{\partial x_1}{\partial t_1} & \frac{\partial x_1}{\partial t_2}\\ \frac{\partial x_2}{\partial t_1} & \frac{\partial x_2}{\partial t_2}\end{pmatrix}^{-1} = \frac{1}{|\frac{\partial F}{\partial t}|}\begin{pmatrix}\frac{\partial x_2}{\partial t_2} & -\frac{\partial x_1}{\partial t_2}\\ -\frac{\partial x_2}{\partial t_1} & \frac{\partial x_1}{\partial t_1}\end{pmatrix}$$

and n_e is defined as

$$n_e = t^\perp = \begin{pmatrix} \frac{\partial x_2}{\partial t_2} \\ -\frac{\partial x_1}{\partial t_2} \end{pmatrix},$$

where $(\cdot)^\perp$ denotes perpendicularity operation, and vector is written as a column-vector for readability.

For the Jn_e term we have:

$$\begin{pmatrix} \frac{\partial x_2}{\partial t_2} & -\frac{\partial x_2}{\partial t_1} \\ -\frac{\partial x_1}{\partial t_2} & \frac{\partial x_1}{\partial t_1} \end{pmatrix} \cdot \begin{pmatrix} \frac{\partial x_2}{\partial t_2} \\ -\frac{\partial x_1}{\partial t_2} \end{pmatrix} = \begin{pmatrix} \left(\frac{\partial x_2}{\partial t_1}\right)^2 + \left(\frac{\partial x_1}{\partial t_2}\right)^2 \\ -\frac{\partial x_2}{\partial t_1}\frac{\partial x_2}{\partial t_1} - \frac{\partial x_1}{\partial t_2}\frac{\partial x_1}{\partial t_1} \end{pmatrix} = \begin{pmatrix} \left(\frac{\partial x_2}{\partial t_1}\right)^2 + \left(\frac{\partial x_1}{\partial t_2}\right)^2 \\ -\frac{\partial F}{\partial t_1} \cdot \frac{\partial F}{\partial t_2} \end{pmatrix}$$

We should also note, that Jacobian J is restricted on the edge e, and, according to our set-up, $\partial_{t_1} \phi^{ref}|_e = 0$. We get:

$$\begin{aligned} \begin{pmatrix} \left(\frac{\partial x_2}{\partial t_1}\right)^2 + \left(\frac{\partial x_1}{\partial t_2}\right)^2 \\ -\frac{\partial F}{\partial t_1} \cdot \frac{\partial F}{\partial t_2} \end{pmatrix} \cdot \begin{pmatrix} \partial_{t_1}\phi^{ref} \\ \partial_{t_2}\phi^{ref} \end{pmatrix} &= \begin{pmatrix} \left(\frac{\partial x_2}{\partial t_1}\right)^2 + \left(\frac{\partial x_1}{\partial t_2}\right)^2 \\ -\frac{\partial F}{\partial t_1} \cdot \frac{\partial F}{\partial t_2} \end{pmatrix} \cdot \begin{pmatrix} 0 \\ \partial_{t_2}\phi^{ref} \end{pmatrix} \\ &= -\frac{\partial F}{\partial t_1} \cdot \frac{\partial F}{\partial t_2} \cdot \partial_{t_2}\phi^{ref} \end{aligned} \tag{2.23}$$

Using this property, we need just one computation of one reference basis function per edge, which would reduce the computation time, taking into account the Gauss-Legendre quadrature scheme, by N_{pd}^2 .

2.2.3 Three dimensional case

Three dimensional extension of the basis function is done similarly to the two dimensional case. We use a tensor-product of one dimensional function $f(x)$ to obtain a three dimensional function. For simplicity, with slight abuse of notation:

$$\phi(x_1, x_2, x_3) := f(x_1) \cdot f(x_2) \cdot f(x_3). \tag{2.24}$$

For a reference cube, we are using a special set $[-1,1]^3$ of all 3-tuples of numbers in the interval $[-1,1]$. It is easily verifiable, that to guarantee desirable properties, namely to eliminate the basis function contribution on the face of the reference cube, we can use the penalty factor $(1-x_1^2)(1-x_2^2)(1-x_3^2)$. This yields the following form:

$$(1-x_1^2)(1-x_2^2)(1-x_3^2)\phi(x_1, x_2, x_3) \tag{2.25}$$

Anti-derivative of Legendre polynomials, would allow to construct the following three-dimensional basis functions:

$$\phi(x_1, x_2, x_3) := \mathscr{L}_n(x_1) \cdot \mathscr{L}_n(x_2) \cdot \mathscr{L}_n(x_3),$$

with $n = 1, \ldots, N$. Applying the penalty factor (2.25) allows us to use it as a global basis function on a discontinuous space.

Product of three C1 polynomials can be used as three-dimensional basis function: And

$$\phi(x_1, x_2, x_3) := C_n(x_1) \cdot C_n(x_2) \cdot C_n(x_3),$$

with $n = 1, \ldots, N$. As in the case of **ADLP**, applying the appropriate penalty factors, yields the global basis functions. Major property of interest of the C1 polynomial is the fact, that its derivative is zero on all but one face of the element. This can be shown, using the mapping:

$$\begin{aligned} \phi^{loc}(x) &= \phi^{ref}(F^{-1}(t)) \\ n_e \cdot \nabla_x \phi^{loc}(x) &= n_e \cdot J^T \nabla_t \phi^{ref}(t). \end{aligned}$$

Lemma 2.2.1. *On a reference cube $K^{ref} = [-1, 1]^3$ with variables given by t_1, t_2, t_3, for any function, restricted on a face $e_r \in \partial K^{ref}$, $\quad r = 1, \ldots, 6$, we have*

$$\nabla_x u|_{e_r} = (\frac{\partial F}{\partial t})^{-T} \nabla_t \tilde{u}|_{e_r} = 0$$

Proof.

$$\nabla_x u|_{e_r} = (\frac{\partial F}{\partial t})^{-T} \nabla_t \tilde{u}|_{e_r},$$

holds due to the construction.

From (2.24), for reference basis functions we have

$$\tilde{u}_{k,l,m} = \phi_k^{ref}(t_1)\phi_l^{ref}(t_2)\phi_m^{ref}(t_3).$$

Considering the lower face given by $t_3 = -1, (t_1, t_2) \in [-1, 1]^2$, we have derivative being equal to zero, as it follows from (2.22)

$$\begin{aligned} \partial_{t_1}\tilde{u}_{k,l,m} &= \partial_{t_1}\phi_k^{ref}(t_1)\phi_l^{ref}(t_2)\underbrace{\phi_m^{ref}(t_3)|_{t_3=-1}}_{\phi_m^{ref}(-1)=0} &= 0 \\ \partial_{t_2}\tilde{u}_{k,l,m} &= \phi_k^{ref}(t_1)\partial_{t_2}\phi_l^{ref}(t_2)\underbrace{\phi_m^{ref}(t_3)|_{t_3=-1}}_{\phi_m^{ref}(-1)=0} &= 0 \\ \partial_{t_3}\tilde{u}_{k,l,m} &= \phi_k^{ref}(t_1)\phi_l^{ref}(t_2)\underbrace{\partial_{t_3}\phi_m^{ref}(t_3)|_{t_3=-1}}_{\partial_{t_3}\phi_m^{ref}(-1)=0.} &= 0 \end{aligned}$$

The rest follows similarly. □

Now that all the necessary basis functions are defined, we can perform the matrix analyses.

2.3 Matrix setup

Using the defined basis functions, we can now define the process of matrix computation. We can perform analyses for each of the small forms shown in (2.3) - (2.8) and (2.9)-(2.12).

We start with volume integral computation (2.3).

2.3.1 Volume integral

For bilinear form $a(u_h, v) = (\nabla u_h, \nabla v)_\Omega$, set the matrix

$$A := (a_{p,q})_{N_{mesh}\cdot N_{loc}}^{N_{mesh}\cdot N_{loc}}.$$

Bilinear form could be calculated using the setting (2.15) and arbitrary chosen set of basis functions:

$$a_{p,q} = \int_\Omega \nabla\phi_p \cdot \nabla\phi_q dx,$$

where $p, q = 1, \ldots, N_{mesh} \cdot N_{loc}$. We choose **local** basis functions such that:

$$\phi_p = \sum_{p=\text{map}(i,k)} \phi_{i,k}^{loc}, \tag{2.26}$$

where $\text{map}(i, k)$ **represents the numbering scheme of the local basis function** $\phi_{i,k}^{loc}$ **on** K_i **of polynomial degree at most** N_{pd} **in relation to the numbering of the global basis function** ϕ_p, $i = 1, \ldots, N_{mesh}$ **and** $k = 1, \ldots, N_{loc}$.

To explain a bit further this notation we first note, that due to the discontinuities we only really have local basis functions and introduce the notation for global basis functions only for completeness. Hence the $\text{map}(i, k)$ structure is more suitable in the dG setting. With the $p = \text{map}(i, k)$ structure, we first go through all i-indexed elements of the mesh, then we observe all local functions $\phi_{i,k}^{loc}$ on that mesh element, which have indices $k = 1, \ldots, N_{loc}$, and add the observed values in the ϕ_p.

Introducing the same setup for the ϕ_q on elements K_j, with local basis functions $\phi_{j,l}^{loc}$, of polynomial degree at most N_{pd}, $j = 1, \ldots, N_{mesh}$ and $l = 1, \ldots, N_{loc}$, we can rewrite the bilinear form:

$$\begin{aligned} a_{p,q} &= \int_\Omega \nabla\Big(\sum_{p=\text{map}(i,k)} \phi_{i,k}^{loc}\Big) \cdot \nabla\Big(\sum_{q=\text{map}(j,l)} \phi_{j,l}^{loc}\Big) dx \\ &= \sum_{p=\text{map}(i,k)} \sum_{q=\text{map}(j,l)} \int_\Omega \nabla\phi_{i,k}^{loc} \cdot \nabla\phi_{j,l}^{loc} dx. \end{aligned}$$

Restricting the local basis functions choice such that that they are non-zero only on the element that they belong to, we need to compute them only on the nodes belonging to the element:

$$a_{p,q} = \sum_{p=\text{map}(i,k)} \sum_{q=\text{map}(i,l)} \int_{K_i \in \mathscr{T}_h(\Omega)} \nabla\phi_{i,k}^{loc} \cdot \nabla\phi_{i,l}^{loc} dx.$$

In order to simplify the construction of the local basis functions, we use reference element K^{ref} and invertible affine mapping T_K both defined in **section 1.3**, and corresponding affine transformation for the coordinates t of reference element to the coordinates x of the local element, such that $t = F_{K_i}^{-1}(x), \quad i = 1, \dots, N_{mesh}$:

$$a_{p,q} = \frac{1}{|J|} \sum_{p=\text{map}(i,k)} \sum_{q=\text{map}(i,l)} \int_{K^{ref}} J \nabla_t \phi_k^{ref}(F_{K_i}^{-1}(x)) \cdot \nabla_t \phi_l^{ref}(F_{K_i}^{-1}(x)) dt,$$

with Jacobian matrix J given by (2.17).

In order to calculate the resulting integral for the polynomial degree set to, at most, N_{pd}, we use Gauss-Legendre quadrature rule and need

$$N_g = N_{pd} \tag{2.27}$$

quadrature points and weights per each function:

$$a_{p,q} = \sum_{r=1}^{N_g} \sum_{s=1}^{N_g} \sum_{p=\text{map}(i,k)} \sum_{q=\text{map}(i,l)} J \omega_r \nabla_t \phi_k^{ref}(t_r, t_s) \cdot \omega_s \nabla_t \phi_l^{ref}(t_r, t_s),$$

where $t_r = F_{K_i}^{-1}(x_r), t_s = F_{K_i}^{-1}(x_s), \quad r,s = 1, \dots, N_g$ are Gauss-Legendre quadrature points, $\omega_r, \omega_s, \quad r,s = 1, \dots, N_g$ are Gauss-Legendre quadrature weights. Gauss-Legendre quadrature scheme is used to numerically estimate the integral. We can use precomputed list of Gauss-Legendre weights and nodes [38]. In the algorithmic view, we are using the function $(\omega, t) =$**Gauss(**N_{pd}**)**, which takes as input the polynomial degree, and outputs vectors $t \in \mathbb{R}^{N_g}$ of quadrature nodes, and vector $\omega \in \mathbb{R}^{N_g}$ of quadrature weights.

The use of the Gauss-Legendre rule is important for the cases when the constants present in the original PDE are non-zero. In addition, non-linear PDEs require extra precision, so we set the Gauss-Legendre quadrature to the value discussed prior.

To be more clear and to show implication, which arise here, we split the matrix computation in two parts - elementary matrix computation and the global matrix assembly.

Elementary matrix is defined as

$$A^{elem} := (a_{i,k,l}^{elem}), \quad i = 1, \dots, N_{mesh}, \quad k,l = 1, \dots, N_{loc}.$$

The elementary matrix computation follows as:

$$a_{i,k,l}^{elem} = \sum_{r=1}^{N_g} \sum_{s=1}^{N_g} \sum_{i=1}^{N_{mesh}} \sum_{k,l=1}^{N_{loc}} J \omega_r \nabla_t \phi_{i,k}^{ref}(t_r, t_s) \cdot \omega_s \nabla_t \phi_l^{ref}(t_r, t_s),$$

And the abstract, naive computation algorithm is shown in 1.

Algorithm 1 Local matrix A^{elem} computation. Naive. 2D

```
1: for $i = 1 : N_{mesh}$ do
2:     $(\omega, t) =$ Gauss($N_{pd}$)
3:     for $r, s = 1 : N_g$ do
4:         $(x_r, x_s) = F_{K_i}(t_r, t_s)$
5:         for $k, l = 1 : N_{loc}$ do
6:             $a^{elem}_{i,k,l} \leftarrow a^{elem}_{i,k,l} + J\omega_r\omega_s\nabla\phi^{loc}_{i,k}(x_r, x_s)\nabla\phi^{loc}_{i,l}(x_r, x_s)$
7:         end for
8:     end for
9: end for
```

The Algorithm 1 immediately yields number of necessary loop repetitions as

$$N_{mesh} \cdot N_g \cdot N_g \cdot N_{loc} \cdot N_{loc}.$$

However, due to the nature of the employed hierarchical basis functions, one would need at most additional $N_{pd} + 1$ (2.16) operations per k, l loop, to construct each basis function on each element, yielding the total number of necessary operations:

$$T^{naive}_{A^{elem}} \approx N_{mesh} \cdot N_g^2 \cdot N_{loc}^2 \cdot N_{pd} \cdot N_{pd}.$$

Here, we use approximate equality, as we omit the number of operations required for by the affine transformation, which is a constant and has very small overall impact. As we are interested in the impact of higher polynomial degree, we define total number of necessary operations in terms of polynomial degree N_{pd}. Taking into account, that we need at most $N_g = N_{pd}$ Gauss-Legendre data, per dimension, have $N_{loc} = (N_{pd} + 1)^d$ basis functions per d-cube. Additionally, hierarchical construction and computation of the basis function require at most N_{pd} operations per dimension. Hence, the naive implementation with all hidden cost is:

$$\begin{aligned} T^{naive}_{A^{elem}}(N_{pd}) &\approx N_{mesh} \cdot N_{pd}^d \cdot N_{loc}^d \cdot N_{pd}^2 \\ &\approx N_{mesh} \cdot N_{pd}^{d+2} \cdot (N_{pd} - 1)^d. \end{aligned}$$

Further we can simplify the number of operations by neglecting coefficients given by the constants:

$$T^{naive}_{A^{elem}}(d) \approx N_{mesh} \cdot N_{pd}^{(2d+2)}. \tag{2.28}$$

For the case $d = 2$,

$$T^{naive}_{A^{elem}}(2) \approx N_{mesh} \cdot N_{pd}^6.$$

We can optimise Algorithm 1. It is possible to calculate the whole vector of transformed Gauss-Legendre data, and access the necessary values in-loop:

$$x_r = F_{K_i}(t_r), \quad R = (x_r)^{N_g}, \quad r = 1, \ldots, N_g, \quad i = 1, \ldots, N_{mesh},$$

and $x_s = F_{K_i}(t_s), \quad S = (x_s)^{N_g}, \quad s = 1, \ldots, N_g, \quad i = 1, \ldots, N_{mesh}$. It is also possible to construct the basis functions at once, compute all the values, for each basis function, as they are hierarchical, and store the values in the vector form:

$$K(1:N_{loc}) \leftarrow \nabla\phi^{loc}_{i,1:N_{loc}}(R,S).$$

The optimised algorithm is shown below.

Algorithm 2 Optimised computation of the local matrix A^{elem}. 2D

```
1: for $i = 1 : N_{mesh}$ do
2:     $(\omega, t) =$ Gauss($N_{pd}$)
3:     for $r, s = 1 : N_g$ do
4:         $(x_r, x_s) = F_{K_i}(t_r, t_s)$
5:         $K(l) \leftarrow \nabla\phi^{loc}_{i,l}(x_r, x_s), \quad l = 1 : N_{loc}$
6:         $a^{elem}_{i,1:N_{loc},1:N_{loc}} \leftarrow a^{elem}_{i,1:N_{loc},1:N_{loc}} + J\omega_r K(1:N_{loc})\omega_s K(1:N_{loc})$
7:     end for
8: end for
```

The optimised version of the Algorithm 2 requires the following number of operations:

$$T^{opt}_{A^{elem}}(d) \approx N_{mesh} \cdot N^d_{loc} \cdot N^d_g,$$

and substituting the values defined for N_{loc} (2.16) and N_g (2.27):

$$T^{opt}_{A^{elem}}(d) \approx N_{mesh} \cdot N^{2d}_{pd},$$

or, for $d = 2$:

$$T^{opt}_{A^{elem}}(2) \approx N_{mesh} \cdot N^4_{pd}.$$

The difference in the implementation costs can be seen on the graph 2.1. The, so-called, "polynomial explosion", can be seen for polynomial degrees $p \geq 6$.

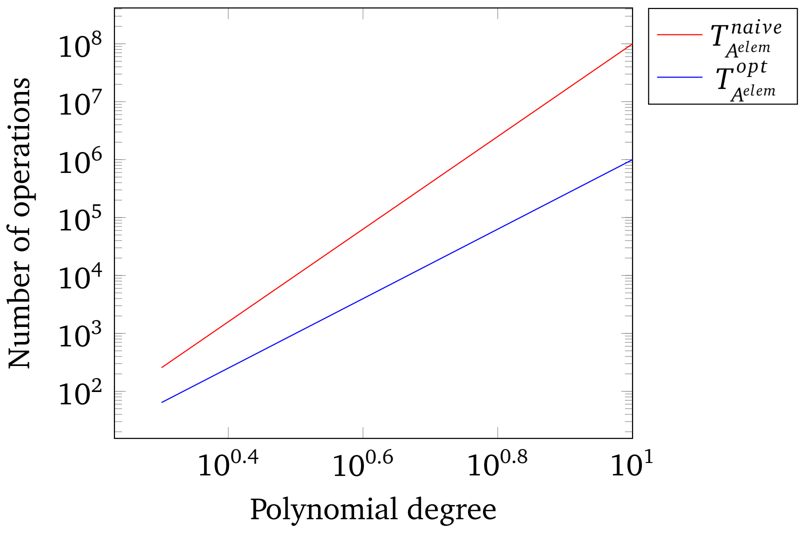


Figure 2.1: Algorithms complexity in 3-d. Number of operations for one mesh element.

To assemble the global matrix A, we use the following algorithm 3.

Algorithm 3 Global matrix A assembly

```
1: for $i = 1 : N_{mesh}$ do
2:     [Compute values of block $i$ of matrix $A^{elem}$]
3:     for $k, l = 1 : N^{loc}$ do
4:         $p \leftarrow \text{map}(i,k)$
5:         $q \leftarrow \text{map}(i,l)$
6:         $a_{p,q} \leftarrow a_{p,q} + a^{elem}_{i,k,l}$
7:     end for
8: end for
```

In reality, we implement local matrix computation and global matrix assembly as one algorithm, so we do not have to repeat i-loop. Global matrix assembly requires map access, and simple summation of local matrix A^{elem} elements with global matrix A elements. Hence, the total number of operations required for $d = 2$ is

$$T_A(d) \approx N_{mesh} \cdot (N_{pd}^4 + N_{pd}^4). \tag{2.29}$$

More generally, for d-dimensions is:

$$T_A(N_{pd}) \approx N_{mesh} \cdot (N_{pd}^{(2d)} + N_{pd}^{(2d)}). \tag{2.30}$$

2.3.2 Edge integral

Similarly as for the volume integral, in the previous subsection, for $a_{vu}(u_h, v)$ (2.5) and $a_{uv}(u_h, v)$ (2.4), using the fact that their composite bilinear form is symmetric, set the matrix $V = (v_{p,q})^{N_{mesh} \cdot N_{loc}}_{N_{mesh} \cdot N_{loc}}$. The matrix element can be calculated:

$$\begin{aligned}
v_{p,q} &= \int_\Gamma [\![\phi_p]\!] \cdot \{\nabla \phi_q\} + \{\nabla \phi_p\} \cdot [\![\phi_q]\!] ds \\
&\quad \text{substituting (2.26)} \\
&= \int_\Gamma [\![\sum_{p=\text{map}(i,k)} \phi^{loc}_{i,k}]\!] \cdot \{\nabla \sum_{q=\text{map}(j,l)} \phi^{loc}_{j,l}\} + \{\nabla \sum_{p=\text{map}(i,k)} \phi^{loc}_{i,k}\} \cdot [\![\sum_{q=\text{map}(j,l)} \phi^{loc}_{j,l}]\!] ds \\
&\quad \text{restricting local basis functions on the elements} \\
&= \sum_{p=\text{map}(i,k)} \sum_{q=\text{map}(j,l)} \int_{\partial K_i \cap \partial K_j} [\![\phi^{loc}_{i,k}]\!] \cdot \{\nabla \phi^{loc}_{j,l}\} + [\![\phi^{loc}_{j,l}]\!] \cdot \{\nabla \phi^{loc}_{i,k}\} ds
\end{aligned}$$

Due to construction and arbitrary choice of the local basis functions, we can concentrate our attention only on the functions on the current element K_i. Neighbouring element's basis functions contributions can be handled with the local basis functions, with support on the edge $e_{i,r}$ which coincide for both elements $K_i, K_j, \quad i = j$:

$$\begin{aligned}
v_{p,q} &= \sum_{p=\text{map}(i,k)} \sum_{q=\text{map}(i,l)} \int_{\partial K_i \cap \partial K_j} [\![\phi^{loc}_{i,k}]\!] \cdot \{\nabla \phi^{loc}_{i,l}\} + [\![\phi^{loc}_{i,l}]\!] \cdot \{\nabla \phi^{loc}_{i,k}\} ds \\
&+ \sum_{p=\text{map}(i,k)} \sum_{r}^{N_e} \sum_{q=\text{map}(\text{neigh}(i,r),l)} \int_{e_{i,r}} [\![\phi^{loc}_{i,k}]\!] \cdot \{\nabla \phi^{loc}_{i_r,l}\} + [\![\phi^{loc}_{i_r,l}]\!] \cdot \{\nabla \phi^{loc}_{i,k}\} ds
\end{aligned}$$

where N_e is a **total number of edges**. The composite index $i_r := \text{neigh}(i,r)$, indicates when the used local basis functions belong to both elements K_i and K_j edge's r, and obtained through the table of neighbours - neigh$(\cdot,\cdot)$.

Due to the nature of the discontinuous Galerkin matrix we use local basis functions with non-zero values only on the edges, so we would have contributions only for elements $K_j = K_i$, e.g. - when edge $e_{i,r}$ of K_i and the edge $e_{j,s}$ of K_j are the same edge. This is due to the fact, that we are only interested in the elements K_j which are neighbours of K_i, so we only consider cases when the edges $e_{i,r}$ and $e_{j,s}$ - are coinciding edges. To collect all contributions only **once**, we use mapping table map$(\cdot,\cdot)$ which require input of index of the current element and number(index) of a local basis function and returns index of a corresponding global basis function.

To separate the contributions on a local level, we introduce the local matrix $V^{elem} = (v^{elem,[s||n]}_{i,k,l,r})$. Upper index s corresponds to self-element contribution and index n corresponds to neighbouring element contribution.

When element's edge is at the boundary of Γ, hence, there are no neighbour,

$$v^{elem,s}_{i,k,l,r} = \int_{e_{i,r}} \phi^{loc}_{i,k} \cdot n_{i,r} \cdot \nabla\phi^{loc}_{i,l} + \phi^{loc}_{i,l} \cdot n_{i,r} \cdot \nabla\phi^{loc}_{i,k} ds,$$

where $n_{i,r}$ is an outward normal on the edge $e_{i,r}$.

For the cases when the element K_j is present. For the cases, when $e_{i,r} = e_{j,s}$ - same edge, and on the same elements, e.g. $i = j$:

$$v^{elem,s}_{i,k,l,r} = \int_{e_{i,r}} \phi^{loc}_{i,k} \cdot n_{i,r} \cdot \frac{1}{2}\nabla\phi^{loc}_{i,l} + \phi^{loc}_{i,l} \cdot n_{i,r} \cdot \frac{1}{2}\nabla\phi^{loc}_{i,k} ds$$

Using agreed mapping, when we have the same edge, but on different elements, e.g. $i \neq j$, although it should be a neighbouring element, hence $i_r = \text{neigh}(i,r)$:

$$v^{elem,n}_{i,k,l,r} = \int_{e_{i,r}} \phi^{loc}_{i,k} \cdot n_{i,r} \frac{1}{2}\nabla\phi^{loc}_{i_r,l} + \phi^{loc}_{i_r,l} \cdot n_{i,r} \cdot \frac{1}{2}\nabla\phi^{loc}_{i,k} ds$$

For simplicity, we keep the integration, so the modified algorithm for the edge-based integration is 4:

Algorithm 4 Edge-based integration local matrix computation. Naive. 2D

```
1: for $i = 1 : N_{mesh}$ do
2:     for $r = 1 : N_e$ do
3:         for $k, l = 1 : N_{loc}$ do
4:             $i_r \leftarrow \text{neigh}(i, r)$
5:             if $i_r = 0$ [no neighbour] then
6:                 $v^{elem,s}_{i,k,l,r} \leftarrow \int_{e_{i,r}} \phi^{loc}_{i,k} \cdot n_{i,r} \cdot \nabla\phi^{loc}_{i,l} + \phi^{loc}_{i,l} \cdot n_{i,r} \cdot \nabla\phi^{loc}_{i,k}$
7:             else
8:                 $v^{elem,s}_{i,k,l,r} \leftarrow \int_{e_{i,r}} \phi^{loc}_{i,k} \cdot n_{i,r} \cdot \frac{1}{2}\nabla\phi^{loc}_{i,l} + \phi^{loc}_{i,l} \cdot n_{i,r} \cdot \frac{1}{2}\nabla\phi^{loc}_{i,k}$
9:                 $v^{elem,n}_{i,k,l,r} \leftarrow \int_{e_{i,r}} \phi^{loc}_{i,k} \cdot n_{i,r}\frac{1}{2}\nabla\phi^{loc}_{i_r,l} + \phi^{loc}_{i_r,l} \cdot n_{i,r} \cdot \frac{1}{2}\nabla\phi^{loc}_{i,k}$
10:            end if
11:        end for
12:    end for
13: end for
```

In the case of edge-based integration, the number of all loop repetitions can be estimated as

$$N_{mesh} \cdot N_e \cdot N_{loc}^2.$$

As in the case of the volume integral, we use hierarchical local basis functions, and should compute at most, in the case when neighbour is present, two values, from both sides of the edge, of the basis functions in each repetition, hence the operation count goes up to

$$N_{mesh} \cdot N_e \cdot N_{loc}^2 \cdot N_{pd}^2.$$

We can use the same Gauss-Legendre data, as we are restricted on one edge, making it one dimensional integration. Number of Gauss-Legendre quadrature points and weights is then $N_g = N_{pd}$. This totals number of operations required for naive matrix computation to:

$$T^{naive}_{V_{elem}} \approx N_{mesh} \cdot N_g \cdot N_e \cdot N_{loc}^2 \times N_{pd}^2,$$

In terms of dimension degree d, and applying the (2.16,2.27):

$$T^{naive}_{V_{elem}}(d) \approx N_{mesh} \cdot N_e \cdot N_{pd}^5,$$

or for the d-dimensional case:

$$\begin{aligned} T^{naive}_{V_{elem}}(N_{pd}) &\approx N_{mesh} \cdot N_e \cdot (N_{pd}^{(d-1)} N_{pd}^{2d} N_{pd}^{d}) \\ &\approx N_{mesh} \cdot N_e \cdot N_{pd}^{(2d+1)}. \end{aligned}$$

We can optimise the naive algorithm 4, by performing the same steps as in case with volume integral. First, we would compute all Gauss-Legendre data into the vectors. Second, we can precompute necessary basis function values into the vectors and reuse the computed values:

Algorithm 5 Edge-based integration local matrix computation. Optimised. 2D

```
1: for $i = 1 : N_{mesh}$ do
2:   for $r = 1 : N_e$ do
3:     $i_r \leftarrow \text{neigh}(i, r)$
4:     $(\omega, t) =$ Gauss($N_{pd}$)
5:     for $s = 1 : N_g$ do
6:       $(t_1^e, t_2^e) = F_{K_i}^e(t_s)$ [ computing the Gauss-Legendre quadrature on a reference edge]
7:       $(x_s, y_s) = F_{K_i}(t_1^e, t_2^e)$ [ computing coordinates on edge $e_r$ from the Gauss-Legendre]
8:       $K(k) \leftarrow \phi_{i,k}^{loc}(x_s, y_s), \forall k = 1 : N_{loc}$
9:       $L(l) \leftarrow \phi_{i,l}^{loc}(x_s, y_s), \forall l = 1 : N_{loc}$
10:      $T(k) \leftarrow n_{i,r} \cdot \nabla\phi_{i,k}^{loc}(x_s, y_s), \forall k = 1 : N_{loc}$
11:      $S(l) \leftarrow n_{i,r} \cdot \nabla\phi_{i,l}^{loc}(x_s, y_s), \forall l = 1 : N_{loc}$
12:      if $i_r > 0$ [neighbour exists!] then
13:        $(t_1^r, t_2^r) = F_{K_{i_r}}^r(t_s)$ [ computing the Gauss-Legendre quadrature on a reference edge]
14:        $(x_s, y_s) = F_{K_{i_r}}(t_1^r, t_2^r)$ [ computing coordinates on edge $e_{i_r}$ from the Gauss-Legendre]
15:        $L_r(l) \leftarrow \phi_{i_r,l}^{loc}(x_s, y_s), \forall k = 1 : N_{loc}$
16:        $S_r(l) \leftarrow \frac{1}{2} n_{i,r} \cdot \nabla\phi_{i_r,l}^{loc}(x_s, y_s), \forall l = 1 : N_{loc}$
17:        $v_{i,1:N_{loc},1:N_{loc},r}^{elem,s} \leftarrow v_{i,1:N_{loc},1:N_{loc},r}^{elem,s} + \omega_s K(1 : N_{loc}) \cdot \frac{1}{2} S(1 : N_{loc}) + \omega_s L(1 : N_{loc}) \cdot \frac{1}{2} T(1 : N_{loc})$
18:        $v_{i,1:N_{loc},1:N_{loc},r}^{elem,n} \leftarrow v_{i,1:N_{loc},1:N_{loc},r}^{elem,n} + \omega_s K(1 : N_{loc}) \cdot S_r(1 : N_{loc}) + \omega_s L_r(1 : N_{loc}) \cdot T(1 : N_{loc})$
19:      else
20:        $v_{i,1:N_{loc},1:N_{loc},r}^{elem,s} \leftarrow v_{i,1:N_{loc},1:N_{loc},r}^{elem,s} + \omega_s K(1 : N_{loc}) \cdot S(1 : N_{loc}) + \omega_s L(1 : N_{loc}) \cdot T(1 : N_{loc})$
21:      end if
22:    end for
23:  end for
24: end for
```

This considerably reduces the amount of computations to

$$\begin{aligned} T_{V_{elem}}^{opt}(d) &\approx N_{mesh} \cdot N_e \cdot (N_g^{(d-1)} N_{loc}^d N_{pd}) \\ &\approx N_{mesh} \cdot N_e \cdot (N_{pd}^{(d-1)} N_{pd}^{2d} N_{pd}) \\ &\approx N_{mesh} \cdot N_e \cdot N_{pd}^{(2d)}. \end{aligned}$$

And for $d = 2$:

$$T_{V_{elem}}^{opt}(d) \approx N_{mesh} \times N_e \times N_{pd}^4.$$

And the global matrix V assembly shown in Algorithm 6, differs from the one for the matrix A in (3), with the extra term, for the self-contribution.

Algorithm 6 Edge-based integration global matrix assembly

```
1:  for $i = 1 : N_{mesh}$ do
2:      for $r = 1 : 4$ do
3:          for $k, l = 1 : N_{loc}$ do
4:              $p \leftarrow \mathrm{map}(i, k)$
5:              $q \leftarrow \mathrm{map}(i, l)$
6:              $i_r \leftarrow \mathrm{neigh}(i, k)$
7:              $q_r \leftarrow \mathrm{map}(i_r, l)$
8:              $v_{p,q} \leftarrow v_{p,q} + v^{elem,s}_{i,k,l,r}$
9:              if $i_r > 0$ [neighbour exists] then
10:                 $v_{p,q_r} \leftarrow v_{p,q} + v^{elem,n}_{i,k,l,r}$
11:             end if
12:         end for
13:     end for
14: end for
```

(**S.1**) For the edge-based integration, we have to take into account the fact that we deal with two cases: internal edge(face) - interface and neighbour's faces contribute to the local element calculation, boundary edge(face) - only self contribution, no neighbour is present. However we have to take into account, that for different choice of the basis functions we would have different number of contributions from each element to our local matrices.

I. General choice. For general choice of the basis functions, when we do not have any special properties and restrictions for the basis functions on a reference elements, we would have the following set-up. The degree of freedom of the global matrix V is $DoF(V) = N_{mesh} \cdot N^d_{pd}$, with N_{mesh} local elements contributing to the global matrix V. Analysing all basis functions' interactions with two abstract neighbouring elements K_I and K_{II}, we get four possible sets:

$\phi_{I,i} \cdot \phi_{II,j}$, $\quad i, j = 1 : N^d_{pd}$ which results in $N^d_{pd} \cdot N^d_{pd}$ non-zero elements in local Galerkin matrix

$\phi_{I,i} \cdot \nabla\phi_{II,j}$, $\quad i, j = 1 : N^d_{pd}$ which results in $N^d_{pd} \cdot N^d_{pd}$ non-zero elements in local Galerkin matrix

$\nabla\phi_{I,i} \cdot \phi_{II,j}$, $\quad i, j = 1 : N^d_{pd}$ which results in $N^d_{pd} \cdot N^d_{pd}$ non-zero elements in local Galerkin matrix

$\nabla\phi_{I,i} \cdot \nabla\phi_{II,j}$, $\quad i, j = 1 : N^d_{pd}$ which results in $N^d_{pd} \cdot N^d_{pd}$ non-zero elements in local Galerkin matrix

Sets two and three are symmetric and can be handled with the same piece of code. Approximation of the corresponding integrals numerically using Gauss-Legendre method requires only N_{pd} nodes per one edge, hence, the total number of operations required for the calculation and assembly of the local blocks is:

$$T_{V^{elem}}(d) \approx N_{pd} \cdot N^{2d}_{pd} \tag{2.31}$$

Then the global V calculation and assembly time is:

$$T_V(d) \approx N_{mesh} \cdot N_{pd}^{2d+1} \tag{2.32}$$

In the case of general basis functions, matrix V is nor dense, nor sparse, but has a block structure, with N_{pd}^{2d} non-zero elements in each block.

For specific choice of the basis functions, we would concentrate on two cases:

II. Antiderivative of Legendre Polynomials Antiderivative of Legendre Polynomials (2.20). As before, we have the same degree of freedom and the same number of local elements. Except that in this particular case, we can exploit the fact that basis functions vanish outside of their domain (edge), which would give us only self-functions' contributions per edge (2.2.2.2). However gradient of the chosen basis functions does not have this property. Again we would have four possible sets of interactions:

$\phi_{I,i} \cdot \phi_{II,j}$, $i, j = 1 : N_{pd}^d$ which results in N_{pd}^{d-1} non-zero elements in local Galerkin matrix

$\phi_{I,i} \cdot \nabla\phi_{II,j}$, $i, j = 1 : N_{pd}^d$ which results in $N_{pd}^{d-1} \cdot N_{pd}^d$ non-zero elements in local Galerkin matrix

$\nabla\phi_{I,i} \cdot \phi_{II,j}$, $i, j = 1 : N_{pd}^d$ which results in $N_{pd}^d \cdot N_{pd}^{d-1}$ non-zero elements in local Galerkin matrix

$\nabla\phi_{I,i} \cdot \nabla\phi_{II,j}$, $i, j = 1 : N_{pd}^d$ which results in N_{pd}^{2d} non-zero elements in local Galerkin matrix

Sets two and three are symmetric and can be handled with the same piece of code. As before, we would need N_{pd} Gauss-Legendre quadrature nodes per edge, to numerically approximate the integrals, resulting in, at most:

$$T_{V^{elem}}(d) \approx N_{pd} \cdot N_{pd}^{2d-1} \tag{2.33}$$

operations per local element, hence the global V calculation and assembly time is:

$$T_V(d) \approx N_{mesh} \cdot N_{pd}^{2d} \tag{2.34}$$

The matrix would have blocks with at most N_{pd}^{2d} non-zero elements in each block, with equal number of blocks with N_{pd}^d non-zero elements and two times number of blocks with $N_{pd}^d \cdot N_{pd}^{d+1}$ non-zero elements.

III The choice of the special case of C1-polynomials, would give more vanishing on all but one edges basis functions, and using the special gradient properties (2.23) would result vanishing gradients (2.2.2.2):

$\phi_{I,i} \cdot \phi_{II,j}$, $i, j = 1 : N_{pd}^d$ which results in $N_{pd}^{d-1} \cdot N_{pd}^{d-1}$ non-zero elements in local Galerkin matrix

$\phi_{I,i} \cdot \nabla\phi_{II,j}$, $i, j = 1 : N_{pd}^d$ which results in $N_{pd}^{d-1} \cdot N_{pd}^{d-1}$ non-zero elements in local Galerkin matrix

$\nabla\phi_{I,i}\cdot\phi_{II,j}$, $i,j=1:N_{pd}^d$ which results in $N_{pd}^{d-1}{\cdot}N_{pd}^{d-1}$ non-zero elements in local Galerkin matrix

$\nabla\phi_{I,i}\cdot\nabla\phi_{II,j}$, $i,j=1:N_{pd}^d$ which results in $N_{pd}^{d-1}{\cdot}N_{pd}^{d-1}$ non-zero elements in local Galerkin matrix

With N_{pd} Gauss-Legendre quadrature nodes per edge, we end up with, at most:

$$T_{V^{elem}}(d)\approx N_{pd}\cdot N_{pd}^{2(d-1)} \tag{2.35}$$

operations per local element, hence the global V calculation and assembly time is:

$$T_V(d)\approx N_{mesh}\cdot N_{pd}^{2d-1} \tag{2.36}$$

The matrix' blocks would have at most N_{pd}^2 non-zero elements.

Using the same principle for $a_{bu}(u_h,v)$ and $a_{ub}(u_h,v)$ we obtain matrix $B=(b_{p,q})$, noting that it is defined only for $\Gamma^0:=\Gamma\setminus\partial\Gamma$:

$$\begin{aligned}
b_{p,q} &= \int_{\Gamma^0}(\beta\cdot[\![\phi_p]\!])[\![\nabla\phi_q]\!]+[\![\nabla\phi_p]\!](\beta\cdot[\![\phi_q]\!])dx\\
&= \int_{\Gamma^0}(\beta\cdot[\![\sum_{p=\text{map}(i,k)}\phi_{i,k}^{loc}]\!])[\![\nabla\sum_{q=\text{map}(j,l)}\phi_{j,l}^{loc}]\!]\\
&+ [\![\nabla\sum_{p=\text{map}(i,k)}\phi_{i,k}^{loc}]\!](\beta\cdot[\![\sum_{q=\text{map}(j,l)}\phi_{j,l}^{loc}]\!])dx\\
&= \sum_{p=\text{map}(i,k)}\sum_{q=\text{map}(j,l)}\int_{\Gamma^0}(\beta\cdot[\![\phi_{i,k}^{loc}]\!])[\![\nabla\phi_{j,l}^{loc}]\!]+[\![\nabla\phi_{i,k}^{loc}]\!](\beta\cdot[\![\phi_{j,l}^{loc}]\!])dx\\
&\quad\text{introducing edges } e_{i,r} \text{ and } e_{i,s}\text{:}\\
&= \sum_{p=\text{map}(i,k)}\sum_{q=\text{map}(i,l)}\int_{\partial K_i\cap\partial K_i}(\beta\cdot[\![\phi_{i,k}^{loc}]\!])[\![\nabla\phi_{i,l}^{loc}]\!]+[\![\nabla\phi_{i,k}^{loc}]\!](\beta\cdot[\![\phi_{i,l}^{loc}]\!])ds\\
&+ \sum_{p=\text{map}(i,k)}\sum_{r}^{N}\sum_{q=\text{map}(\text{neigh}(i,r),l)}\int_{e_{i,r}}(\beta\cdot[\![\phi_{i,k}^{loc}]\!])[\![\nabla\phi_{i_r,l}^{loc}]\!]+[\![\nabla\phi_{i,k}^{loc}]\!](\beta\cdot[\![\phi_{i_r,l}^{loc}]\!])ds,
\end{aligned}$$

where, like in previous case, $i_r:=\text{neigh}(i,r)$.

We compute the local element matrix $B^{elem}=(b_{i,k,l,r}^{elem,[s||n]})$ in a similar way to the matrix V. Inner product form for this formulation is symmetric. Using the same scheme, as for V^{elem}, noting that B^{elem} is computed on $\Gamma^0:=\Gamma\setminus\partial\Gamma$, hence not defined on the boundary. For $b_{i,k,l,r}^{elem,s}$, we get: when $e_{i,r}=e_{j,s}$ - same edge, and on the same elements, e.g. $i=j$:

$$b_{i,k,l,r}^{elem,s}=\int_{e_{i,r}}(\beta\phi_{i,k}^{loc}\cdot n_{i,r})\cdot(\nabla\phi_{i,l}^{loc}\cdot n_{i,r})+(\beta\phi_{i,l}^{loc}\cdot n_{i,r})\cdot(\nabla\phi_{i,k}^{loc}\cdot n_{i,r})ds$$

when $e_{i,r}=e_{j,s}$ - same edge, but on different elements, e.g. $i\neq j$:

$$b_{i,k,l,r}^{elem,n}=\int_{e_{i,r}}\beta\phi_{i,k}^{loc}\cdot n_{i,r}\cdot\nabla\phi_{i_r,l}^{loc}\cdot n_{i,r}+\beta\phi_{i_r,l}^{loc}\cdot n_{i,r}\cdot\nabla\phi_{i,k}^{loc}\cdot n_{i,r}ds$$

And a modified assembly algorithm for the matrix B is:

Algorithm 7 Local matrix B^{loc} computation

```
1:  for $i = 1 : N_{mesh}$ do
2:      for $r = 1 : N_e$ do
3:          for $k, l = 1 : N_{loc}$ do
4:              $i_r \leftarrow \mathrm{neigh}(i, k)$
5:              if $i_r > 0$ [neighbour exists] then
6:                  $b^{elem,s}_{i,k,l,r} \leftarrow b^{elem}_{i,k,l,r} + \int_{e_{i,r}} (\beta \phi^{loc}_{i,k} \cdot n_{i,r}) \cdot (\nabla \phi^{loc}_{i,l} \cdot n_{i,r}) + (\beta \phi^{loc}_{i,l} \cdot n_{i,r}) \cdot (\nabla \phi^{loc}_{i,k} \cdot n_{i,r})$
7:                  $b^{elem,n}_{i,k,l,r} \leftarrow b^{elem}_{i,k,l,r} + \int_{e_{i,r}} \beta \phi^{loc}_{i,k} \cdot n_{i,r} \cdot \nabla \phi^{loc}_{i_r,l} \cdot n_{i,r} + \beta \phi^{loc}_{i_r,l} \cdot n_{i,r} \cdot \nabla \phi^{loc}_{i,k} \cdot n_{i,r}$
8:              end if
9:          end for
10:     end for
11: end for
```

where, we check for the neighbour's existence, otherwise, the edge of the interest is not shared with other elements, hence, is on the boundary.

Algorithm 8 Matrix B assembly

```
1:  for $i = 1 : N_{mesh}$ do
2:      for $r = 1 : N_e$ do
3:          for $k, l = 1 : N_{loc}$ do
4:              $p \leftarrow \mathrm{map}(i, k)$
5:              $q \leftarrow \mathrm{map}(i, l)$
6:              $i_r \leftarrow \mathrm{neigh}(i, k)$
7:              $q_r \leftarrow \mathrm{map}(i_r, l)$
8:              if $i_r > 0$ [neighbour exists] then
9:                  $b_{p,q} \leftarrow b_{p,q} + b^{elem,s}_{i,k,l,r}$
10:                 $b_{p,q_r} \leftarrow b_{p,q} + b^{elem,n}_{i,k,l,r}$
11:             end if
12:         end for
13:     end for
14: end for
```

This matrix' assembly time estimation is similar to that of matrix V, as here, we consider all edges (faces) contributions.

At last, for the bilinear form $a_s(u_h, v)$ we get matrix $S = (s_{p,q})$

$$\begin{aligned}
s_{p,q} &= \int_\Gamma \alpha[\![\phi_p]\!]\cdot[\![\phi_q]\!]dx \\
&= \int_\Gamma \alpha[\![\sum_{p=\text{map}(i,k)}(\phi_{i,k}^{loc})]\!]\cdot[\![\sum_{q=\text{map}(j,l)}(\phi_{j,l}^{loc})]\!]dx \\
&= \sum_{p=\text{map}(i,k)}\sum_{q=\text{map}(j,l)}\int_{\partial K_i\cap\partial K_j}\alpha[\![\phi_{i,k}^{loc}]\!]\cdot[\![\phi_{j,l}^{loc}]\!]dx \\
&\ \text{introducing edge } e_{i,r}\text{:} \\
&= \sum_{p=\text{map}(i,k)}\sum_{q=\text{map}(i,l)}\int_{\partial K_i}\alpha[\![\phi_{i,k}^{loc}]\!]\cdot[\![\phi_{j,l}^{loc}]\!]ds \\
&+ \sum_{p=\text{map}(i,k)}\sum_{r=1}^{N_e}\sum_{q=\text{map}(\text{neigh}(i,r),l)}\int_{e_{i,r}}\alpha[\![\phi_{i,k}^{loc}]\!]\cdot[\![\phi_{i_r,l}^{loc}]\!]ds
\end{aligned}$$

Introducing the local element matrix $S^{elem} = (s_{i,k,l,r}^{elem,[s||n]})$, similarly to V^{elem} and B^{elem}. For the case, when $e_{i,r}$ is a boundary edge:

$$s_{i,k,l,r}^{elem,s} = \int_{e_{i,r}} (\alpha\phi_{i,k}^{loc}\cdot n_{i,r})\cdot(\phi_{i,k}^{loc}\cdot n_{i,r})ds.$$

Exploiting the fact that outward normals should be perpendicular to each other:

$$s_{i,k,l,r}^{elem,s} = \int_{e_{i,r}} \alpha\phi_{i,k}^{loc}\cdot\phi_{i,k}^{loc}ds.$$

And for the case $e_{i,r} = e_{j,s}$, when $i = j$

$$s_{i,k,l,r}^{elem,s} = \int_{e_{i,r}} \alpha\phi_{i,k}^{loc}\cdot\phi_{i,l}^{loc}ds,$$

when $i \neq j$, employing orthogonality between outwards normals:

$$s_{i,k,l,r}^{elem,n} = -\int_{e_{i,r}} \alpha\phi_{i,k}^{loc}\cdot\phi_{i_r,l}^{loc}ds$$

Assembly algorithm is the same as for matrix V. The number of operations required is similar, to the one of matrix V.

2.3.3 Lifting operators' matrix formulation

Now, to represent lifting operators in their matrix form, we could agree to use $c^{[\cdot]}$ coefficient vector. For simplicity, we are not going to define any dimensions of the vector. Analyse the given definition of r-lifting operator:

$$\int_\Omega r(\varphi)\cdot\tau dx = -\int_\Gamma \varphi\cdot\{\tau\}ds \quad r : [L^2(\Gamma)]^2 \to [W_{hp}]^d,$$

where $\tau \in [W_{hp}]^d$.

Choosing $\psi_p \in [W_{hp}]^d$ as a translation of τ on an element K_p, $c_p^r \cdot \psi_p$ as a translation of $r(\varphi)$, and $c_p^\varphi \cdot \psi_p$ as a projection of φ:

$$r([\![u]\!]) = \sum_p^{N_{loc}} c_p^{ru} \psi_p$$

finally,

$$\begin{aligned} \int_\Omega r([\![u]\!]) \tau dx &= -\int_\Gamma [\![u]\!] \cdot \{\tau\} ds & (2.37) \\ \sum_p^{N_{loc}} c_p^{ru} \int_\Omega \psi_p \cdot \psi_q dx &= -\int_\Gamma [\![u]\!] \cdot \{\psi_q\} ds, \quad q = 1, \ldots, N_{loc}, & (2.38) \end{aligned}$$

with N_{loc} being a number of Raviart-Thomas elements.

Here, the following element on the left hand side can be isolated as a Mass matrix:

$$M = (m_{p,q}); \quad m_{p,q} = \sum_p^{N_{loc}} \int_\Gamma \psi_q \cdot \psi_p dx, \quad q = 1, \ldots, N_{loc}.$$

Recalling the definition

$$u = \sum_p^{N_{pd}} c_p^u \phi_p, \quad \phi \in V_{hp}$$

and substituting into the right hand side of the (2.38):

$$\begin{aligned} & - \int_\Gamma [\![\sum_p^{N_{pd}} c_p^u \phi_p]\!] \cdot \{\psi_q\} ds, \quad q = 1, \ldots, N_{loc} \\ = & - \sum_p^{N_{pd}} c_p^u \int_\Gamma [\![\phi_p]\!] \cdot \{\psi_q\} ds, \quad q = 1, \ldots, N_{loc}. \qquad (2.39) \end{aligned}$$

Again, now in (2.39), we can separate the following:

$$\begin{aligned} m_{p,q}^\Gamma &= \int_\Gamma [\![\phi_p]\!] \cdot \{\psi_q\} ds \\ &= \int_\Gamma [\![\sum_{p=\mathrm{map}(i,k)} \phi_{i,k}^{loc}]\!] \cdot \{ \sum_{q=\mathrm{map}(j,l)} \psi_{j,l}^{loc} \} ds \\ &= \sum_{p=\mathrm{map}(i,k)} \sum_{q=\mathrm{map}(j,l)} \int_{\partial K_i \cap \partial K_j} [\![\phi_{i,k}^{loc}]\!] \cdot \{\psi_{j,l}^{loc}\} ds. \end{aligned}$$

Due to construction and arbitrary choice of the local basis functions, we can concentrate our attention only on the functions on the current element K_i. Neighbouring element's basis functions interactions can be handled with the local basis functions, with support on the edge $e_{i,r}$ which coincide for both elements K_i, K_j. We also introduce a special index $i_r :=$ neigh(i,r), which indicates when the used local basis functions belong to both elements

K_i and K_j edge's r, and the index is obtained through the table of neighbours - neigh$(\cdot,\cdot)$. Substituting:

$$\begin{aligned} m^{\Gamma}_{p,q} &= \sum_{p=\text{map}(i,k)} \sum_{q=\text{map}(i,l)} \int_{\partial K_i \cap \partial K_i} [\![\phi^{loc}_{i,k}]\!] \cdot \{\psi^{loc}_{i,l}\} ds \\ &+ \sum_{p=\text{map}(i,k)} \sum_{r=1}^{N^e} \sum_{q=\text{map}(i_r,l)} \int_{e_{i,r}} [\![\phi^{loc}_{i,k}]\!] \cdot \{\psi^{loc}_{i_r,l}\} ds. \end{aligned}$$

Expanding the jump and average notation. Due to the nature of the discontinuous Galerkin matrix we use local basis functions with non-zero values only on the edges, so we would have interaction only when edge $e_{i,r}$ of K_i and the edge $e_{j,s}$ of K_j are the same edge. Due to this fact, we are only interested in the elements K_j which are neighbours of K_i, so we only consider cases when the edges $e_{i,r}$ and $e_{j,s}$ - are coinciding edges. To collect all contributions only **once**, we use mapping table map$(\cdot,\cdot)$ which require input of index of the current element and number(index) of a local basis function and returns index of a corresponding global basis function. Introducing the local matrix $M^{\Gamma,elem} = (m^{\Gamma,elem,[s||n]}_{i,k,l,r})$. Upper index s corresponds to self-element contribution and index n corresponds to neighbouring element contribution.

When element's edge is at the boundary of Γ, hence, there are no neighbour,

$$m^{\Gamma,elem,s}_{i,k,l,r} = \int_{e_{i,r}} \phi^{loc}_{i,k}\left(n_{i,r} \cdot \psi^{loc}_{i,l}\right) ds,$$

where $n_{i,r}$ is an outward normal on the edge $e_{i,r}$.

For the cases when the element K_j is present. For the cases, when $e_{i,r} = e_{j,s}$ - same edge, and on the same elements, e.g. $i = j$:

$$m^{\Gamma,elem,s}_{i,k,l,r} = \int_{e_{i,r}} \phi^{loc}_{i,k}\left(n_{i,r} \cdot \frac{1}{2} \psi^{loc}_{i,l}\right) ds$$

Using agreed mapping, when we have the same edge, but on different elements, e.g. $i \neq j$. Due to the restrictions on the local basis functions, we should ensure, that the right neighbouring element is employed, hence $j \equiv i_r :=$ neigh(i,r):

$$m^{\Gamma,elem,n}_{i,k,l,r} = \int_{e_{i,r}} \phi^{loc}_{i,k}\left(n_{i,r} \frac{1}{2} \psi^{loc}_{i_r,l}\right) ds.$$

Matrix M^{Γ} can be assembled, using the non-modified algorithm (6). We should pay attention to the elementary matrix' special index q_r, which ensures, that the contribution from element K_{i_r}, neighbouring element K_i through the edge e_r, is added to the proper position of the matrix M^{Γ}.

After computing and assembling all the defined above matrices M and M^{Γ}, we can substitute them into the (2.38) and get the matrix-vector representation:

$$\begin{aligned}
(c^{ru})^T \cdot M &= -(c^u)^T \cdot M^\Gamma \\
M^T \cdot c^{ru} &= -(M^\Gamma)^T \cdot c^u \\
&\quad \text{using the fact that the } M \text{ is symmetric, omit transposition} \\
c^{ru} &= -(M)^{-1} \cdot (M^\Gamma)^T \cdot c^u. \qquad (2.40)
\end{aligned}$$

For the l-lifting operator, choosing the same setting, as for the r-lifting operator, $\psi \in [W_{hp}]^d$, $\phi \in V_{hp}$, and $\beta \in \mathbb{R}^d$ being a constant on each edge, we can start the analysis of the operator:

$$\int_\Omega l(\varphi) \cdot \tau dx = -\int_{\Gamma^0} \varphi [\![\tau]\!] ds \quad l : L^2(\Gamma^0) \to [V_{hp}]^d.$$

Setting

$$l(\beta [\![u]\!]) = \sum_p^{N_{loc}} c_p^{lu} (\beta \cdot \psi_p),$$

we get

$$\begin{aligned}
\int_\Omega \sum_p^{N_{loc}} c_p^{lu} (\psi_p \cdot \psi_q) dx &= -\int_{\Gamma^0} (\beta \cdot [\![u]\!]) [\![\psi_q]\!] ds, \quad q = 1, \dots, N_{loc} \\
\sum_p^{N_{loc}} c_p^{lu} \int_\Omega \psi_p \cdot \psi_q dx &= -\int_{\Gamma^0} (\beta \cdot [\![\sum_p^{N_2} c_p^u \cdot \phi_p]\!]) [\![\psi_q]\!] ds, \quad q = 1, \dots, N_{loc} \\
\sum_p^{N_{loc}} c_p^{lu} \int_\Omega \psi_p \cdot \psi_q dx &= -\sum_p^{N_2} c_p^u \int_{\Gamma^0} (\beta \cdot [\![\phi_p]\!]) [\![\psi_q]\!] ds, \quad q = 1, \dots, N^{RT} \quad (2.41)
\end{aligned}$$

Again, as in (2.38), we can separate the Mass matrix M, and introduce the matrix M^{Γ^0} with the entries:

$$\begin{aligned}
M_{p,q}^{\Gamma^0} &= \int_{\Gamma^0} (\beta \cdot [\![\phi_p]\!]) [\![\psi_q]\!] ds \\
&= \int_{\Gamma^0} (\beta \cdot [\![\sum_{p=\mathrm{map}(i,k)} \phi_{i,k}^{loc}]\!]) [\![\sum_{q=\mathrm{map}(j,l)} \psi_{j,l}^{loc}]\!] ds \\
&= \sum_{p=\mathrm{map}(i,k)} \sum_{q=\mathrm{map}(j,l)} \int_{\partial K_i \cap \partial K_j} (\beta \cdot [\![\phi_{i,k}^{loc}]\!]) [\![\psi_{j,l}^{loc}]\!] ds.
\end{aligned}$$

Expanding, using the fact, that the local basis functions are only non-zero on their edge:

$$\begin{aligned}
m_{p,q}^{\Gamma^0} &= \sum_{p=\mathrm{map}(i,k)} \sum_{q=\mathrm{map}(i,l)} \int_{\partial K_i \cap \partial K_i} (\beta \cdot [\![\phi_{i,k}^{loc}]\!]) [\![\psi_{i,l}^{loc}]\!] ds \\
&+ \sum_{p=\mathrm{map}(i,k)} \sum_{r=1}^{N^e} \sum_{q=\mathrm{map}(i_r,l)} \int_{e_{i,r}} (\beta \cdot [\![\phi_{i,k}^{loc}]\!]) [\![\psi_{i,l}^{loc}]\!] ds.
\end{aligned}$$

Similarly, to the $M^{\Gamma,elem}$ definition, we can split the $M^{\Gamma^0,elem}$ computation into the self-interaction and neighbouring interaction. We should note, that Γ^0 does not include any boundary edges, hence there would be no contributions if element K_i has no neighbour. The self-interaction case then is $e_{i,r} = e_{j,s}$ are the same edge, and on the same elements, e.g. $i = j$:

$$m^{\Gamma^0,elem,s}_{i,k,l,r} = \int_{e_{i,r}} \Big((\beta \phi^{loc}_{i,k}) \cdot n_{i,r} \Big) (\psi^{loc}_{i,l} \cdot (-n_{i,r})) ds.$$

And for the same edge, but on different elements, e.g. $i \neq j$, $e_{i,r} = e_{j,s}$, we set $j \equiv i_r :=$ neigh(i, r):

$$m^{\Gamma^0,elem,n}_{i,k,l,r} = \int_{e_{i,r}} \Big((\beta \phi^{loc}_{i,k}) \cdot n_{i,r} \Big) (\psi^{loc}_{i_r,l} \cdot (-n_{i,r})) ds.$$

The matrix M^{Γ^0} can be assembled, using the non-modified algorithm (8). As in previous case, extra care should be given to the special index i_r.

Writing (2.41) in matrix notation:

$$\begin{aligned} (c^{lu})^T \cdot M &= -(c^u)^T \cdot M^{\Gamma^0} \\ M^T \cdot c^{lu} &= -(M^{\Gamma^0})^T \cdot c^u \\ c^{lu} &= -(M)^{-1} \cdot (M^{\Gamma^0})^T \cdot c^u. \end{aligned} \tag{2.42}$$

Auxiliary matrices defined to handle the l- and r- lifting operators:

$$M = (m_{p,q}) \in \mathbb{R}^{N^{RT} \times N^{RT}}; \quad m_{p,q} = (\psi_p, \psi_q)_\Gamma \tag{2.43}$$

$$M^\Gamma = (m^\Gamma_{p,q}) \in \mathbb{R}^{N_{pd} \times N^{RT}}; \quad m^\Gamma_{p,q} = ([\![\phi_p]\!], \{\psi_q\})_\Gamma \tag{2.44}$$

$$M^{\Gamma^0} = (m^{\Gamma^0}_{p,q}) \in \mathbb{R}^{N_{pd} \times N^{RT}}; \quad m^{\Gamma^0}_{p,q} = ([\![\phi_p]\!], [\![\psi_q]\!])_{\Gamma^0} \tag{2.45}$$

Matrix M is a classical **DG**-mass matrix, which is constructed using vector-valued Raviart-Thomas functions. M is a positive definite invertible matrix. Matrices M^Γ and M^{Γ^0} have a scalar-valued basis functions on the left and vector-valued basis functions on the right side.

Last step in matrix system form definition is to obtain a proper setup for lifting operators' interactions.

$a_{rr}(\cdot, \cdot)$-case:

$$\int_\Omega r([\![u]\!]) \cdot r([\![v]\!]) dx$$

could be rewritten as a global sum form using the projection on the elements and coefficient vectors:

$$\int_\Omega \Big((\sum_p^{N_{loc}} c^{ru}_i \psi_p) \cdot (\sum_q^{N_{loc}} c^{rv}_q \psi_q) \Big) dx.$$

This is equivalent to the following vector-matrix relation

$$(c^{ru})^T \cdot M \cdot c^{rv}$$

with previously defined coefficient-vectors (2.40), we have

$$\begin{aligned} c^{ru} &= -M^{-1} \cdot (M^{\Gamma})^T \cdot c^u \\ c^{rv} &= -M^{-1} \cdot (M^{\Gamma})^T \cdot c^v. \end{aligned}$$

Notation c^{ru} reads as - coefficient-vector c of r-lifting operator applied to u. Also, vector c^u and vector c^v are the translation of an the functions $u_h, v \in V_{hp}$, what gives us right to set it up to the values of the basis function on elements of our discretisation. The resulting contribution of **rr** (2.9) to primal bilinear formulation $B_h(u_h, v)$ is

$$(c^u)^T \cdot M^{\Gamma} \cdot M^{-1} \cdot (M^{\Gamma})^T \cdot c^v. \tag{2.46}$$

$a_{rl}(\cdot,\cdot)$-case:

Similarly, $\int_\Omega r([\![u]\!]) \cdot l([\![v]\!]) dx$ can be expanded, as

$$\int_\Omega \Big((\sum_i^N c_i^{ru} \cdot \psi_i) \cdot (\sum_j^N c_j^{lv} \cdot \psi_j) \Big),$$

resulting in

$$(c^{ru})^T \cdot M \cdot c^{lv},$$

with contribution

$$(c^u)^T \cdot M^{\Gamma} \cdot M^{-1} \cdot (M^{\Gamma^0})^T \cdot c^v.$$

$a_{lr}(\cdot,\cdot)$-case:

For $\int_{\Omega_i} l([\![u]\!]) \cdot r([\![v]\!]) dx$ resulting contribution

$$(c^v)^T \cdot M^{\Gamma^0} \cdot M^{-1} \cdot (M^{\Gamma})^T \cdot c^u$$

equals to the transposed contribution of **rl**-case (2.10).

$a_{ll}(\cdot,\cdot)$-case:

$\int_{\Omega_i} l([\![u]\!]) \cdot l([\![v]\!]) dx$ using the same technique could be represented as

$$(c^u)^T \cdot M^{\Gamma^0} \cdot M^{-1} (\cdot M^{\Gamma^0})^T \cdot c^v.$$

Now, the next problem arises, the number of operations required for computation of such system, can be obtained in the same way as the matrices A, B, V. First, in the worst-case, we would need at most N_{pd}^5 operations to compute each edge-based local matrix. We can also multiply matrices on a local level, reducing the total number of operations for matrix-matrix-matrix multiplication to $2N_{pd}^{2d}$, as we have two $N_{pd}^d \times N_{pd}^d$ sparse matrices. For the latter, we have to add computation of the inverse of the mass-matrix M, which is N_{pd}^{3d}, using the Gauss-Jordan algorithm. Finally, we have to repeat it for every element of the mesh, to get the total:

$$T_{rr,rl,lr,ll}(N_{pd}) \approx N_{mesh}(3N_{pd}^{2d+1} + 3N_{pd}^{3d}). \tag{2.47}$$

The matrix multiplication on a local level, can be performed using the fact, that we deal with block-sparse matrices. Analyse on an example of first contribution (2.46)

$$M^{rr} = M^{\Gamma} \cdot M^{-1} \cdot (M^{\Gamma})^T$$

First, we introduce block notation $M^{\Gamma}_{I,K}$, which translates as *all interactions on element K_i for all $k = 1, \dots, N_{loc}$ basis functions belonging (non-zero) to (on) that element*. Obviously, for mass-matrix, blocks would described as $M^{-1}_{K,L}$ for all cross-element interactions on $[W_{hp}]^d$. This can be described more precise with sum notation [46]:

$$M^{\Gamma}_{I,K} = \sum_k M^{\Gamma}_{i,k} \tag{2.48}$$

Or index notation:

$$M^{\Gamma}_{I,K} = M^{\Gamma}_{i,(1,\dots,k)}$$

Now set:

$$M^{rr}_{I,J} = \sum_K \sum_L M^{\Gamma}_{I,K} M^{-1}_{K,L} (M^{\Gamma}_{J,L})^T, \quad I, J = 1, \dots, N$$

Here, we should note, that mass-matrix is symmetric, and would only have diagonal blocks with non-zero values. This collapses L sum:

$$M^{rr}_{I,J} = \sum_K M^{\Gamma}_{I,K} M^{-1}_{K,K} (M^{\Gamma}_{J,K})^T, \quad I, J = 1, \dots, N$$

In this setting we only have two cases, for $M^{\Gamma}_{I,K}$ (and for $M^{\Gamma}_{J,L}$) is not zero, only when $I = K \quad$ or $\quad K = I_r$, which can be generalised as follows:

$$M^{\Gamma}_{I,K} = M^{\Gamma}_{I_s}, \quad s = 0, \dots, N_e$$

where, $M^{\Gamma}_{I_s} = M^{\Gamma}_{I,I_s} \quad s = 1, \dots, N_e$ is interactions through edges (faces) of elements in block I, excluding the self-interactions when $s = 0$, which are described with and $M^{\Gamma}_{I_0} = M^{\Gamma}_{I,I}$. Finally this can be reduced to:

$$M^{rr}_{I,J} = \sum_r M^{\Gamma}_{I,I_r} M^{-1}_{I_r,I_r} M^{\Gamma}_{I_r,J}, \quad I, J = 1, \dots, N \tag{2.49}$$

This can be rewritten as a full element-wise sum:

$$M^{rr} = \sum_i \sum_{r,s} M^{\Gamma}_{i_r,i} M^{-1}_{i,i} M^{\Gamma}_{i,i_s}, \quad I, J = 1, \dots, N \tag{2.50}$$

For this, we need to substitute the (2.48) into the (2.49), and for the block-index I_r, we are using, as in previous examples, entry from neighbouring elements list $i_r = \text{neigh}(i,k)$. Hence (2.49) is equivalent to (2.50). Moreover, this ensures, that only the entries which have an actual interaction, no multiplications with zero, are taking part in the M^{rr}'s assembly.

2.4 Right hand side

We consider all penalty elements, to be moved inside the primal formulation. We should define vector form for the right hand side of a system $F_h(v) = \int_\Omega f v dx$, taking ϕ_i as a projection of v on an element K_i yields the following

$$F_i = \int_\Omega \sum_{i=1}^{N_{mesh}\cdot N_{pd}^d} f(x)\cdot\phi_i(x)dx.$$

Thus we could set vector

$$F = (F_i)_{N_{mesh}\cdot N_{pd}^d} := ((f(x),\phi_i(x)))_{N_{mesh}\cdot N_{pd}^d}.$$

The time consumption for the right hand side calculation and assembly is similar to the one of the Galerkin matrices calculation and setup and can be estimated with, considering $N_g = N_{pd}$ numerical quadrature points, required for integral computation

$$T_{rhs} \approx N_{mesh}\cdot N_{pd}^d \cdot N_{pd}^d \tag{2.51}$$

2.5 Global matrix formulation

Finally, the primal bilinear form (2.2) in its full linear matrix system representing Local Discontinuous Galerkin method has the following form

$$(A - V + B + M^{rr} + M^{rl} + (M^{rl})^T + M^{ll} + S)\cdot c^u = F, \tag{2.52}$$

where matrices M^{rr}, M^{rl} and M^{ll} represent the corresponding rr, rl and ll contributions.

And setting $A_h := (A - V + B + M^{rr} + M^{rl} + (M^{rl})^T + M^{ll} + S)$ as Global Galerkin Matrix, we get the primal matrix form, suitable for solving as a linear system:

$$A_h \cdot c^u = F, \tag{2.53}$$

with c^u being a vector of unknowns.

The global matrix formulation is a matrix-matrix addition operation and would require at most:

$$T_{GM,LDG,addition} \approx 8\cdot N_{mesh}\cdot N_{pd}^{2d} \tag{2.54}$$

For the Symmetric Interior Penalty method, we have a more compact version of (2.52):

$$(A - V + S)\cdot c^u = F. \tag{2.55}$$

The global Galerkin matrix (2.53) stays the same, with $A_h := (A - V + S)$. The matrix-matrix addition operation would be:

$$T_{GM,IP,addition} \approx 3\cdot N_{mesh}\cdot N_{pd}^{2d}. \tag{2.56}$$

2.6 A-priori estimates

2.6.1 Error estimate for the model problems

To estimate how close the obtained solution $u_h \in V_{hp}$ to the exact solution $u \in V$, we can find a bound for the approximation error $||u - u_h||_{DG}$.

Following [29, section 2.2], introduce the broken Sobolev space of composite order s, for the partition $\mathscr{T}_h$,

$$H^s(\Omega, \mathscr{T}_h) = \{u \in L^2(\Omega) : \quad u|_K \in H^{s_K}(K), \quad \forall K \in \mathscr{T}_h\},$$

equipped with the broken Sobolev seminorm

$$|u|_{s,\mathscr{T}_h} = \left(\sum_{K \in \mathscr{T}_h} |u|^2_{H^{s_K}} \right)^{1/2}$$

Theorem 2.6.1. *From [29, Remark 3.12]. For uniform approximation orders $p_K = N_{pd} \geq 0, s_K = s, 1 \leq s \leq \min(N_{pd} + 1, k), k \geq 1$, and $h = \max_{K \in \mathscr{T}_h}(h_K)$, the approximation error bound is*

$$||u - u_h||_{DG} \leq C \left(\frac{h}{p_K + 1} \right)^{s - \frac{1}{2}} |u|_{s,\mathscr{T}_h},$$

with $C > 0$ independent of the polynomial degree and mesh size. The integer number $k \geq 1$ is arising from the broken Sobolev space order, chosen for the Discontinuous Galerkin approximation [29, Theorem 3.11].

For the proof we refer to the source, noting that it trivially follows from the study shown in [29, subsections 3.3 and 3.4].

2.6.2 hp-Condition number estimate

Condition number of a Galerkin matrix, which arise from discontinuous Galerkin method, serve the purpose of measuring how sensitive the approximation would be to the roundings and cancellations during the solution process.

The spectral condition number of a symmetric positive definite matrix A is given as a ratio of the largest $\lambda_{max}(A)$ and smallest $\lambda_{min}(A)$ eigenvalues of the matrix

$$\kappa(A) = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}.$$

From [4, Corollary 2.9] we derive the following

Theorem 2.6.2. *Condition number of a stiffness matrix A of a symmetric discontinuous Galerkin method can be bounded by*

$$\kappa(A) \leq \gamma \frac{\min_{K \in \mathscr{T}_h} p_K^4}{\min_{K \in \mathscr{T}_h} h_K^2} \frac{\max_{K \in \mathscr{T}_h} h_K^d}{\min_{K \in \mathscr{T}_h} h_K^d}.$$

for $\gamma > 1$ being meshsize and approximation polynomial degree independent,h_K is an element K size, p_K is a maximum polynomial degree on that element. Therefore, if the mesh $\mathscr{T}_h$ and the polynomial approximation degrees are globally uniform we have

$$\kappa(A) \leq C_\delta \gamma N_{pd}^{2d} h^{-d}, \tag{2.57}$$

with $C_\delta > 0$, $h = \max(h_K)$, and $N_{pd} = \max_{K \in \mathscr{T}_h}(p_K)$.

The proof can be obtained in [4, subsection 2.4]. Essentially, we use $p_K := N_{pd}$.

2.7 Post-processing

2.7.1 Condition number computation

The spectral condition number $\kappa(A)$ of the symmetric positive definite matrix A, is defined as a direct relation between the smallest λ_{min} and largest λ_{max} eigenvalues of the matrix

$$\kappa(A) := \frac{\lambda_{max}}{\lambda_{min}}.$$

Condition number is used as one of the measures of the stability of the approximation obtained by solving system of a form $Ax = b$, and, generally for $\kappa(A) = 10^n$, n is a number of digits lost in numerical precision of the solution method.

One of the methods to rapidly obtaining the extreme measures of the eigenvalues of the matrix is a symmetric Lanczos Algorithm, which defaults to an Arnoldi's method for symmetric positive definite matrices [46, section 6.6]. If matrix A is symmetric then the associated Hessenberg matrix H_m becomes symmetric tridiagonal, which in turn requires just three vectors to store. This reduces the general algebraic complexity of the algorithm.

Generally, the Hessenberg matrix

$$H_m = V^T AV, \quad H_m \in \mathbb{R}^{m\times m},$$

where $V \in \mathbb{R}^{n\times m}$ is a matrix with orthonormal columns and A is a matrix which eigenvalues are sought. For simplicity, we restrict ourselves to the case $m = n$.

The Lanczos algorithm is shown in an algorithm (9).

Algorithm 9 Lanczos algorithm

```
 1: Choose an initial vector $v$ of 2-norm unity.
 2: $\beta_1 \leftarrow 0$
 3: $v_0 \leftarrow 0$
 4: for $i = 1, 2, ..., m$ do
 5:     $w_i \leftarrow Av_i - \beta_i v_{i-1}$
 6:     $\alpha_i \leftarrow (w_i, v_i)$
 7:     $w_i \leftarrow w_i - \alpha_i v_i$
 8:     $\beta_{i+1} \leftarrow \|w_i\|_2$
 9:     if $\beta_{i+1} == 0$ then
10:         [STOP]
11:     end if
12:     $v_{i+1} \leftarrow w_i / \beta_{i+1}$
13: end for
```

This algorithm results in the Hessenberg matrix, assembled as follows:

$$H_m = \begin{pmatrix} \alpha_1 & \beta_2 & & 0 \\ \beta_2 & \ddots & \ddots & \\ & \ddots & \ddots & \beta_m \\ 0 & & \beta_m & \alpha_m \end{pmatrix}.$$

Let $m = n$, then if λ is an eigenvalue of a H_m, then it is also an eigenvalue of A and if $H_m x = \lambda x$, x is an eigenvector of H_m, then $y = Vx$ is the corresponding eigenvector of A, this allows to compute the eigenvector [46, Section 6.6]:

$$Ay = AVx = VH_mV^TVx = VH_mIx = VH_mx = V(\lambda x) = \lambda Vx = \lambda y.$$

This transforms the problem of eigendecomposition of potentially dense matrix A into the eigendecomposition of a tridiagonal matrix H_m.

Lanczos algorithm is cheap in terms of implementation, and is built-in into the conventional or preconditioned **CG** algorithm. This allows improved the numerical stability of the results obtained by Lanczos [46, algorithm 6.1.16]. At the same time it is not useful for larger m.

Results presented in Chapter **4** are calculated using the Lanczos algorithm.

2.7.2 Convergence rate calculation

According to lecture notes [37], if one knows the error of the approximated solution in L^2 or H_0^1, then it's possible to calculate convergence rate of the problem, using previous result and knowing the degree of freedom (number of unknowns), which is a result of the chosen space discretisation:

$$\alpha_i = -\frac{\log(e_{i+1}/e_i)}{\log(N_{i+1}/N_i)}$$

Where e_i is an L^2 or H_0^1 error estimate on a chosen partition, N_i is a degree of freedom of the chosen partition.

This estimate is easily obtained by noting that if

$$e_i \approx \frac{C}{N_i^\alpha}$$

then

$$\begin{aligned}\log(e_i) &\approx \log(C) - \alpha\log(N_i),\\ \log(e_{i+1}) &\approx \log(C) - \alpha\log(N_{i+1})\end{aligned}$$

leading to the estimate α_i of α just given.

Convergence rate is an important indicator for the numerical approximation tasks, as it is allows to see the actual gain in the precision of the solution with the growth in degree of freedom.

In order to check the implementation, we considered problems with known or computable approximation error in L^2 and H_0^1.

Chapter 3

Iterative solvers

3.1 Conjugate Gradients method

In order to solve the global linear system, shown in (2.53), we can use the iterative solver.

Algorithm 10 Conjugate Gradients method

```
 1: [Initialize] $r_0 \leftarrow b - Ax_0$
 2: $p_0 \leftarrow r_0$
 3: $k \leftarrow 0$
 4: while $|r_{k+1}| \geq \epsilon$ do
 5:     $q_k \leftarrow Ap_k$
 6:     $\alpha_k \leftarrow \frac{r_k^T r_k}{p_k^T q_k}$
 7:     $x_{k+1} \leftarrow x_k + \alpha_k p_k$
 8:     $r_{k+1} \leftarrow r_k - \alpha_k q_k$
 9:     $\beta_k \leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
10:     $p_{k+1} \leftarrow r_{k+1} + \beta_k p_k$
11:     $k \leftarrow k + 1$
12: end while
```

As a main solver we use the Conjugate Gradients (**CG**) iterative method. For the model problem of a form $Ax = b$, where A is real, symmetric, positive definite. The initial input vector x_0 can be an approximate solution or 0. The algorithm employs residual vector r, A-conjugate vector p, and scalars α and β in-line, making it cheap to control the stability of the solution. **CG**-method uses vectors b, x, r, p, q and input matrix A, which in the scope of our research is given by the global system matrix (2.55) or (2.52). Main operations which one should perform in order to implemented method is a matrix-vector multiplication and vector-vector dot product. Vector dot product can be performed by distributing the pieces of the vectors across processes. Each process then calculates vector sum of component-wise multiplication of the vectors from the assigned blocks and then that local sums can be gathered and summarised in order to obtain full sum of the dot product. For matrix-vector multiplication full row of the matrix should be multiplied on a full vector and summarised, what will give as a result element of the resulting vector, corresponding to the matrix's row. This implies that for matrix-vector multiplication one need to have full input vector which, but resulting vector can be split into pieces and distributed between all processes. However matrix should be split preserving the rows. That shows that only vector p should be stored as a global vector (on all processes) and all others can easily be computed locally (only the part required for computation on the local process).

The algorithm 12 uses expensive matrix-vector multiplication just once per iteration, making it attractive for parallel implementation. In case of parallel implementation, we do not have to store whole vectors r, x, q on all processes, but rather split the vectors in parts amongst processes. The same can be done with the Galerkin Matrix A. On the contrary to split vectors r, x, q, we would have to store full vector p on all processes and update it for every iteration. Due to a block-sparse nature of our Galerkin Matrix, we can define which entries of the employed vectors are necessary and reduce the split vector data even more.

3.1.1 Linear Algebra parallelisation techniques. Computational complexity

Here, in order to analyse the possibility of parallel implementation we need to determine complexity of the method in terms of operation count. We additionally should expand the notation.

First, we set, for square matrix A, $N_1 \equiv N_2 := N_{mesh} \cdot N_{pd}^d$, and define Galerkin matrix $A \in \mathbb{R}^{(N_{mesh} \cdot N_{pd}^d) \times (N_{mesh} \cdot N_{pd}^d)}$, and vectors $q, p \in \mathbb{R}^{N_{mesh} \cdot N_{pd}^d}$. Then, the matrix-vector multiplication operation $q_k = Ap_k$ used in the **CG** method, omitting the iteration counter k:

$$q = Ap, \tag{3.1}$$

Expanding the sum notation with $A := (a_{i,j})_{1,\dots,N_2}^{1,\dots,N_1}$, $q := (q_i)^{1,\dots,N_1}$ and $p := (p_j)_{1,\dots,N_2}$, we get:

$$q_i = \sum_{i=1}^{N_1} a_{i,j} p_j, \quad j = 1, \dots, N_2. \tag{3.2}$$

Now, we can introduce parallel computation using index notation. Say, we have $N_p \in \mathbb{N}^+$ processes, numbered as $\# = 0, 1, \dots, N_P - 1$. For simplicity, we establish limits for each process # as

$$1 \le N_{0,\#} < N_1, \tag{3.3}$$

$N_{0,\#}$ - being the lower bound for indices and,

$$1 < N_{1,\#} \le N_1, \tag{3.4}$$

$N_{1,\#}$ - being the upper bound for indices, with $N_{0,\#} \le N_{1,\#}$. Computation of the local limits is not as trivial as it seems and will be discussed further. We would use # as an upper index, denoting the process where the part of the vector q is stored. In order to assemble full vector, from it's distributed partitions, we use concatenation operation $||$, which is defined, for two vectors $a := (a_1, a_2, \dots, a_n)$ and $b := (b_1, b_2, \dots, b_n)$, as:

$$a || b := (a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_n).$$

We can now define the parallel matrix vector multiplication, for distributed vector $q^{\#} := (q_i)^{N_{0,\#},\dots,N_{1,\#}}$ and distributed matrix $A^{\#} := (a_{i,j})_{1,\dots,N_2}^{N_{0,\#},\dots,N_{1,\#}}$, as:

$$q^{\#} = A^{\#} p, \tag{3.5}$$

which requires full vector p. Full vector p will ensure that each element of $q^{\#}$ is computed fully, and there is no need to reduce the sum across the processes. Expanded form of (3.5) then is:

$$q_i = \sum_{i=N_{0,\#}}^{N_{1,\#}} a_{i,j} p_j, \quad j = 1, \dots, N_2. \tag{3.6}$$

Now, the computational complexity of the serial matrix-vector multiplication, for some general matrix $M \in \mathbb{R}^{(N_M \times N_M)}$, with arbitrary chosen N_M, and vector $v \in \mathbb{R}^{(N_M)}$ is

$$\mathbb{O}_{MV}^{dense}(N_M^2),$$

for dense M, and

$$\mathbb{O}_{MV}^{sparse}(N_M),$$

for the block-sparse M. This, when applied to (3.1,3.5), yields number of operations for serial and parallel matrix-vector multiplication for general choice, hence *pessimistic* estimate, in terms of problem's dimension d:

$$\begin{aligned}
T_{MV,ser}^{dense}(d) &= (2d+1)(N_{mesh} \cdot N_{pd}^d)^2 && (3.7)\\
T_{MV,ser}^{sparse}(d) &= (2d+1) N_{mesh} \cdot N_{pd}^{2d} && (3.8)\\
T_{MV,par}^{dense}(d) &= \frac{(2d+1)(N_{mesh} \cdot N_{pd}^d)^2}{N_P} && (3.9)\\
T_{MV,par}^{sparse}(d) &= \frac{(2d+1) N_{mesh} \cdot N_{pd}^{2d}}{N_P}, && (3.10)
\end{aligned}$$

where the factor $(2d+1)$ arise from the maximum number of neighbours through an edge and the element self-contribution per one element.

We should note, that we do not require to transfer any matrix data during or after the matrix-vector multiplication for **CG** method.

Next operation to expand, is computation of scalars $\alpha, \beta \in \mathbb{R}$. To compute α, we first split computation of the numerator and denominator, which are vector dot-products. Set $t := r_k^T r_k$ and $s := p_k^T q_k$, where vector $r \in \mathbb{R}^{N_{mesh} \cdot N_{pd}^d}$, given as $r := (r_i)^{1,\dots,N_1}$. Applying the same index limits (3.4,3.3) and omitting k, we get:

$$t^{\#} = \sum_{i=N_{0,\#}}^{N_{1,\#}} r_i r_i, \qquad t = \sum_{\#=0}^{N_P-1} t^{\#}, \tag{3.11}$$

$$s^{\#} = \sum_{i=N_{0,\#}}^{N_{1,\#}} p_i q_i, \qquad s = \sum_{\#=0}^{N_P-1} s^{\#},$$

$$\alpha = \frac{t}{s}. \tag{3.12}$$

Summation of scalars over processes, like in (3.11), is called scalar **reduction**, and requires transfer of a real number $t^{\#}$ from local process # to **root** process (conventionally 0-th process), where received numbers are summed. After all scalars reduced, we can perform (3.12), and transfer it from **root** process to all local processes. Knowing computational complexity of a dot-product $\mathbb{O}_{dot-product}(N_{mesh} \cdot N_{pd}^d)$, we can compute number of operations required, for serial and parallel algorithms:

$$\begin{aligned} T_{sc,ser}(d) &= 2N_{mesh} \cdot N_{pd}^d + 1, \\ T_{sc,par}(d) &= \frac{2N_{mesh} \cdot N_{pd}^d}{N_P} + 2N_P + 1, \end{aligned}$$

where $2N_P+1$ arise from the transfer operation. Two N_P-long transfer operations arise from the vector transfer, required for the scalar operation, and one 1-unit transfer operations arise from redistributing the scalar value amongst all the processes. We note that this is "at-most" value. Additionally, we note that one 1-unit transfer operation depends on a chosen precision for the data-type.

Storing value of t for current iteration, allows us to perform one less dot-product, computing β_k. This in turn decreases, in this particular case, number of operations for computation of α_k and β_k in total to:

$$T_{sc2,ser}(d) = 3N_{mesh} \cdot N_{pd}^d + 2, \tag{3.13}$$

$$T_{sc2,par}(d) = \frac{3N_{mesh} \cdot N_{pd}^d}{N_P} + 3N_P + 2, \tag{3.14}$$

where $3N_P+2$ arise from the transfer operation. We save one N_P-long transfer operation by storing values of vector t_k, as mentioned in paragraph before. This results in three N_P-long transfer operations for computation of both scalars α_k and β_k for current iteration k. Two 1-unit transfer operations arise from the scalar redistribution.

In **CG** method, we use so-called **zaxpy** operations. Considering $x_{k+1} = x_k + \alpha_k p_k$, set $z = x_{k+1}$, to separate new and old values for current x. This operation can be performed locally on each process #. Results of the computation, can be stored locally. In expanded notation for vector $x := (x_i)^{1,\dots,N_1}$, omitting k:

$$z_i = x_i + \alpha p_i, \quad i = 1, \dots, N_1.$$

And for local vectors $x^{\#} := (x_i)^{N_{0,\#},\dots,N_{1,\#}}, \quad z^{\#} := (z_i)^{N_{0,\#},\dots,N_{1,\#}}$:

$$z_i = x_i + \alpha p_i, \quad i = N_{0,\#}, \dots, N_{1,\#}.$$

Obviously, computational complexity of **zaxpy** operation will be a sum of complexities of vector-by-scalar multiplication $\mathbb{O}_{vsm}(N_{mesh} \cdot N_{pd}^d)$ and vector summation $\mathbb{O}_{vsm}(N_{mesh} \cdot N_{pd}^d)$.

This yields number of operations required:

$$T_{zaxpy,ser}(d) = 2N_{mesh} \cdot N_{pd}^d, \quad (3.15)$$

$$T_{zaxpy,par}(d) = \frac{2N_{mesh} \cdot N_{pd}^d}{N_P}. \quad (3.16)$$

We can now compute a total number of operations required per iteration of **CG** method, **excluding** the communication operations cost. We have three **zaxpy** operations with length of vectors being $N_{mesh} \cdot N_{pd}^d$, two operations for computation of search directions, with three unique dot-products, we reuse results of residual's dot-product, and, finally, the matrix-vector multiplication. Summing over the time required for operations, we get:

$$T_{CG,It,ser}(d) = 3 \cdot T_{zaxpy,ser}(d) + T_{sd2,ser}(d) + T_{MV,ser}(d), \quad (3.17)$$

$$T_{CG,It,par}(d) = 3 \cdot T_{zaxpy,par}(d) + T_{sd2,par}(d) + T_{MV,par}(d). \quad (3.18)$$

Now, we can introduce T_{tr} as a time required for one bite to be transferred from process to process in seconds. Additionally we require to add the time necessary for establishing connection among hosts, and account for network\bus latency time. We require, to send $N_P - 1$ real numbers during the scalars computations. This transfer is required for (one) computation of $r_k^T r_k$, (one) computation of $p_k^T q_k$, (one) computation of $r_{k+1}^T r_{k+1}$. We need to send resulting α_k and β_k after computation from **root** to distributed processes, totalling to $N_P - 1$ per each scalar. This totals to:

$$T_{sc,tr}(d) = 5(N_P - 1). \quad (3.19)$$

Now we analyse the computation and transfer of conjugate vector p_{k+1} of length $N_{mesh} \cdot N_{pd}^d$. Computation is a standard **zaxpy** operation with number of operations shown in (3.16). Parallel **zaxpy** operation will only update elements of p_{k+1}, unique for each process #, with indices $i = N_{0,\#}, \ldots, N_{1,\#}$. As we require to perform matrix-vector multiplication in each consecutive iteration, we need whole vector p_{k+1} on all processes. At most, we require N_P processes to send $\frac{N_{mesh} \cdot N_{pd}^d}{N_P}$ real numbers to $N_P - 1$ other processes:

$$\begin{aligned} T_{p_{k+1},tr}(d) &= N_P(N_P - 1)\frac{N_{mesh} \cdot N_{pd}^d}{N_P}, \\ &= (N_P - 1) \cdot (N_{mesh} \cdot N_{pd}^d). \end{aligned} \quad (3.20)$$

Adding all the transfer operations (3.19,3.20) with the algebraic cost of **CG** (3.18), yields:

$$T_{CG,It,par}(d) = 3 \cdot T_{zaxpy,par}(d) + T_{sd2,par}(d) + T_{MV,par}(d) + T_{p_{k+1},tr}(d) + T_{sc,tr}(d). \quad (3.21)$$

Now, that we have defined general tools, for complexity analysis, we can perform an analysis of the matrix-vector multiplication for a specific choice of the basis functions **ADLP** and **C1**.

3.1.2 Effect of the basis functions on matrix-vector multiplication

We will now analyse the cost of parallel matrix-vector multiplication, as the most expensive operation within **CG** method. We analyse it for different choice of the local basis functions and for the different ways of the distribution of data with the processes. This allows us to compute the *optimistic* operation counts for the **CG** algorithm.

We perform analysis of the three cases with block-sparse system matrices and a general case with no optimization to matrix structure.

First, we will take a look at the serial implementation of **CG** for the general structure of the system matrix. Substitute (3.8,3.13,3.15) into (3.17):

$$\begin{aligned} T^{GEN}_{CG,It,ser}(d) &= 6N_{mesh}\cdot N^d_{pd} + 3N_{mesh}\cdot N^d_{pd} + 2 + (2d+1)N_{mesh}\cdot N^{2d}_{pd} \\ &= 9N_{mesh}\cdot N^d_{pd} + (2d+1)N_{mesh}\cdot N^{2d}_{pd} + 2. \end{aligned}$$

Parallel algorithm requires, for computation only:

$$\begin{aligned} T^{GEN}_{CG,It,par,comp}(d) &= 3\frac{2N_{mesh}\cdot N^d_{pd}}{N_P} + \frac{3N_{mesh}\cdot N^d_{pd}}{N_P} + 3N_P + 2 + \frac{(2d+1)N_{mesh}\cdot N^{2d}_{pd}}{N_P} \\ &= \frac{9N_{mesh}\cdot N^d_{pd}}{N_P} + \frac{(2d+1)N_{mesh}\cdot N^{2d}_{pd}}{N_P} + 3N_P + 2. \end{aligned} \quad (3.22)$$

Now summing with all transfer costs (3.19,3.20) with computation cost for *pessimistic* implementation of **CG** (3.22):

$$\begin{aligned} T^{GEN}_{CG,IT,par}(d) &= \frac{9N_{mesh}\cdot N^d_{pd}}{N_P} + \frac{(2d+1)N_{mesh}\cdot N^{2d}_{pd}}{N_P} \\ &+ 3N_P + 2 + 5(N_P-1) + (N_P-1)\cdot(N_{mesh}\cdot N^d_{pd}) \\ &= \frac{9N_{mesh}\cdot N^d_{pd}}{N_P} + \frac{(2d+1)N_{mesh}\cdot N^{2d}_{pd}}{N_P} \\ &+ (N_{mesh}\cdot N^d_{pd} + 5) + (3N_P + 2) \end{aligned} \quad (3.23)$$

From equation (3.23), we conclude that an increase in the number of processes, while decreasing computational complexity per iteration, will increase the number of transfer operations undertaken. At the same time, the number of transfer operations grows slower than the gain from parallel computation. However, conventionally, transfer operations are much more slower than those of summation and multiplication, and parallelisation effect can be lost for massively-parallel execution.

Now, for the special choice of the basis functions, we analyse the global matrix structure. It should have a block structure, where each diagonal block is contributed by matrix A, and all off-diagonal blocks are sums of the blocks of edge-based matrices V, B, S etc. Each block-row, for quadrilateral space discretisation, would have at most $1+2d$ blocks. In some cases we would have less off-diagonal blocks, which depends on a position of a quadrilateral element in space, e.g. when the element have all four neighbours, we would have four blocks, when the element is on the boundary, we would have only two or three blocks, depending on the actual position and geometry of the space. All diagonal blocks would be obtained from a volume integral for the bilinear form A, hence these blocks are dense, with N_{pd}^{2d} non-zero elements.

Off-diagonal blocks would have different structure, as shown in ((**S.1**)2.3.2). Comparing that structure against the terms of primal bilinear form (2.2) given in (2.8) it is clear, that we do not face gradient-gradient interaction, but, at most, function-gradient interaction. This yields the following structure:

I **General case.** All dense with N_{pd}^{2d} non-zero elements,

II **Antiderivative of Legendre Polynomials.** All dense with $N_{pd}^{d} \cdot N_{pd}^{d-1}$ non-zero elements,

III **C1 polynomials.** All p-sparse with $N_{pd}^{d-1} \cdot N_{pd}^{d-1}$ non-zero elements.

List 3.1: Count of non-zero elements in the system matrix off-diagonal blocks. SIPG method.

The off-diagonal entries in Symmetric **IPG** arise from the element-neighbour interactions, knowing that for d-cube domain discretization, we would have at most $2d$ neighbours with shared edge per element, we can determine number of total contributions from off-diagonal blocks.

The structure of the system matrices can be derived using the basis function properties and the statements above. Considering Interior Penalty Discontinuous Galerkin method, we construct the following image of the system matrices:

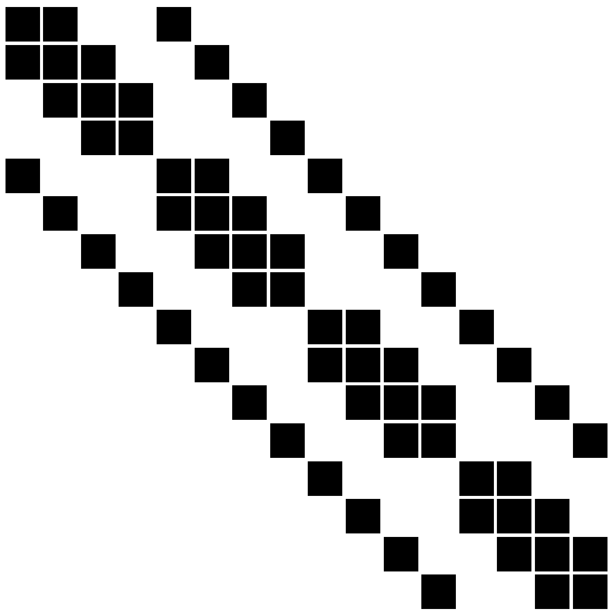


Figure 3.1: Interior Penalty method Galerkin matrix for General choice of basis functions

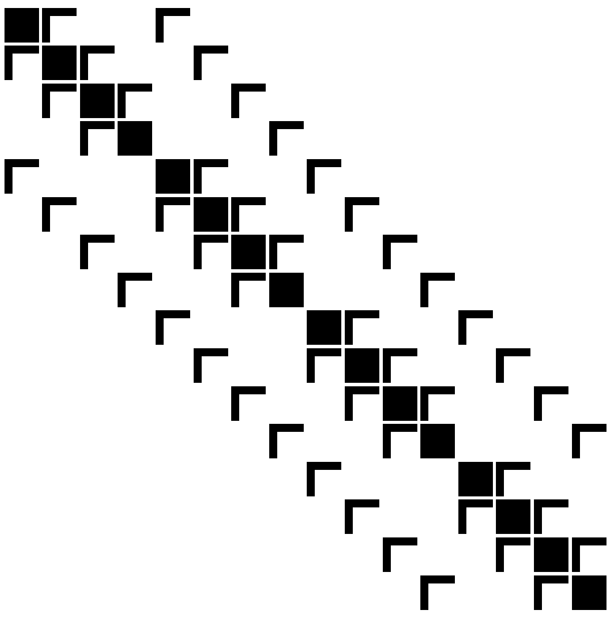


Figure 3.2: Interior Penalty method Galerkin matrix for ADLP basis functions

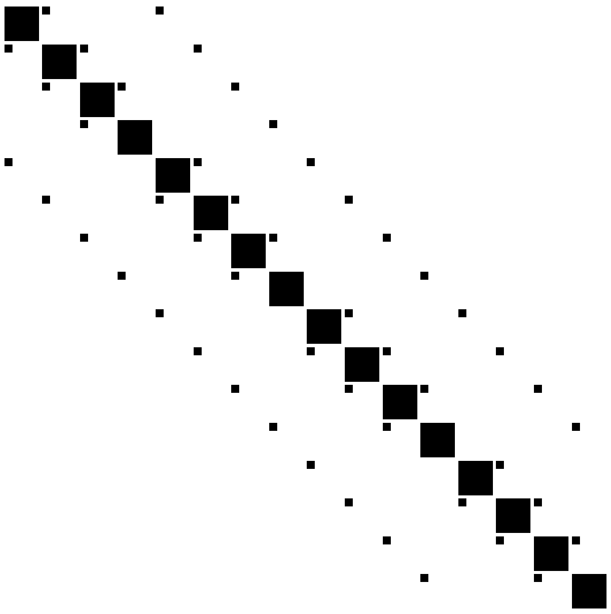


Figure 3.3: Interior Penalty method Galerkin matrix for C1 basis functions

For general data storage, we choose to transfer all data, computed on each process from that process to all other processes. This is the most straightforward, but the most time consuming case.

Instead we can base the distributed data structure for storage and transfer, purely on the system matrix structure or choice of basis functions. In order to define such a structure, we first introduce some basic tools required for the distributed data structure.

For a system matrix $A := (a_{i,j})_{N_{mesh}\cdot N_{pd}^d}^{N_{mesh}\cdot N_{pd}^d}$, with total number of entries $(N_{mesh} \cdot N_{pd}^d)^2$, for each process $\# = 0, \dots, N_P - 1$ we create a lists χ of a size at most $3 \cdot N_P \times (N_{mesh} \cdot N_{pd}^d)^2$. For each process number #, there is a triplet of columns at positions $\{3(\#)+1, 3\#+2, 3\#+3\}$, from total of $1, \ldots, 3 \cdot N_P$ columns. For the parallel split based on the internal structure of the matrix A, the columns' $(N_{mesh} \cdot N_{pd}^d)^2$ entries store the following information:

1. $3\# + 1$ Column index of matrix A - i
2. $3\# + 2$ Row index of matrix A - j
3. $3\# + 3$Process number k which stores the entry $a_{i,j}$

The list above can be optimized in terms of the storage size. We can do so, by not including entries $a_{i,j}$ of matrix A which are equal to zero.

Now, the determination of which process stores particular entries of the matrix A, in the most general way, can be done in the following way. Introducing local, for the process $\#$, lower limit (3.3):

$$1 \le N_{0,\#} < N_{mesh} \cdot N_{pd}^d,$$

being the lower bound for indices and, upper limit (3.4):

$$1 < N_{1,\#} \le N_{mesh} \cdot N_{pd}^d,$$

being the upper bound for indices, with $N_{0,\#} \le N_{1,\#}$.

So-called equal split allows to define them as:

$$N_{0,\#} = \begin{cases} 1 & \text{if } \# = 0 \\ \# \cdot \left(\frac{N_{mesh} \cdot N_{pd}^d}{N_P} \right) & \text{else} \end{cases}$$

and

$$N_{1,\#} = \begin{cases} N_{mesh} \cdot N_{pd}^d & \text{if } \# = N_P - 1 \\ \# \cdot \left(\frac{N_{mesh} \cdot N_{pd}^d}{N_P} \right) & \text{else} \end{cases}$$

The lower and upper index limiters, allows us to reduce the size of the list χ even further, limiting the number of required rows N_r to:

$$N_r = \max(N_{1,\#} - N_{0,\#}), \# = 0, \ldots, N_P - 1.$$

For our choice of the basis functions, we can use the geometric split of the mesh, and then accounting only for basis functions on the elements, which belong to different processes. This requires analysis of the mesh to determine which of the basis functions are going to be used during the local matrices computation. We would have to perform an analysis for each element in order to determine if that element's local basis functions are required on more than one process. We can establish which process $\#$ will posses the coefficients which arise from the particular basis function and which other processes will require those coefficients, if any. Once this is established, we can use the data structure χ, keeping its internal structure. This type of split requires more storage for the list χ itself, but minimizes the data transfer, due to the fact that we essentially require only to transfer coefficients which are employed on more than one process. This also allows to parallelize the fine grid level exact solvers of the Additive two-level non-overlapping Schwarz method.

The total amount of data to transfer during the matrix-vector multiplication, according to the aforementioned choice is present in the following table 3.1, for different choice of the basis functions and type of the data storage.

Table 3.1: Matrix-vector multiplication per off-diagonal block and required data transfer for special choice of basis functions. Data transfer for all processes.$N_{mesh}^{1/d} = h^{-1}$ arises from the geometrical splitting.

B.F.	MV mult	Data transfer		
		Full broadcast	Matrix based	Geometry based
General	$N_{mesh}N_{pd}^{2d}$	$(N_p-1)N_{mesh}N_{pd}^{d}$	-	-
ADLP	$N_{mesh}N_{pd}^{2d-1}$	$(N_p-1)N_{mesh}N_{pd}^{d}$	$(N_p-1)N_{mesh}^{1/d}N_{pd}^{d}$	$(N_p-1)N_{mesh}^{1/d}N_{pd}^{d}$
C1	$N_{mesh}N_{pd}^{2d-2}$	$(N_p-1)N_{mesh}N_{pd}^{d}$	$(N_p-1)N_{mesh}^{1/d}N_{pd}^{d-1}$	$(N_p-1)N_{mesh}^{1/d}N_{pd}^{d-1}$

Using table 3.1 we can derive necessary number of operations for each matrix-vector multiplications. For ADLP basis functions, in parallel implementation, accounting for at most $2d$ neighbours (second column), and the main diagonal block, and the data transfer (fourth or fifth columns), we have

$$T_{MV,ADLP} = \frac{N_{mesh}\cdot N_{pd}^{2d}}{N_P} + \frac{2dN_{mesh}\cdot N_{pd}^{2d-1}}{N_P} + (N_P-1)\cdot N_{mesh}^{1/d}N_{pd}^{d}, \tag{3.24}$$

and substituting it and (3.13,3.15) into the (3.21), yields total cost per iteration

$$\begin{aligned} T_{CG,IT,par}^{ADLP}(d) &= \frac{9N_{mesh}\cdot N_{pd}^{d}}{N_P} + \frac{N_{mesh}\cdot N_{pd}^{2d}}{N_P} + \frac{2dN_{mesh}\cdot N_{pd}^{2d-1}}{N_P} + (N_P-1)\cdot N_{mesh}^{1/d}N_{pd}^{d} \\ &+ 3N_P + 2 + 5(N_P-1) + (N_P-1)(N_{mesh}\cdot N_{pd}^{d}). \end{aligned} \tag{3.25}$$

Similarly to (3.24) for C1 basis function we have:

$$T_{MV,C1} = \frac{N_{mesh}\cdot N_{pd}^{2d} + 2dN_{mesh}\cdot N_{pd}^{2d-2}}{N_P} + (N_P-1)N_{mesh}^{1/d}N_{pd}^{d-1}. \tag{3.26}$$

Now, by substituting (3.26,3.13,3.15) into the (3.21), we obtain the **CG** total operations count, including the data transfer:

$$\begin{aligned} T_{CG,IT,par}^{C1}(d) &= \frac{9N_{mesh}\cdot N_{pd}^{d}}{N_P} + \frac{N_{mesh}\cdot N_{pd}^{2d} + 2dN_{mesh}\,N_{pd}^{2d-2}}{N_P} + (N_P-1)N_{mesh}^{1/d}N_{pd}^{d-1} \\ &+ 3N_P + 2 + 5(N_P-1) + (N_P-1)\cdot(N_{mesh}\cdot N_{pd}^{d}) \\ &= \frac{9N_{mesh}\cdot N_{pd}^{d}}{N_P} + \frac{N_{mesh}\cdot N_{pd}^{2d} + 2dN_{mesh}\,N_{pd}^{2d-2}}{N_P} + (N_P-1)N_{mesh}^{1/d}N_{pd}^{d-1} \\ &+ (N_P-1)\cdot(N_{mesh}\cdot N_{pd}^{d}+3) + 5N_P \end{aligned} \tag{3.27}$$

Using C1 basis functions saves asymptotically, as $N_{mesh}\to\infty$:

80% of memory and operations in 2d in comparison to **ADLP**

86% of memory and operations in 3d in comparison to **ADLP**

We also should introduce the initialisation time T_{Init} for the MPI protocol, which have unavoidable impact on the overall time, but happens only once during the first iteration. So, the total time is:

$$T_{TTL} := N_{It} \cdot T_{It} + T_{Init}.$$

One should note, that the initialisation time T_{Init} depends on network topology, number of physical machines in cluster, number of assigned sockets per physical machine, number of processes and other hardware- and software-related properties. For simplicity, we will omit any analysis of the initialization time, as it goes beyond the scope of this work.

3.1.3 Convergence estimate

Introducing N_{It} to denote number of required iterations yields the final total number of operations cost required for CG algorithm,

$$T_{CG} \approx N_{It} \cdot T_{CG,It}. \tag{3.28}$$

We will now analyse the number of iterations N_{It} and upper bound, in order to have a definite comparison between classical parallel **CG** and the method extended with preconditioner. From [46, Theorem 6.29] , we know, that the upper bound for convergence holds as:

$$\|x_* - x_k\|_A \leq 2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k \|x^* - x_0\|_A,$$

where, x_* is the exact solution, x_k is the solution on k-th step, x_0 is the first approximation, $\kappa(A)$ is the condition number of matrix A,

$$\kappa(A) := \frac{\lambda_{max}(A)}{\lambda_{min}(A)}, \tag{3.29}$$

with $\lambda_{max}, \lambda_{min}$, are respectively maximal and minimal eigenvalues of the matrix A, and the norm $\|x\|_A^2 = \langle Ax, x\rangle$. The error estimate on the step k can be bounded with desirable value ϵ as $\|x_* - x_k\|_A \leq \epsilon \cdot \|x_* - x_0\|_A$ [5].

Lemma 3.1.1. *The number of steps k to reach a relative approximation error of $\epsilon > 0$ i.e.*

$$\|x_* - x_k\|_A \leq \epsilon \cdot \|x_* - x_0\|_A,$$

is given by

$$k \geq \ln\left(\frac{2}{\epsilon}\right)\frac{(\sqrt{\kappa(A)}+1)}{2}.$$

Proof. We need to find k, such that

$$2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k \leq \epsilon.$$

First:

$$\begin{aligned} 2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k &\leq \epsilon \\ \Leftrightarrow \ln(2)+k\ln(\left[\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right]) &\leq \ln(\epsilon) \\ \Leftrightarrow k &\geq \frac{\ln(\frac{2}{\epsilon})}{-\ln(\left[\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right])} \end{aligned} \tag{3.30}$$

Analysing the $\ln(\left[\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right]) = \ln(\left[1-\frac{2}{\sqrt{\kappa(A)}+1}\right])$.

Set $x = -\frac{2}{\sqrt{\kappa(A)+1}}$ and obtain $\ln(1+x)$. We next show that $\ln(1+x) \leq x, \quad x > -1$. We have, with $f(x) = \ln(1+x) - x$ that

$$f(0) = 0$$

and

$$f'(x) = \frac{1}{x+1} - 1 = \frac{1-x-1}{x+1} = -\frac{x}{x+1} < 0, \qquad \text{for } x > 0.$$

And for $-1 < x < 0$:

$$f'(x) = -\frac{x}{x+1} > 0.$$

Therefore $f(x) \leq 0$ for $x > 0$, and therefore

$$\ln(1+x) \leq x.$$

Substituting the x back:

$$\ln(\left[1-\frac{2}{\sqrt{\kappa(A)}+1}\right]) \leq -\frac{2}{\sqrt{\kappa(A)}+1}$$

And substituting the result into (3.30):

$$k \geq \ln(\frac{2}{\epsilon})\frac{(\sqrt{\kappa(A)}+1)}{2} \tag{3.31}$$

This completes the proof.

□

Corollary 3.1.2. *Condition number can be bounded by (2.57) of Theorem 2.6.2*

$$\kappa(A) \leq C_\delta \gamma N_{pd}^{2d} h^{-d}, \tag{3.32}$$

and noting that iteration number k is always a positive integer, the inequality (3.31) can be written as:

$$k \geq \left\lceil \ln(\frac{2}{\epsilon})\frac{\sqrt{C_\delta h^{-2} p^4}+1}{2} \right\rceil \tag{3.33}$$

3.1.4 Preconditioned Conjugate Gradient method

The **CG** method analysed in the previous section, is considered numerically stable solver ([46]), however, for the cases when number of iterations k is too high ($\kappa(A)$ is large enough), e.g. $k \to \mathrm{DoF}(A)$, we might better use a preconditioner for our system. Given a symmetric, positive definite matrix B, with slight abuse of the notation, we can consider the modified linear system (2.1) for $u_h \in V_{hp}$:

$$BA_h(u_h, v) = BF_h(v), \tag{3.34}$$

for $v \in V_{hp}$ is an arbitrary basis function.

We can obtain a modified linear system in matrix notation, from (2.53):

$$BA_h c^u = BF, \tag{3.35}$$

We note that even though BA_h is not necessarily symmetric, it is positive definite and it reduces to the identity in case $B = A_h^{-1}$. Substituting aforementioned into the CG algorithm (12), we get:

Algorithm 11 Preconditioned CG method

```
1: [Initialize] $r_0 \leftarrow b - Ax_0$
2: $z_0 \leftarrow Br_0$
3: $p_0 \leftarrow z_0$
4: $k \leftarrow 0$
5: **while** $|r_{k+1}| \geq \epsilon$ **do**
6:     $q_k \leftarrow Ap_k$
7:     $\alpha_k \leftarrow \frac{z_k^T r_k}{p_k^T q_k}$
8:     $x_{k+1} \leftarrow x_k + \alpha_k p_k$
9:     $r_{k+1} \leftarrow r_k - \alpha_k q_k$
10:    $z_{k+1} \leftarrow Br_{k+1}$
11:    $\beta_k \leftarrow \frac{z_{k+1}^T r_{k+1}}{r_k^T r_k}$
12:    $p_{k+1} \leftarrow z_{k+1} + \beta_k p_k$
13:    $k \leftarrow k + 1$
14: **end while**
```

We should note, that the line (10) of the algorithm 11 is a Fletcher-Reeves formula. It can be substituted by Polak-Ribiére formula $\beta_k \leftarrow \frac{z_{k+1}^T (r_{k+1} - r_k)}{z_k^T r_k}$ which adds additional vector to store, but adds more flexibility to the method [46, Section 6.7].

In this case we are interested in the structure of the preconditioning matrix B, how it impacts the algebraic complexity of the solver and communication cost compared to the non-preconditioned system.

Another factor which is of our most attention is choosing B such that $\kappa(BA) < \kappa(A)$, so the number of iterations of preconditioned solver is smaller than the one of the original $k_{prec} < k$. Later we will show, that k_{prec} should satisfy $k_{prec} \leq C_{prec}(BA)k$, to be worthwhile the preconditioning for a factor $C_{prec}(BA)$, depending on the precontidioned system.

The Additive two level Schwarz method can be used, to construct the preconditioner for our model problem [52].

3.2 The Additive Schwarz Method

A component which should be introduced in this work for successful preconditioning of the resulting Discontinuous Galerkin system

$$Au = f$$

is the domain decomposition method. In order to do this we should introduce the classical injection and projection operators and the auxiliary spaces.

We introduce N_S non-overlapping subdomains Ω_i with respect to the original domain Ω to support the finite decompositions as in (1.3):

$$\Omega = \cup_{i=1}^{N_S} \Omega_i.$$

As in [2, section 3], we can now denote by $\mathscr{T}_S$ a family of partitions Ω_i and start with introduction of the coarse partition $\mathscr{T}_H$ with H being a mesh size, in addition to the original partition $\mathscr{T}_h$(1.4). As in [2, (19)], we only consider nested partitions:

$$\mathscr{T}_S \subseteq \mathscr{T}_H \subseteq \mathscr{T}_h.$$

Additionally, we introduce the sets of all faces $\mathscr{F}_h^i$ for each subdomain $\Omega_i, \quad i = 1, \ldots, N_S$ similarly to original $\mathscr{F}_h$ including both interior and boundary faces. We can introduce the local and coarse solvers as given by Antonietti et al [5]. Local spaces are defined as restrictions of the finite element space V_{hp} (1.7) to the subdomains Ω_i:

$$V_{hp}^i := \{v \in L^2(\Omega_i) \quad : v\,|_K \in P(K) \quad \forall \hat{K} \in \mathscr{T}_H\}, \quad i = 1, \ldots, N_S, \tag{3.36}$$

and to restrict the finite element space W_{hp} (1.9) we use the second term of (1.10):

$$W_{hp}^i := [V_{hp}^i]^d, \quad i = 1, \ldots, N_S.$$

The local primal forms are then given as:

$$A_i : V_{hp}^i \times V_{hp}^i \to \mathbb{R}, \quad A_i(v_i, w_i) = A_h(R_i^T v_i, R_i^T w_i) \quad \forall v_i, w_i \in V_{hp}^i, \quad i = 1, \ldots, N_S \tag{3.37}$$

with $R_i^T : V_{hp}^i \to V_{hp}, \quad i = 1, \ldots, N_S$ being a classical injection operator from V_{hp}^i to V_{hp}.

We now can choose polynomial degree q, such that $1 \leq q \leq p$ and define a finite element space, analogous to the space V_{hp} (1.7):

$$V_{Hq} := \{v \in L^2(\Omega) \quad : v\,|_K \in P(K) \quad \forall \hat{K} \in \mathscr{T}_H\}, \tag{3.38}$$

where $\hat{K}$ is an element of a coarse space discretization $\mathscr{T}_H$ and $P(K) = [\mathscr{P}_q(\hat{K})]^d$ with polynomial degree q, and the following inclusion holds

$$V_{Hq} \subseteq V_{hp}, \tag{3.39}$$

in addition, for the method considered in this work, we set

$$V_{hp}^0 := V_{Hq}. \tag{3.40}$$

The coarse bilinear form would look as:

$$A_0 : V_{Hq} \times V_{Hq} \to \mathbb{R}, \quad A_0(v_0, w_0) = A_h(R_0^T v_0, R_0^T w_0) \quad \forall v_0, w_0 \in V_{Hq} \tag{3.41}$$

with $R_0^T : V_{Hq} \to V_{hp}$ a classical injection operator from V_{Hq} to V_{hp}. Introducing the projection operators $P_i = R_i^T \tilde{P}_i : V_{hp} \to V_{hp}, \quad i = 0, 1, \ldots, N_S$, where

$$\tilde{P}_i : V_{hp} \to V_{hp}^i, \quad A_i(\tilde{P}_i v_h, w_i) = A_h(v_h, R_i^T w_i) \quad \forall w_i \in V_{hp}^i, \quad i = 1, \ldots, N_S, \tag{3.42}$$

$$\tilde{P}_0 : V_{hp} \to V_{Hq}, \quad A_0(\tilde{P}_0 v_h, w_0) = A_h(v_h, R_0^T w_0) \quad \forall w_0 \in V_{Hq},$$

the additive Schwarz operator is defined by:

$$P_{ad} = \sum_{i=0}^{N_S} P_i, \tag{3.43}$$

for a total of N_S subspace partitions for V_{hp}.

Then, the Schwarz method consists of solving, by a suitable Krylov iterative solver, the system of equations

$$P_{ad} u_h = F_h, \tag{3.44}$$

where F_h is an appropriate right hand side, and $u_h \in V_{hp}$ is a corresponding approximated solution. Recalling the space partitioning (3.36) and introduced finite element space (3.38), we need to account which and how the basis functions on the coarse partition would look and how they implicate the Discontinuous Galerkin system and the preconditioned system (3.44). We note, that the Schwarz operator is invertible by construction. To do this we first find and analyse the local preconditioner in matrix form. Consider the form defined as local primal form for the system (3.37):

$$A_i(v_i, w_i) = A_h(R_i^T v_i, R_i^T w_i) \quad \forall v_i, w_i \in V_{hp}^i, \quad i = 1, \ldots, N_{mesh}.$$

To derive the method's matrix formulation, we start with the matrices C^i, which are, sometimes, called *restriction* matrices and $(C^i)^T$ are sometimes called *prolongation* matrices [35]. To understand the formation of matrices C^i, we should recall the domain decomposition $\Omega = \Omega_1 \cup \ldots \cup \Omega_{N_S}$, such that each Ω_i is a union of polyhedra in $\mathscr{T}_h$, we can form restriction matrices $C^0, \ldots, C^{N_S}$ which restrict to those vertices in the interior of Ω_i. These matrices are uniquely determined up to permutation of the rows of global matrix A_h. We should also note that the dimension of those matrices, in our case, is the

$$\begin{aligned} C^i &\in \mathbb{R}^{\frac{N_{mesh}N_{pd}^d}{N_S} \times N_{mesh}N_{pd}^d}, \quad i = 1, \dots, N_S, \\ C^0 &\in \mathbb{R}^{N_{mesh}(\frac{h}{H})^d q^d \times N_{mesh}N_{pd}^d}. \end{aligned}$$

To compute the matrices C^i, we first need define the **number of non-zero coefficients of C^i required for the fine grid partitioning** as $N_{sub,i}, \quad i = 0, \dots, N_S$, designating separate $N_{sub,0}$ as a **number of coefficients for the coarse partitioning**. As we have N_S fine partitions on the original one, for each matrix A_i being a local part of an original matrix A_h, with at-most N^2 elements, for

$$N = N_{mesh} \cdot N_{pd}^d,$$

we can set

$$N_{sub,i} = \frac{N_{mesh} \cdot N_{pd}^d}{N_S}, \quad i = 1, \dots, N_S \tag{3.45}$$

$$N_{sub,0} = N_{mesh}(\frac{h}{H})^d \cdot q^d, \tag{3.46}$$

Obviously, $N_{sub,i}$ is a constant for any $\quad i = 1, \dots, N_S$ for non-overlapping Additive Schwarz Method, when basis functions on both fine and sub- grids have the same polynomial degree. We set:

$$N_{sub,f} := N_{sub,1} \equiv N_{sub,2} \equiv \dots \equiv N_{sub,N_S}. \tag{3.47}$$

Additionally, we restrict ourselves to such N_S, that $N_{sub,f}$ and $N_{sub,0}$ are positive integers.

Recall the definition of the classical injection operator $R_i^T : V_{hp}^i \to V_{hp}, \quad i = 1, \dots, N_{mesh}$, we can find such $u_i \in V_{hp}^i$, that $R_i^T u_i := u, \quad R_i^T u_i \in V_{hp}$. Recalling the basis function notation for u_h (2.15), we choose the basis

$$\begin{aligned} V_{hp} &= \text{span}\{\phi_j, \quad j = 1, \dots, N\} \\ V_{hp}^i &= \text{span}\{\phi_k^i, \quad k = 1, \dots, N_{sub,i}, \quad i = 0, \dots, N_S\}, \end{aligned}$$

such that,

$$u_i = \sum_{k=1}^{N_{sub,f}} d_k \phi_k^i(x), \quad u_i \in V_{hp}^i,$$

where $\overrightarrow{d} = (d_k), \quad k = 1, \dots, N_{sub,f}$ is u_i-th coefficients vector.

Applying the operator to the basis function:

$$R_i^T \phi_k^i(x) \in V_{hp} \Rightarrow R_i \phi_k^i(x) := \sum_{j=1}^{N} C_{k,j}^i \phi_j(x),$$

where C^i is the local basis functions' linear configuration matrix, which maps the fine grid's basis functions numerical values from V_{hp} onto V_{hp}^i. Here, we should note that we employ the same basis functions on both fine and coarse grids.

Applying the operator R_i^T on u_i with $i = 1, \dots, N_S$:

$$\begin{aligned} R_i^T \sum_{k=1}^{N_{sub,i}} d_k \phi_k^i(x) &= \sum_{k=1}^{N_{sub,i}} d_k^T R_i \phi_k^i(x) \\ &= \sum_{k=1}^{N_{sub,i}} d_k \sum_{j=1}^{N} C_{k,j}^i \phi_j(x), \end{aligned} \tag{3.48}$$

And substituting both operator resulting in (3.48) and discrete u in identity $R_i^T u_i := u$:

$$\begin{aligned} \sum_{k=1}^{N_{sub,i}} d_k \sum_{j=1}^{N} C_{k,j}^i \phi_j(x) &= \sum_{j}^{N} c_j \phi_j \\ \sum_{j=1}^{N} \sum_{k=1}^{N_{sub,i}} d_k C_{k,j}^i \phi_j(x) &= \sum_{j}^{N} c_j \phi_j, \end{aligned}$$

for $i = 1, \dots, N_S$, yields matrix identity:

$$\overrightarrow{d}^T C^i = c. \tag{3.49}$$

This yields the matrix formulation of the preconditioning matrix, for $\bar{A}_i$ and $\bar{A}_{Hq}$ being a matrix representation of the A_i (3.37) and A_0 being a matrix representation of the $\bar{A_0}$ (3.41):

$$B_{ad} = \left(\sum_{i=1}^{N_S} C^i \bar{A}_i^{-1} (C^i)^T \right) + C^0 \bar{A}_{Hq}^{-1} (C^0)^T. \tag{3.50}$$

Then the matrix representation of the additive Schwarz operator P_{ad} (3.43) is given as:

$$P_{ad} = \sum_{i=0}^{N_S} P_i = \left(\sum_{i=1}^{N_S} C^i \bar{A}_i^{-1} (C^i)^T \right) + C^0 \bar{A}_{Hq}^{-1} (C^0)^T.$$

In this work we will consider the special case $q = 1$. This will reduce the complexity of both the coarse level restriction\prolongation matrices, and the coarse level preconditioner matrix. We will analyse this case in the following subsection.

3.2.1 Computational complexity

To compute the algebraic cost of the preconditioner computation on a local level, we use the assembly scheme, shown in (3.50).

Second, we should estimate the matrix product cost, namely the $C^i A_i^{-1} (C^i)^T$. Knowing the number of non-zero entries both in the C^i matrices and, in the worst case, in the most dense global matrix blocks as N_{pd}^{2d}, we can compute the matrix product cost:

$$T_{\bar{A}} := (N_{pd}^{2d})^2 + ((\frac{h}{H})^d \cdot q^d)^2 + (N_{pd}^{2d})^2 = 2N_{pd}^{4d} + (\frac{h}{H})^{2d} \cdot q^{2d}$$

We now estimate the cost of the matrix B local partitions B_i computation. We add the algebraic operations costs for the inverse of the matrix $\bar{A}_i$. Then we add the cost of matrix-

matrix and two matrix-vector multiplications. We also note, that we deal with matrices of different sizes at the same time. The cost of the computation of B_i then looks as:

$$T_{B_i} := (N_{pd}^{2d})^3 + ((\frac{h}{H})^{2d} \cdot q^{2d})^2 + (N_{pd}^{2d})^2 + ((\frac{h}{H})^{2d} \cdot q^{2d})^2 = N_{pd}^{6d} + N_{pd}^{4d} + 2(\frac{h}{H})^{4d} \cdot q^{4d}.$$

From this we see, that preconditioner matrix may be computed and assembled in full and serial implementation before the solving process, but it would require a total of $N_{pd}^{6d} + N_{pd}^{4d} + 2(\frac{h}{H})^{4d} \cdot q^{4d}$ operations. The resulting matrix would not be sparse, which would heavily, by order at most, $N_{mesh}^d \cdot N_{pd}^{2d}$ impact **each** iteration of the preconditioned **CG** algorithm 11.

To avoid complexity explosion we can, instead, do it on a local level, for each iteration, keeping the precomputed local matrices A_i, and the restriction matrices C^i, $i = 1, \dots, N_S$. First can be computed during the local primal form's computation. The latter can be set-up during the spline assembly. The multiplication for the $z_{k+1} \leftarrow B r_{k+1}$ performed in a distributed way, as:

$$z_{k+1} \leftarrow \sum_{i=0}^{N_S} C^i A_i^{-1} (C^i)^T r_{k+1},$$

starting from the right and going to the left. The actual parallelisation for coarse and fine grid levels is explained in the next Subsection **3.2.2**.

We introduce temporary matrix and vectors, required for the split computations. Let $T_{i,1}, T_{i,2} \in \mathbb{R}^{N_{sub,f}}$, $T_{i,3} \in \mathbb{R}^N$ for $i = 0, \dots, N_S$. This allows us to perform the multiplication in three steps:

I $T_{i,1} = (C^i)^T \cdot r_{k+1}$

II $T_{i,2} = A_i^{-1} \cdot T_{i,1}$

III $T_{i,3} = C^i \cdot T_{i,2}$,

IV $z_{k+1} = \sum_{i=0}^{N_S} T_{i,3}$,

which is not reducing the algebraic complexity, but allows us to reduce the machine time by distributing the computation of fine grid elements amongst the processes.

Considering, first, the fine grid levels $i = 1, \dots, N_S$. The inverse of a partition A_i^{-1}, would be computed once, during the set up routine, and stored with a designated process, this allows not to compute it every iteration. Then, for the general choice of the basis function, we have:

$$\begin{aligned} T_{ASM,it}^{fine,gen} &:= N_S \left(N_{sub,f} \cdot N + N_{sub,f}^2 + N \cdot N_{sub,f} \right) \\ &= N_S (N_{sub,f}^2 + 2N \cdot N_{sub,f}). \end{aligned} \quad (3.51)$$

Now, choosing the same basis functions on both fine and sub grids, would yield restriction\prolongation matrices C_i, $\quad i = 1, \dots, N_S$ to be a selection matrices, having non-zero values on the main diagonal only, additionally those values would be equal to one. Then:

$$\begin{aligned} T_{ASM,it}^{fine,sel} &:= N_S \left(N_{sub,f} + N_{sub,f}^2 + N_{sub,f} \right) \\ &= N_S (N_{sub,f}^2 + 2N_{sub,f}). \end{aligned} \quad (3.52)$$

Coarse grid level computation comprise an inverse computation for each iteration, but this can be avoided, and inverse can be computed prior. The operation count looks as:

$$\begin{aligned} T^{coarse}_{ASM,it} &:= \big(N_{sub,0} \cdot N + N_{sub,0} \cdot N_{sub,0} + N \cdot N_{sub,0}\big) \\ &= N^2_{sub,0} + 2N_{sub,0} \cdot N. \end{aligned} \tag{3.53}$$

Adding (3.51) and (3.53) gives the general operation count for Additive Schwarz method:

$$\begin{aligned} T^{gen}_{ASM,it} &:= N_S(N^2_{sub,f} + 2N \cdot N_{sub,f}) \\ &+ N^2_{sub,0} + 2N_{sub,0} \cdot N. \end{aligned} \tag{3.54}$$

Adding (3.52) and (3.53) gives the operation count for optimized Additive Schwarz method:

$$\begin{aligned} T^{sel}_{ASM,it} &:= N_S(N^2_{sub,f} + 2N_{sub,f}) \\ &+ N^2_{sub,0} + 2N_{sub,0} \cdot N. \end{aligned} \tag{3.55}$$

Now, substituting (3.45) and (3.46) into (3.54), and expand:

$$\begin{aligned} T^{gen}_{ASM,it} &:= N_S((\frac{N_{mesh} \cdot N^d_{pd}}{N_S})^2 + 2N_{mesh} \cdot N^d_{pd}\frac{N_{mesh} \cdot N^d_{pd}}{N_S}) \\ &+ (\frac{N_{mesh}}{N_S}(\frac{h}{H})^d q^d)^2 + \frac{N_{mesh}}{N_S}(\frac{h}{H})^d q^d \cdot N_{mesh} \cdot N^d_{pd} \\ &= \frac{(N_{mesh} \cdot N^d_{pd})^2}{N_S} + 2(N_{mesh} \cdot N^d_{pd})^2 \\ &+ (\frac{N_{mesh}}{N_S}(\frac{h}{H})^d q^d)(\frac{N_{mesh}}{N_S}(\frac{h}{H})^d q^d + N_{mesh} \cdot N^d_{pd}). \end{aligned} \tag{3.56}$$

And for the optimized case, substituting (3.45) and (3.46) (3.55), and expand:

$$\begin{aligned} T^{sel}_{ASM,it} &:= N_S((\frac{N_{mesh} \cdot N^d_{pd}}{N_S})^2 + 2\frac{N_{mesh} \cdot N^d_{pd}}{N_S}) \\ &+ (\frac{N_{mesh}}{N_S}(\frac{h}{H})^d q^d)^2 + \frac{N_{mesh}}{N_S}(\frac{h}{H})^d q^d \cdot N_{mesh} \cdot N^{2d}_{pd} \\ &= \frac{(N_{mesh} \cdot N^d_{pd})^2}{N_S} + 2N_{mesh} \cdot N^d_{pd} \\ &+ (\frac{N_{mesh}}{N_S}(\frac{h}{H})^d q^d)(\frac{N_{mesh}}{N_S}(\frac{h}{H})^d q^d + N_{mesh} \cdot N^d_{pd}). \end{aligned} \tag{3.57}$$

We also consider special case with $q = 1$

$$\begin{aligned} T^{sel,opt}_{ASM,it} &:= \frac{(N_{mesh} \cdot N^d_{pd})^2}{N_S} + 2N_{mesh} \cdot N^d_{pd} \\ &+ (\frac{N_{mesh}}{N_S}\frac{h}{H})^d(\frac{N_{mesh}}{N_S}(\frac{h}{H})^d + N_{mesh} \cdot N^d_{pd}). \end{aligned} \tag{3.58}$$

The parallel version in the genaral case, would then look as:

$$\begin{aligned} T^{gen}_{ASM,it,par} &:= \frac{N_S(N^2_{sub,f}+2N\cdot N_{sub,f})}{N_P}+(N_P-1)(\frac{N}{N_P}) \\ &+ N^2_{sub,0}+\frac{2N_{sub,0}\cdot N}{N_P}+2(N_P-1)(\frac{N_{sub,0}}{N_P}). \end{aligned} \tag{3.59}$$

And for the optimized case:

$$\begin{aligned} T^{sel}_{ASM,it,par} &:= \frac{N_S(N^2_{sub,f}+2N_{sub,f})}{N_P}+(N_P-1)(\frac{N}{N_P}) \\ &+ N^2_{sub,0}+\frac{2N_{sub,0}\cdot N}{N_P}+2(N_P-1)(\frac{N_{sub,0}}{N_P}). \end{aligned} \tag{3.60}$$

Now, substituting (3.45) and (3.46) into (3.59), and expand:

$$\begin{aligned} T^{gen}_{ASM,it,par} &:= \frac{(N_{mesh}\cdot N^d_{pd})^2}{N_PN_S}+\frac{2(N_{mesh}\cdot N^d_{pd})^2}{N_P} \\ &+ (\frac{N_{mesh}}{N_S}(\frac{h}{H})^dq^d)^2+\frac{N_{mesh}(\frac{h}{H})^dq^d\cdot N_{mesh}\cdot N^d_{pd}}{N_P} \\ &+ (N_P-1)(\frac{N_{mesh}\cdot N^d_{pd}}{N_P})+2(N_P-1)\frac{N_{mesh}(\frac{h}{H})^dq^d}{N_P}. \end{aligned} \tag{3.61}$$

And for the optimized case, substituting (3.45) and (3.46) into (3.60), considering $q=1$, and expand:

$$\begin{aligned} T^{sel}_{ASM,it,par} &:= \frac{(N_{mesh}\cdot N^d_{pd})^2}{N_PN_S}+\frac{2N_{mesh}\cdot N^d_{pd}}{N_P} \\ &+ (\frac{N_{mesh}}{N_S}(\frac{h}{H})^d)^2+\frac{N_{mesh}(\frac{h}{H})^d\cdot N_{mesh}\cdot N^d_{pd}}{N_P} \\ &+ (N_P-1)(\frac{N_{mesh}\cdot N^d_{pd}}{N_P})+2(N_P-1)\frac{N_{mesh}(\frac{h}{H})^d}{N_P}. \end{aligned} \tag{3.62}$$

3.2.2 Computation of the restriction\prolongation matrices

First, we can establish the coarse and fine grid total number of elements ratio:

$$N^{coarse}_{mesh}=N_{mesh}\cdot(\frac{h}{H})^d,$$

for d-dimensional case. Next, we establish the total number of functions in use on the elements of the coarse mesh:

$$N^{coarse}:=N^{coarse}_{mesh}\cdot q^d.$$

This allows us to estimate the amount of non-zero entries for matrices C^i as $(\frac{h}{H})^d\cdot q^d$, effectually the storage size for each local C^i.

Next, we will show in detail the process of the parallel implementation of the preconditioner routines. Here, we should note, that if we are reusing the same basis functions and the same space partition for V_{hp} and V^i_{hp} given in (3.36), then the restriction matrices would be mainly sparse, for the piecewise constant basis function, with values of ones for the entries which correspond to the values of the global basis functions with support on both fine and coarse grids. However, for the choice of the local basis functions with higher polynomial degrees, the restriction matrix would be dense for the coarse grid and associated partition V^0_{hp}. In order to calculate such a matrix we would need to solve a system shown before in (3.49):

$$\overrightarrow{d} = c^T C^i$$

For better understanding of this process, let us analyse the fine and coarse grids in a real example on a one dimensional mesh, shown in figures (3.4,3.5).

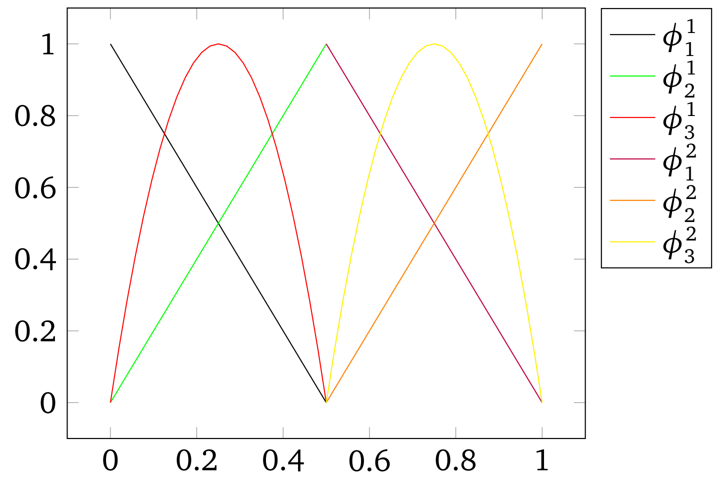


Figure 3.4: Fine grid mesh in 1-d with local basis functions with polynomial degree p up-to 2. Two elements on a grid.

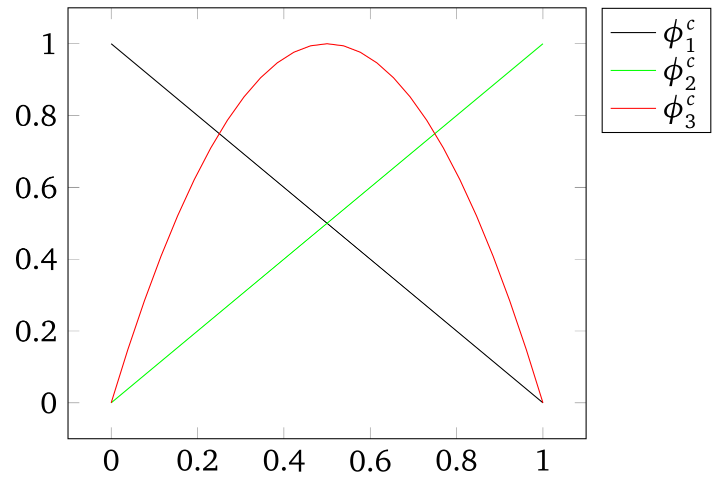


Figure 3.5: Coarse grid mesh in 1-d with local basis functions with polynomial degree p up-to 2. One element on a grid.

We need to clarify, that each element of the grid uses local coordinate system. In order, to transform the coordinates to global coordinates system, we use, $x = F_1(t)$, $x = F_2(t)$ and $x = F_c(t)$, for each subspace V^1_{hp}, V^2_{hp} and the space on a coarse grid V^0_{hp} respectively, with t being a local coordinates, independent in each space\subspace.

As for the local basis functions indices, we use super script, to specify the element of the fine grid, where the local basis function is non-zero, index is in range $1,\dots,N_{mesh}$. Sub script is in range $1,\dots,N_{loc}$ and specifies the number of the local basis functions on the aforementioned element. As, the chosen local basis functions, exist everywhere, but by construction restricted to have non-zero value only on one element, on their grids we get the following equations, for the first local basis function, on a coarse grid:

$$\begin{aligned}\phi_1^c(F_c^{-1}(x)) &= C_{1,1}^1\phi_1^1(F_1^{-1}(x)) + C_{1,2}^1\phi_2^1(F_1^{-1}(x)) + C_{1,3}^1\phi_3^1(F_1^{-1}(x)) \\ &+ C_{2,1}^1\phi_1^2(F_2^{-1}(x)) + C_{2,2}^1\phi_2^2(F_2^{-1}(x)) + C_{2,2}^1\phi_3^2(F_2^{-1}(x)),\end{aligned}$$

for $C_{j,k}^i$, $i = 1,\dots,N^{coarse}$ - the number of coarse grid local basis funcitons, $j = 1,\dots,N_{mesh}$ - number of elements of the fine grid and $k = 1,\dots,N_{loc}$ - number of the fine grid basis functions.

This can be extended on all coarse grid local basis functions:

$$\begin{aligned}\phi_i^c(F_c^{-1}(x)) &= \sum_{k=1}^{N_{loc}} C_{1,k}^i\phi_k^1(F_1^{-1}(x)) \\ &+ \sum_{k=1}^{N_{loc}} C_{2,k}^i\phi_k^2(F_2^{-1}(x)),\end{aligned} \tag{3.63}$$

for $i = 1,\dots,N^{coarse}$.

As we only need to calculate the local basis function's value on the element, where it is a non-zero, we split the (3.63) in parts, for each fine grid element, and use the local to that element coordinates:

$$\begin{aligned}\phi_i^c|_{K_1}(F_c^{-1}(F_1(t))) &= \sum_{k=1}^{N_{loc}} C_{1,k}^i\phi_k^1(t) \\ \phi_i^c|_{K_2}(F_c^{-1}(F_2(t))) &= \sum_{k=1}^{N_{loc}} C_{2,k}^i\phi_k^2(t),\end{aligned} \tag{3.64}$$

for $i = 1,\dots,N^{coarse}$, K_1, K_2 represent the fine grid elements, and t is a local coordinate, independent and non-consecutive per element.

A linear system, arising from the (3.64), can be obtained, by introducing the interpolation nodes $t_l \in [-1,1]$, with $l >= N^{coarse}$, and setting

$$A_{i,l} = \phi_i^c(F_c^{-1}(F_1(t_l))), \quad i,l = 1,\dots,N^{coarse},$$

$$B_{k,l} = \phi_k^1(t_l), \quad k = 1,\dots,N_{loc}, \quad l = 1,\dots,N^{coarse},$$

and $\mathscr{C}_{i,k} = C_{0,k}^i$:

$$A_{i,l} = \sum_{k=1}^{N_{loc}} \mathscr{C}_{i,k}\cdot B_{k,l},$$

So, to get the local restriction matrix $\mathscr{C} = (\mathscr{C}_{i,k})$, we need to evaluate:

$$\mathscr{C} = AB^{-1}$$

It is clear, that to obtain a restriction\prolongation matrix C in this case shown in figures 3.4 and 3.5, we need to solve the following system of equations:

$$\begin{aligned}
\phi_1^c(x^c)|_{K_1} &= C_{1,1}\phi_1(x)+C_{1,2}\phi_2(x)+C_{1,3}\phi_3(x)\\
\phi_1^c(x^c)|_{K_2} &= C_{1,4}\phi_4(x)+C_{1,5}\phi_5(x)+C_{1,6}\phi_6(x)\\
\phi_2^c(x^c)|_{K_1} &= C_{2,1}\phi_1(x)+C_{2,2}\phi_2(x)+C_{2,3}\phi_3(x)\\
\phi_2^c(x^c)|_{K_2} &= C_{2,4}\phi_4(x)+C_{2,5}\phi_5(x)+C_{2,6}\phi_6(x)\\
\phi_3^c(x^c)|_{K_1} &= C_{3,1}\phi_1(x)+C_{3,2}\phi_2(x)+C_{2,3}\phi_3(x)\\
\phi_3^c(x^c)|_{K_2} &= C_{3,4}\phi_4(x)+C_{3,5}\phi_5(x)+C_{2,6}\phi_6(x), \qquad (3.65)
\end{aligned}$$

where, $C_{j,i}, \quad j=1,\dots,N^{coarse}, \quad i=1,\dots,N_{mesh}\cdot N_{pd}^d$ are the elements of a restriction matrix, $\phi_i, \quad i=1,\dots,N_{mesh}\cdot N_{pd}^d$ are the local basis function on a fine grid and $\phi_j^c, \quad j=1,\dots,N^{coarse}$ are the local basis functions on a coarse grid. The formulae (3.65) can be, for all $j=1,\dots,N^{coarse}$, then written as:

$$\sum_{e=1}^{N_e}\phi_j^c(x^c)|_{K_e}=\sum_{i}^{N_{loc}}C_{j,i}\cdot\phi_i(x),$$

here we know all the coarse grid basis functions ϕ^c values on the corresponding edges $e=1,\dots,N_e$, and all the global basis functions ϕ on all elements $i=1,\dots,N_{loc}$, with $N_{loc}=N_{mesh}\cdot N_{pd}^d$. This can be simplified, by interpolating the local basis functions from a fine grid to coarse one, to find a value of the coarse grid basis function:

$$\sum_{e=1}^{N_e}R_0^T\phi_i(\Phi_{K_e}(x^c))=\sum_{i}^{N_{loc}}C_{j,i}\cdot\phi_i(x),$$

where $j=1,\dots,N^{coarse}$ and local coarse grid coordinates transformation $\Phi_{K_e}(x^c)$. The interpolation on a coarse grid, can be performed by applying Newton formulae. The resulting system of equations can be then solved using LU decomposition, to find the unknowns $C_{j,i}, \quad j=1,\dots,N^{coarse}, \quad i=1,\dots,N_{loc}$.

To compute the restriction matrix $C_i\in\mathbb{R}^{(N_{mesh}\cdot N_{pd}^d)\times(N^{coarse})}$, one would need to solve $N_{mesh}\cdot N_{pd}^d$ system of equations of N^{coarse} unknowns.

As the next step, we should determine the communication cost for each iteration of the preconditioned **CG** method.

3.2.3 Iteration count

Stability of the preconditioner in question was proven in [5] and involves introduction of additional coefficient:

$$C_\#^2 := C_\sigma\frac{H}{h}\frac{N_{pd}^2}{q},$$

for $C_\sigma>0$ being a mesh and polynomial degree - independent constant, and the spectral condition number estimate:

$$\kappa(BA)\leq C_\#^2(N_H+2),$$

with N_J introduced to denote the maximum number of adjacent partitions that any given subdomain in the partition $\mathcal{T}_H$ might posses.

It was shown in Corollary 3.1.2, namely in (3.33):

$$k \geq \ln(\frac{2}{\epsilon})\frac{(C_{\#}\sqrt{(N_H+2)}+1)}{2}$$

Hence, the number of steps k for the preconditioned **CG** method, for desirable $\epsilon > 0$ can be found with:

$$k = \lceil \ln(\frac{2}{\epsilon})\frac{(C_{\#}\sqrt{(N_J+2)}+1)}{2} \rceil \tag{3.66}$$

3.2.4 Parallelization techniques

In order to properly explain the optimal parallel implementation, we would start with domain partitioning.

For defined domain finite decomposition Ω (1.3), with space of rectangular decomposition V_{hp} (1.8), define sub-grid V_{Hq} as in (3.38), with every logical partition element of V_{hp} smaller than the logical partition element of V_{Hq}. Every element of V_{Hq} contains at least two elements of V_{hp}. Furthermore, for each element of V_{Hq}, define spaces of finite rectangular decomposition V_{hp}^i as in (3.36). The arbitrary case can be seen on figure (3.6):

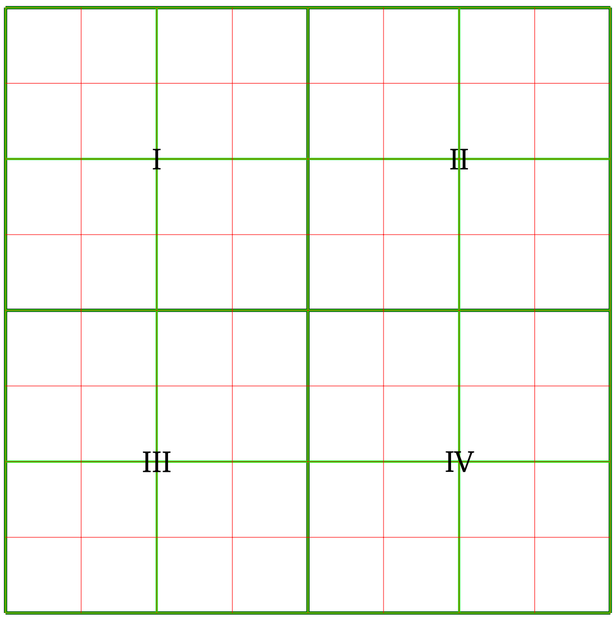


Figure 3.6: Domain decomposition. Black - subgrid. Green - coarse grid. Red - fine grid.

In figure 3.7 we zoom in on subgrid element I we can see that numbering of the coarse grid elements within the subgrid element is not consecutive. Let I be the set corresponding to the first top left subgrid element. The set

$$I := \{E_1^c, E_2^c, E_5^c, E_6^c\}, \tag{3.67}$$

with sets of neighbouring coarse grid elements being

$$\begin{aligned} I^{\text{neigh}(II)} &:= & I \cap II = & \{E_3^c, E_7^c\}, \\ I^{\text{neigh}(III)} &:= & I \cap III = & \{E_9^c, E_{10}^c\}, \\ I^{\text{neigh}(IV)} &:= & I \cap IV = & \{E_{11}^c\}. \end{aligned}$$

From the observed picture we conclude, that "internal" coarse grid element E_1^c has no interaction outside of its subgrid, and require no data transfer for global system matrix computation, nor for solver, nor for pre-conditioner step in chosen CG solver.

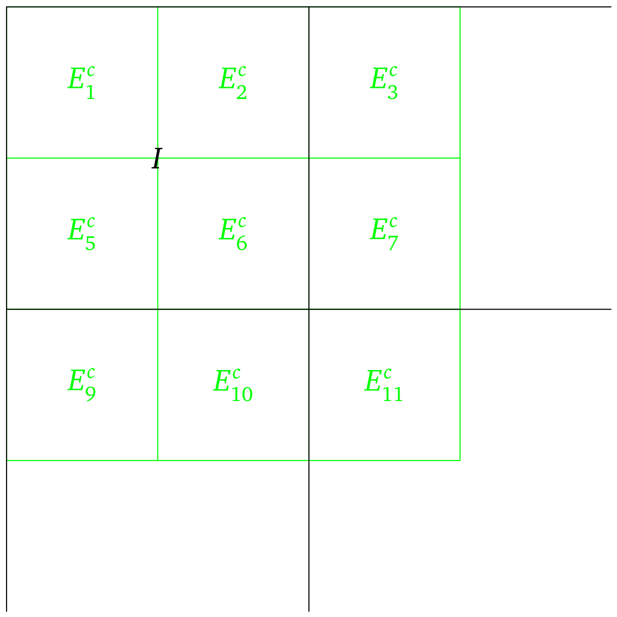


Figure 3.7: Domain decomposition. Subgrid element I with coarse grid elements and neighbourhood. Black - subgrid. Green - coarse grid.

Here, we zoom in further in figure 3.8, to coarse grid elements E_2^c and E_3^c, which belong to different subgrid elements I and II correspondingly and require data transfer for computation. Let $X(E_2^c)$ and $X(E_3^c)$ be the sets of all fine grid elements contained within E_2^c and E_3^c correspondingly, as in:

$$\begin{aligned} X(E_2^c) &= \{E_3^f, E_4^f, E_{11}^f, E_{12}^f\}, \\ X(E_3^c) &= \{E_5^f, E_6^f, E_{13}^f, E_{14}^f\}. \end{aligned}$$

Corresponding set $X(E_2^c)^{\text{neigh}(X(E_3^c))}$ of neighbouring elements of sets $X(E_2^c)$ and $X(E_3^c)$ is then:

$$X(E_2^c)^{\text{neigh}(X(E_3^c))} := X(E_2^c) \cap X(E_3^c) := \{E_4^f, E_{12}^f, E_5^f, E_{13}^f\}$$

Due to the nature of the chosen basis functions, only the functions on shared edges between fine grid elements require data transfer for parallel computation.

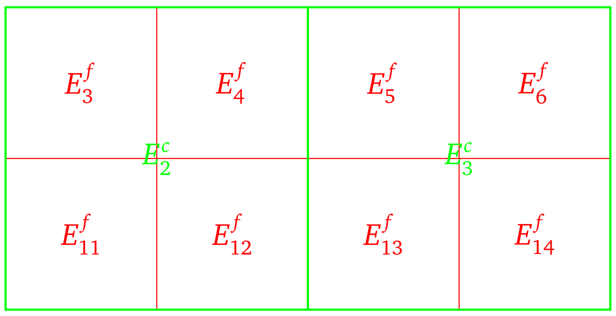


Figure 3.8: Domain decomposition. Coarse grid elements E^c_2 and E^c_3 from neighbouring subgrids elements. With fine grid elements contained within aforementioned. Green - coarse grid. Red - fine grid.

This allows us to choose suitable uniform grid domain discretisation. Furthermore, the optimal subgrid restraints irrespective of the geometry and dimension of the problem would be to develop a subgrid, such that a number of interacting interfaces of the corresponding coarse grid is minimal. Say for interface between I and II to have the smallest possible cardinality of $|I \cap II|$.

First, we discuss the case for uniform grid of quadrilateral elements. Here, we should introduce the parallel version of the Conjugate Gradient method with Additive Schwarz as preconditioner. We are going to expand the algorithm given in 11.

Consider the problem:

$$Ax = b$$

where A is a system matrix arising from (2.1), b is a right hand side, and x is a vector of unknowns. We set

$$N_{DoF} := N_{mesh} \cdot N^2_{pd}$$

being the degrees of freedom. Additionally we define number of threads (parallel processes) which will be used in computation $N_p \geq 1$, and number of partitions

$$N_S \leq N_p$$

for Additive Schwarz method.

We do not require to store whole vectors and matrices during the parallel computation on all threads. We can instead store the spread data in chunks of size

$$N_C := N_{DoF}/N_p,$$

where we round the N_C to the closest integer. Chunks could be identified with corresponding thread (process) ID:$\# \in 0, \dots, N_p - 1$. This will require to set auxiliary boundaries per thread (process) for parallel computation. We will design boundaries for the vectors and matrices, used in parallel **CG** as a function of a thread (process) ID.

$$n_{lC}(\#) = \begin{cases} \# \cdot N_C & \text{if} \quad \# \geq 1; \\ 0 & \text{if} \quad \# = 0 \end{cases} \tag{3.68}$$

As lower boundary for the $\#$-th thread (process).

$$n_{uC}(\#) = \begin{cases} \min((\# + 1) \cdot N_C, N_{mesh} \cdot N_{pd}^d) & \text{if} \quad \# < N_p - 1; \\ N_{mesh} \cdot N_{pd}^d & \text{if} \quad \# = N_p - 1. \end{cases} \tag{3.69}$$

As upper boundary for the $\#$-th thread (process). We note that we need the $\# = N_p - 1$ condition of the (3.69) explicitly, for the cases when the $N_{mesh} \cdot N_{pd}^d$ is an odd integer and can result in $\min((\# + 1) \cdot N_C, N_{mesh} \cdot N_{pd}^d)$ being less than the $N_{mesh} \cdot N_{pd}^d$.

Analysing the difference between serial and parallel matrix-vector multiplication. As an example we take line 6 of algorithm 11. Set $q := q_k$ and $p := p_k$. End-to-end multiplication, then looks as:

$$q_i = \sum_{j=1}^{N_{DoF}} A_{i,j} p_j, \quad i = 1, \ldots, N_{DoF},$$

whereas parallel multiplication on thread(process) with ID id will look as:

$$q_i = \sum_{j=n_l(id)}^{n_u(id)} A_{i,j} p_j, \quad i = 1, \ldots, N_{DoF}.$$

Additionally we set the auxiliary boundaries for restriction\prolongation matrices C^i for corresponding spaces $V_{hp}^i, \quad i \in 1, \ldots, N_S$. First we set number of matrices per thread (process)

$$N_{CS} := N_S / N_p.$$

We define the boundaries per thread (process), in the same way as (3.68,3.69).

$$n_{lCS}(\#) = \begin{cases} \# \cdot N_{CS} & \text{if} \quad \# \geq 1; \\ 0 & \text{if} \quad \# = 0 \end{cases} \tag{3.70}$$

As lower boundary for the $\#$-th thread (process).

$$n_{uCS}(\#) = \begin{cases} \min((\# + 1) \cdot N_{CS}, N_{mesh} \cdot N_{pd}^d) & \text{if} \quad \# < N_p - 1; \\ N_{mesh} \cdot N_{pd}^d & \text{if} \# = N_p - 1. \end{cases} \tag{3.71}$$

As upper boundary for the $\#$-th thread (process).

Here, we should note, that although only the restriction matrices, which are necessary on a thread (process) for computation, would be computed on each thread (process), the matrix corresponding for a coarse partition V_{Hq} would only be computed on a thread (process) with ID 0.

Before proceeding, we would analyse this operation further, as both, system matrix and preconditioner matrix are distributed between processes. From algorithm 11, line 6:

$$q_k \leftarrow A_{\{i\}} p_k.$$

Here, $A_{\{i\}}$ is a partition $\{i\}, i = 1, \ldots, N_p$, of a system matrix A_h, computed and distributed amongst N_p process. Vectors q_k and p_k would require appropriate partitioning with $\{i\}, i = 1, \ldots, N_p$ to distribute them among processes.

Later q_k, which is distributed amongst N_p processes used in residual direction vector computation, from (11), $r_{k+1} \leftarrow r_k - \alpha_k q_k$. And this new vector r_{k+1} is used in computation of preconditioner expansion vector $z_{k+1} \leftarrow Br_{k+1}$. At the same time preconditioner B requires expansion to three operations (3.2.1). This brings additional restraints onto the boundaries design (3.68, 3.69, 3.70, 3.71). Matrix B can be rewritten as:

$$B = C_0^T \cdot A_0^{-1} \cdot C_0 + \Sigma_{k=1}^{N_S} C_k^T \cdot A_j^{-1} \cdot C_k$$

Introducing partitioning $\{i\}$,$i = 1, \ldots, N_p$, same as for system matrix A_h, for distribution of preconditioner matrix among the processes, we assume, that subgrid-to-fine restriction\prolongation matrices C_k are stored fully on corresponding processes. Whereas coarse-to-fine restriction\prolongation matrices C_C would be computed and stored distributively across processes:

$$B^{\{i\}} = (C_C^{\{i\}})^T \cdot (A_C^{\{i\}})^{-1} \cdot C_C^{\{i\}} + \Sigma_{k \in \{i\}} C_k^T \cdot A_{\{i\}}^{-1} \cdot C_k.$$

Here, we should note, that the storage size allocated for any chunks of matrices would be equal to the whole matrix, indexing would also be preserved as in original matrices. Partition of corresponding vectors would be the same. Hence, the steps to compute z_{k+1}, would require similar partitioning of resulting vector, transfer of intermediate results $z_{k+1}^{\{i\}}$ from local processes and reduction (appropriate communication and summation of intermediate result) of the results z_{k+1} from all processes.

$$z_{k+1}^{\{i\}} = (C_C^{\{i\}})^T (A_C^{\{i\}})^{-1} (C_C^{\{i\}} \cdot r_{k+1}^{\{i\}})) + \Sigma_j^{N_S} C_j^T \cdot (A_{\{i\}}^{-1} \cdot (C_j r_{k+1}^{\{i\}})). \tag{3.72}$$

Now, to illustrate the actual split among processes, we would separate (3.72) into two parts:

$$z_{k+1}^{\{i\}} = \quad (C_C^{\{i\}})^T (A_C^{\{i\}})^{-1} (C_C^{\{i\}} \cdot r_{k+1}^{\{i\}})) \tag{3.73}$$

$$+ \quad \Sigma_j^{N_S} C_j^T \cdot (A_{\{i\}}^{-1} \cdot (C_j r_{k+1}^{\{i\}})) \tag{3.74}$$

Matrix $(A_{\{i\}})^{-1}$, $i = 1, \ldots, N_{mesh}^{coarse}$ will be distributed among processes, according to the subgrid elements' configuration, which they are supported by. Here, we should note, that original matrix A is a sparse matrix, with non-zero elements on main diagonal.

We will illustrate computation z_{k+1} on a particular case, for 4 processes and coinciding subgrid partition (3.6) with 4 elements I, II, III, IV, coarse grid (3.7) with elements E_i^C, $i = 1, \ldots, N_{mesh}^{coarse}$, $N_{mesh}^{coarse} = 16$ and fine grid (3.8) with elements E_i^f, $i = 1, \ldots, N_{mesh}^{fine}$, $N_{mesh}^{fine} = 64$.

We first want to show subgrid-to-fine grid part of (3.74) computation, as it would require no data transfer. We enumerate restriction/prolongation matrices C according to the fine grid elements' (3.8) enumeration. Furthermore, we can compute selection matrices C_j, according to the subgrid's process distribution.

The operation we need to perform is:

$$z_{k+1} = \Sigma_j^{N_S} C_j^T \cdot (A_{\{i\}}^{-1} \cdot (C_j r_{k+1}^{\{i\}}).$$

Each prolongation matrix, in our chosen domain partition, would be represented as a selection matrix, due to the fact, that subgrid's elements are supported on fine grid elements inside them. This allows us to untangle the Σ notation and perform multiplication of corresponding selection matrices C_j with parts of the vector r_{k+1}. Just to demonstrate how the multiplication is done, say on process I, we choose C^1 and C^9 which are on process I, and corresponding part $r^{\{1\}}_{(k+1),1}$ and $r^{\{1\}}_{(k+1),9}$ of the vector r_{k+1}:

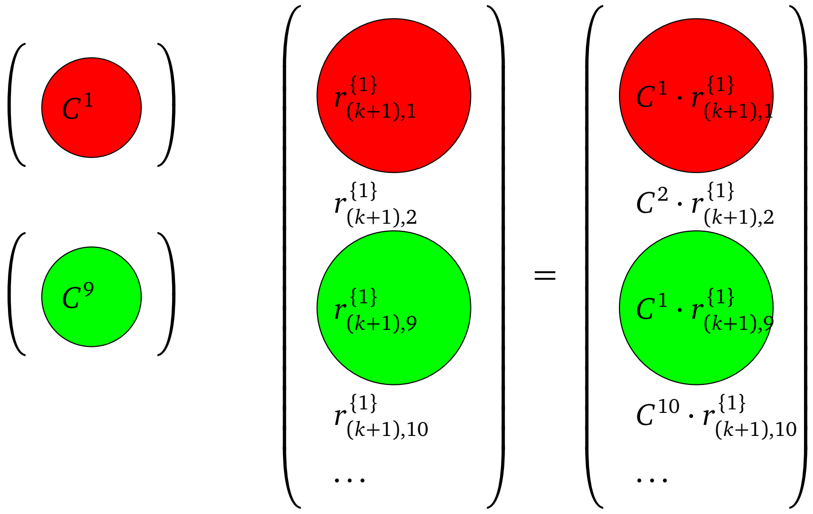

C^1
C^9
$r^{\{1\}}_{(k+1),1}$
$r^{\{1\}}_{(k+1),2}$
$r^{\{1\}}_{(k+1),9}$
$r^{\{1\}}_{(k+1),10}$
$\dots$
$=$
$C^1 \cdot r^{\{1\}}_{(k+1),1}$
$C^2 \cdot r^{\{1\}}_{(k+1),2}$
$C^1 \cdot r^{\{1\}}_{(k+1),9}$
$C^{10} \cdot r^{\{1\}}_{(k+1),10}$
$\dots$

Figure 3.9: Multiplication of matrices C^1 and C^9 with parts of vector r_{k+1}. Results are stored appropriately in vector t on the r.h.s

Then, the full multiplication $t^{\{i\}} := C_j r^{\{i\}}_{k+1}$, can be illustrated on process I:

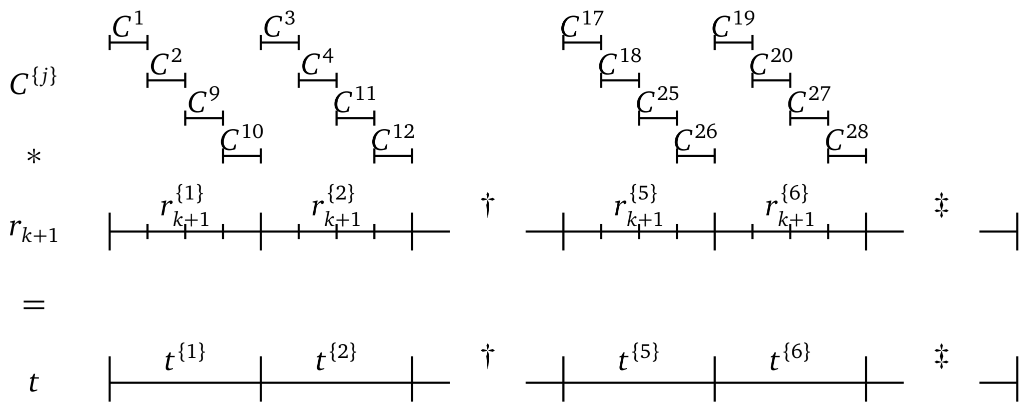


Figure 3.10: Preconditioner coarse-to-fine $t^{\{i\}} := C_j r^{\{i\}}_{k+1}$ computation on process I. Note, that $\dagger$ and $\ddagger$ are standing for chunks and parts $3, 4$ and $7, \dots, 16$ correspondingly, and which are filled with zeros and are not in use on the current process.

The shown domain decomposition (3.6) then will require distribution of chunks of matrix A among processes. For process I, we need only chunks which arise from elements in set I (3.67). For simplicity we enumerate them with accordance to the coarse element number, so then, the first process I would store the following four chunks $A^{-1}_{\{1\}}, A^{-1}_{\{2\}}, A^{-1}_{\{5\}}, A^{-1}_{\{6\}}$. The chunks on the second process II would then be enumerated as $A^{-1}_{\{3\}}, A^{-1}_{\{4\}}, A^{-1}_{\{7\}}, A^{-1}_{\{8\}}$. Etc.

From the matrix vector multiplication, we know, that corresponding chunk of the matrix, is going to be multiplied with corresponding part of the vector. Hence, we can distribute only necessary parts of the vector t among the processes, and enumerate them in the similar way. For multiplication of the matrix A^{-1} with the vector t on process I we would require parts $t^{\{1\}}, t^{\{2\}}, t^{\{5\}}, t^{\{6\}}$. We need no data transfer, as all necessary parts are stored on the processes, where multiplication is performed.

Next step in preconditioner is the multiplication of A^{-1} and t. On process I it can be written as:

$$l^I := A^{-1}_{\{1\}} \cdot t^{\{1\}} + A^{-1}_{\{2\}} \cdot t^{\{2\}} + A^{-1}_{\{5\}} \cdot t^{\{5\}} + A^{-1}_{\{6\}} \cdot t^{\{6\}} \tag{3.75}$$

Or in alternative notation:

$$l^I := A^{-1}_{\{I\}} \cdot t^{\{I\}}$$

And can be illustrated as:

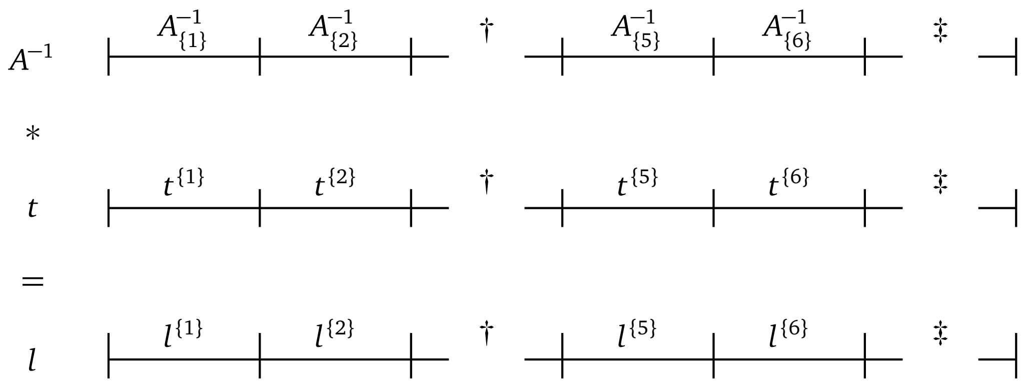


Figure 3.11: Preconditioner step $l^i := (A^{\{i\}})^{-1} \cdot t^{\{i\}}$ computation on process I. Note, that † and ‡ are standing for chunks and parts $3,4$ and $7,\ldots,16$ correspondingly, and which are filled with zeros and are not in use on the current process.

The last step of preconditioner computation is $z^{\{i\}}_{k+1} := (C_j)^T l^{\{i\}}$. The process is similar to the one illustrated in (3.10):

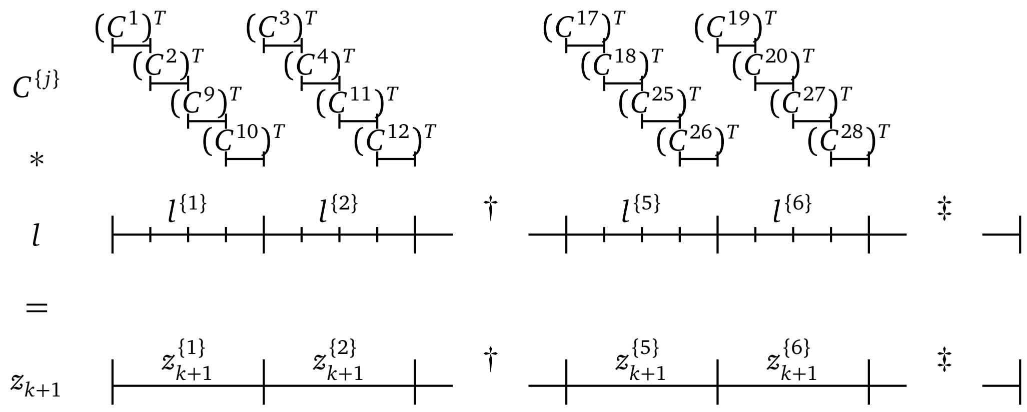


Figure 3.12: Preconditioner step $z^{\{i\}}_{k+1} := (C_j)^T l^{\{i\}}$ computation on process I. Note, that † and ‡ are standing for chunks and parts $3,4$ and $7,\ldots,16$ correspondingly, and which are filled with zeros and are not in use on the current process.

Finally, the coarse-to-fine part of the equation (3.73) requires computation. The coarse-to-fine prolongation matrix C_0 is computed and stored in chunks across all processes, with the account of coarse grid elements. They will be enumerated similarly. Using the same partition as in (3.75):

$$t^I := C_0^{\{1\}} \cdot r_{k+1}^{\{1\}} + C_0^{\{2\}} \cdot r_{k+1}^{\{2\}} + C_0^{\{5\}} \cdot r_{k+1}^{\{5\}} + C_0^{\{6\}} \cdot r_{k+1}^{\{6\}}$$

Or in alternative notation:

$$t^I := C_0^{\{I\}} \cdot r_{k+1}^{\{I\}}$$

And can be illustrated as:

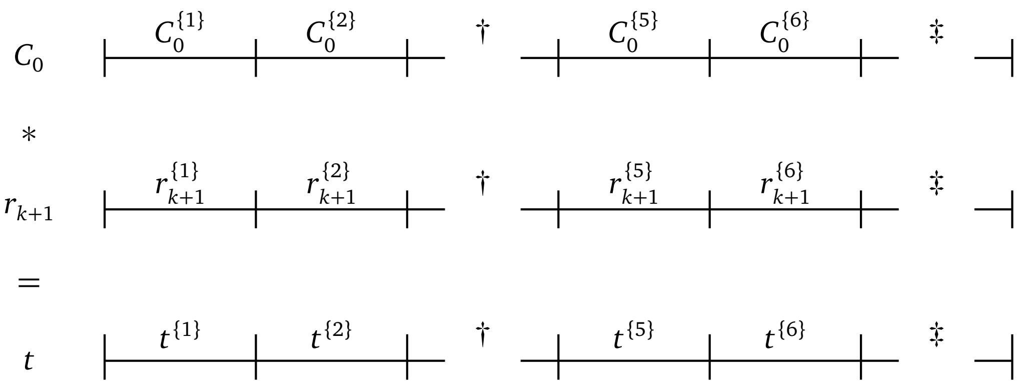


Figure 3.13: Preconditioner step $t^i := C_0^{\{i\}} \cdot r_{k+1}^{\{i\}}$ computation on process I. Note, that † and ‡ are standing for chunks and parts $3, 4$ and $7, \dots, 16$ correspondingly, and which are filled with zeros and are not in use on the current process.

Due to the fact, that inverse of the full matrix A_0 would be dense, it would be very expensive to compute inverse and communicate it on all processes. Additionally, non sparse structure disallows convenient chunk-to-part matrix vector multiplication. Instead, we would communicate the result of $t^i := C_0^{\{i\}} \cdot r_{k+1}^{\{i\}}$, from each process $\{i\}$ into the t^{core}, which would be stored at our core process. Matrix vector multiplication $l := A_0^{-1} \cdot t^{core}$ would be performed on the core process, and result would be communicated to corresponding processes. This can be illustrated as such:

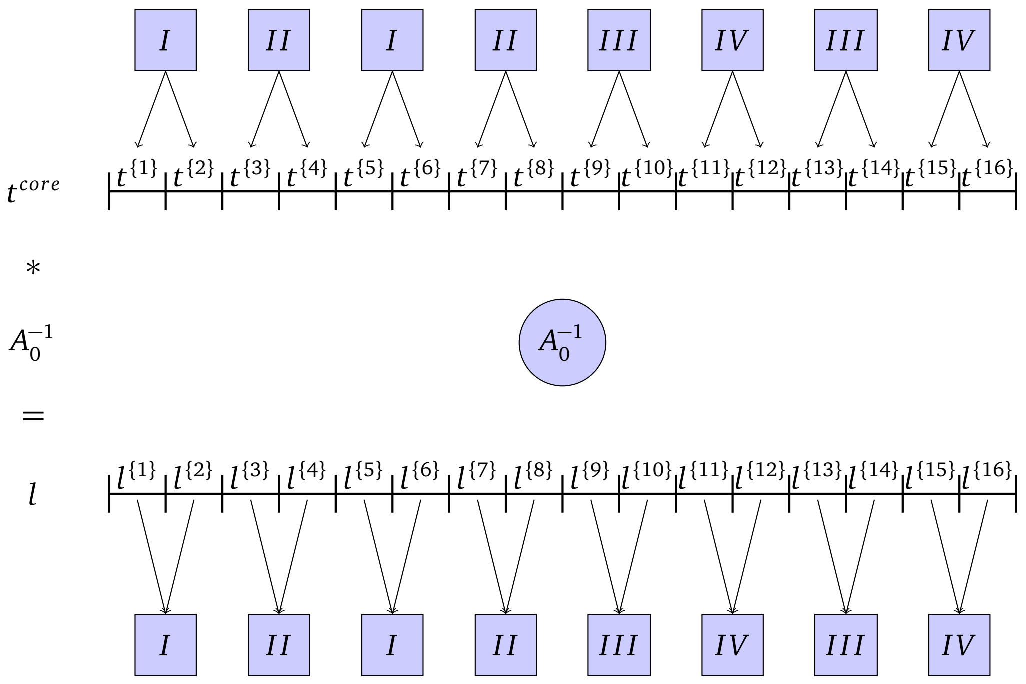

I
II
I
II
III
IV
III
IV
t^{core}
$t^{\{1\}}$ $t^{\{2\}}$ $t^{\{3\}}$ $t^{\{4\}}$ $t^{\{5\}}$ $t^{\{6\}}$ $t^{\{7\}}$ $t^{\{8\}}$ $t^{\{9\}}$ $t^{\{10\}}$ $t^{\{11\}}$ $t^{\{12\}}$ $t^{\{13\}}$ $t^{\{14\}}$ $t^{\{15\}}$ $t^{\{16\}}$
$*$
A_0^{-1}
A_0^{-1}
$=$
l
$l^{\{1\}}$ $l^{\{2\}}$ $l^{\{3\}}$ $l^{\{4\}}$ $l^{\{5\}}$ $l^{\{6\}}$ $l^{\{7\}}$ $l^{\{8\}}$ $l^{\{9\}}$ $l^{\{10\}}$ $l^{\{11\}}$ $l^{\{12\}}$ $l^{\{13\}}$ $l^{\{14\}}$ $l^{\{15\}}$ $l^{\{16\}}$
I
II
I
II
III
IV
III
IV

Figure 3.14: Preconditioner step for $l := A_0^{-1} \cdot t^{core}$ computation. Data communication to core process for t^{core} precedes the matrix multiplication, hence shown prior to multiplication. Communication of vector l is not simultaneous from core to all processes.

And the last step is $z_{k+1}^{\{i\}} := (C_0)^T l^{\{i\}}$, which is performed in similar way to (3.13).

Vector z_{k+1} is then reduced (communicated and summed appropriately) among processes, ensuring that corresponding part is stored on the process. This ensures that all subsequent steps in the preconditioned parallel Conjugate Gradient method's algorithm do not require full vector on any of the processes, henceforth minimizing the communication to sole scalar values.

Now, taking into account that all the computations, i.e dot product, addition and subtraction, will be performed on each thread (process) separately, division can not be done in such a way. Hence, we would have to introduce temporary variables to store intermediate results, transfer (reduce) all of the results to core thread (process) . Division operation would be then performed on the core thread (process), and result would be broadcast to all other threads (processes). According to the aforementioned the coarse level of the Additive Schwarz in fact creates biggest bottleneck and does not allow effective parallelization. In this work we tackle this by using the lowest polynomial approximation order possible. Nevertheless, as shown further in the tables 3.2-3.5, we can overcome this bottleneck in Additive Schwarz by introducing the Multigrid Method as a coarse level solver.

3.3 Multigrid method as a coarse level solver

Here we are going to present a modification of the Additive Schwarz Method. Recalling (3.50):

$$B_{ASM} = \sum_{i=1}^{N_S} C^i \bar{A}_i^{-1} (C^i)^T + C^0 \bar{A}_{Hq}^{-1} (C^0)^T,$$

As we have noted in the previous section, we would require computation of matrix $\bar{A}_{Hq}$, computation of the corresponding inverse, or solving the system. Instead, we can use the Multigrid Method (**MG**), then modified preconditioner looks as:

$$B_{ASM,MG} = \sum_{i=1}^{N_S} C^i \bar{A}_i^{-1} (C^i)^T + C^0 B_{MG} (C^0)^T, \tag{3.76}$$

with the coarse level solver being:

$$B_{MG,coarse} = C^0 B_{MG} (C^0)^T \tag{3.77}$$

Here, we can note, that, although, inverse of the system matrix would be the best choice in terms of the accuracy and convergence, approximation is cheaper in terms of computation time. Additionally, we should point out, that the presented **MG** method is not a Krylov Subspace Method. Main difference is that to solve a system of equations as $Ax = b$ any Krylov Subspace method requires only defined matrix A, and right hand side vector b [46]. Additional information, such as the nature of (partial-)differential equation to solve, choice of the basis function, space discretisation method etc do not matter for successful implementation of Krylov Subspace method. Whereas **MG** method requires extra information about the problem itself, e.g. in the case of Poisson equation, string relation between eigenfunctions of the iteration matrix and the chosen mesh.

In this section, to avoid confusion, we will refer to the polynomial approximation degree as $p_{MG,k}$. We are also considering the case with polynomial approximation degree to be uniformly $p_{MG,k} = 1$, but start with performing the analysis for general positive $p_{MG,k}$.

In addition we define **the number of Multigrid levels** as N_{MG}.

Define the Multigrid Method, according to [14] and [13], assume that we are given a sequence of finite-dimensional vector spaces

$$V_k := \{v \in L^2(\Omega) \quad : v\,|_K \circ T_K \in \mathbb{M}^{p_{MG,k}}(\bar{K}) \quad \forall K \in \mathscr{T}_{H,k}\}, \quad k = 0, \dots, N_{MG} \tag{3.78}$$

where $\mathscr{T}_{H,k}$ is a quasi-uniform partition as defined earlier (1.3) of a set $\{\mathscr{T}_{H,k}\}_{k=1}^{N_{MG}}$, and $\mathbb{M}^{p_{MG,k}}(\bar{K})$ is a space of polynomials $\mathscr{P}_{p_{MG,k}}, \quad p_{MG,k} \geq 1$, (1.5) in case of a $\bar{K} := \triangle$ being a reference simplex, otherwise a space of polynomials $\mathscr{Q}_{p_{MG,k}}, \quad p_{MG,k} \geq 1$ (1.6) in case of a reference hypercube $\bar{K} := \square$. We consider $p_{MG,k}$ to be uniform on $\mathscr{T}_{H,k}$, where $k = 1, \dots, N_{MG}$. We consider our finite decomposition to comply with $H_k := \max_{K \in \mathscr{T}_{H,k}} H_K$, for all diameters H_K on each element K, with $k = 0, \dots, N_{MG}$. By suitably choosing the sequence $\{\mathscr{T}_{H,k}, V_k\}_{k=0}^{N_{MG}}$, we can obtain $h-$, $p-$ and $hp-$multigird methods as defined in [3]. We also should note, according to [3, (2.1)], that sequence of spaces (3.78) would be nested

$$V_0 \subseteq V_1 \subseteq V_2 \subseteq \cdots \subseteq V_{N_{MG}}.$$

In order to comply with additive Schwarz method's convergence framework, we restrict the modification with inherited space discretization. We also have

$$V_0 \equiv V_{Hq} \tag{3.79}$$

This allows to represent the primal bilinear **DG** form (2.2) on each subspace. Designating primal bilinear form as

$$A_0(\cdot,\cdot) : V_{Hq} \times V_{Hq} \to \mathbb{R},$$

set

$$A_0(\cdot,\cdot) \equiv A_{MG,0}(\cdot,\cdot). \tag{3.80}$$

Now, we introduce for each $V_k, \quad k = 0,\ldots,N_{MG}$ the bilinear form

$$A_k(\cdot,\cdot) : V_k \times V_k \to \mathbb{R}, \quad k = 0,\ldots,N_{MG}.$$

We also introduce for each $V_k, \quad k = 0,\ldots,N_{MG}$ a corresponding inner product of the form

$$(\cdot,\cdot)_k : V_k \times V_k \to \mathbb{R}, \quad k = 1,\ldots,N_{MG}. \tag{3.81}$$

We now need to introduce the tools for the **MG** method. Introduce linear operators

$$I_k : V_{k-1} \to V_k,$$

for $\quad k = 1,\ldots,N_{MG}$. The operators $\{I_k\}$ are called *prolongation* operators and defined as a classical linear interpolation operation in [14, page 4]. The operator maps functions from one level Multigrid level to another:

$$I_k v_{k-1} = v_k, \qquad \text{for all } v_k \in V_k \text{ and } v_{k-1} \in V_{k-1}.$$

Introduce the operator $A_k : V_k \to V_k$ by:

$$(A_k u, v)_k = A_k(u,v) \quad \text{ for all } v \in V_k.$$

Additionally, we define the linear operators $P_{MG,k} : V_k \to V_{k-1}$, for $\quad k = 1,\ldots,N_{MG}$ and special case $P^0_{MG,k} : V_k \to V_{k-1}$, for $\quad k = 1,\ldots,N_{MG}$ by

$$A_{k-1}(P_{MG,k-1}u, v) = A_k(u, I_k v) \quad \text{ for all } u \in V_{k-1} \tag{3.82}$$

and

$$(P^0_{MG,k-1}u, v)_{k-1} = (u, I_k v)_k \quad \text{ for all } u \in V_{k-1}.$$

Operator $I_k P_{MG,k-1}$ is a symmetric operator with respect to the A_k form [49, Chapter 4, section 2] and [14, Page 4].

We now define the smoothing process using the linear operator $R_{MG,k} : V_k \to V_k$ for $k = 1,\ldots,N_{MG}$. Symmetry of the operator is scrutinised in [14]. If $R_{MG,k}$ is nonsymmetric, then we define $R^t_{MG,k}$ to be its adjoint and we set:

$$R^{(l)}_{MG,k} = \begin{cases} R_{MG,k}, & \text{if } l \text{ is odd,} \\ R^t_{MG,k}, & \text{if } l \text{ is even.} \end{cases}$$

where l is current iteration number of pre-smoothing operation, on level k.

The multigrid operator

$$B_{MG,k} : V_k \to V_k \tag{3.83}$$

is then given as follows [49, p.137]:

Algorithm 12 Multigrid algorithm. For input arguments g and m_k, receive an output collection $B_{MG,k}$, for $k = 0, \dots, N_{MG}$

```
1: [For a given] $g$,   $m$
2: if $k = 0$ then
3:     [Initialize] $B_{MG,0} \leftarrow A_0^{-1}$
4: else
5:     $x^0 \leftarrow 0$
6:     $q^0 \leftarrow 0$
7:     for $l = 1, \dots, m_k$ do [PRESMOOTHING]
8:         $x^l \leftarrow x^{l-1} + R_{MG,k}^{l+m_k}(g - A_k x^{l-1})$
9:     end for
10:    for $i = 1, \dots, \nu$ do [COARSE GRID CORRECTION]
11:        $q^i \leftarrow q^{i-1} + B_{MG,k-1}\left[P_{MG,k-1}^0(g - A_k x^{m_k}) - A_{k-1} q^{i-1}\right]$
12:    end for
13:    $y^{m_k} \leftarrow x^{m_k} + I_k q^\nu$
14:    for $l = m_k + 1, \dots, 2m_k$ do [POSTSMOOTHING]
15:        $y^l \leftarrow y^{l-1} + R_{MG,k}^{l+m_k}(g - A_k y^{l-1})$
16:    end for
17:    $B_{MG,k} g \leftarrow y^{2m_k}$
18: end if
19: [Output] $B_{MG,0:k}$
```

In the above algorithm, each

$$m_k \in \mathbb{Z}^+ \tag{3.84}$$

is a variable for each level and determines the number of smoothing operations. $\nu \in \mathbb{Z}^+$, if $\nu = 1$ the above algorithm is a V-cycle, whereas for $\nu = 2$ the algorithm is a W-cycle. $B_{MG,k}$ is a linear and symmetric operator for each k, which additionally implies that it is symmetric with respect to the $(\cdot, \cdot)_k$ inner product[14]. Moreover, denoting Id as an identity operator. Setting

$$G_k := \mathrm{Id} - R_{MG,k} A_k,$$

yields

$$G_k^* = \mathrm{Id} - R_{MG,k}^t A_k$$

being adjoint with respect to

$$A_{MG,k}(\cdot, \cdot),$$

which in turn leads to spectrum of $G_k^* G_k$ being defined in the interval $[0, 1)$, which is shown in [14, A.1]. Additionally, from the same source,

$$\mathrm{Id} - B_{MG,k} A_k$$

is a symmetric operator on V_k with respect to the $A_{MG,k}$ form.

Additionally we note, that the cases

$$A_k(I_k u, I_k v) = \mathscr{A}_{k-1}(u,v) \quad \text{for all } u, v \in V_{k-1},$$

are only allowed when the forms A_k are inherited from the finest grid.

Introducing the error propagation operator as in [3], applicable for Multigrid method for hp-Discontinuous Galerkin methods,

$$\mathbb{E}_{k,m_1,m_2} = G_k^{m_2}(\mathrm{Id}_k - I_{k-1}^k P_k^{k-1}) G_k^{m_1}. \tag{3.85}$$

The Multigrid preconditioner, being a complex structure, can now be derived from the algorithm 12. Considering the multigrid process $\mathrm{MG}_W(k, g, z_0, m_1, m_2)$ on a system, arising from the preconditioned **CG** algorithm, shown in 11, by denoting with $A_{h,k}$ the matrix corresponding to the bilinear form $\mathscr{A}_k$, and $g = r$ for residual vector r from the algorithm 11. The error propagation operator, for the first and last cycles of the multigrid process is

$$\mathbb{E}_{k,m_1,m_2}(z - z_0) := z - \mathrm{MG}_V(k, g, z_0, m_1, m_2).$$

According to the method $z_0 = 0$, and $z = A_{h,k}^{-1} g$, yielding

$$\mathbb{E}_{k,m_1,m_2} z = z - B_{MG} g,$$

and taking into account $A_{h,k} z = g$, it follows

$$\mathbb{E}_{k,m_1,m_2} z = z - B_{MG} A_{h,k} z. \tag{3.86}$$

Here, we can give the bound for the error propagation operator for the Multigrid method as in [3]. We note, that $\|\cdot\|_{1,k}$ is a DG norm on a level k of MG algorithm. We start by defining necessary constant, from [3, Theorem 4.5], we have the following

Theorem 3.3.1. *There exists a positive constant C_{2lvl} independent of the mesh size, the polynomial approximation degree, and the level k such that*

$$\|\mathbb{E}_{k,m_1,m_2}^{2lvl} v\|_{1,k} \le C_{2lvl} \delta_k \|v\|_{1,k}$$

for any $v \in V_k, \quad 0, \dots, N_{MG}$, with

$$\delta_k := \frac{p_{MG,k}^{2+\mu}}{(1+m_1)^{1/2}(1+m_2)^{1/2}}, \tag{3.87}$$

$m_1, m_2 \ge 1$, and $\mu = 0, 1$ for optimal and suboptimal estimates, respectively. Therefore, the two-level method converges uniformly provided the number pre- and postsmoothing steps satisfy

$$(1+m_1)^{1/2}(1+m_2)^{1/2} \ge \iota p_{MG,k}^{2+\mu}$$

for a positive constant $\iota > C_{2lvl}$.

Additionally, the **DG**-norms $\|\cdot\|_{1,k}, \quad k = 1, \dots, N_{MG}$ trivially exist on $V_k, \quad k = 1, \dots, N_{MG}$

and arise from the forms (3.81). For the proof we refer to [3, page 608].

Lemma 3.3.2. *From [3, Lemma 4.6, page 608]. There exists a positive constant C_{stab} independent of the mesh size, the polynomial approximation degree, and the level k such that*

$$\begin{aligned} \|I_{k-1}^k v\|_{1,k} &\leq C_{stab}\|v\|_{1,k-1}, \quad \forall v \in V_{k-1} \\ \|P_{k-1}^k v\|_{1,k-1} &\leq C_{stab}\|v\|_{1,k}, \quad \forall v \in V_k. \end{aligned}$$

Now, we present

Theorem 3.3.3. *From [3, Theorem 4.7, page 609].**Error propagation operator estimate.** Let δ_k and C_{2lvl} be defined as in Theorem 3.3.1, and let C_{stab} be defined as in Lemma 3.3.2. Then, there exists a positive constant $\hat{C} > C_{2lvl}$ such that, if the number of pre- and postsmoothing steps satisfies*

$$(1+m_1)^{1/2}(1+m_2)^{1/2} \geq \iota p_{MG,k}^{2+\mu} \frac{C_{stab}^2 \hat{C}^2}{\hat{C} - C_{2lvl}}, \tag{3.88}$$

it holds that

$$\|\mathbb{E}_{k,m_1,m_2} v\|_{1,k} \leq \hat{C}\delta_k \|v\|_{1,k} \quad v \in V_k$$

with

$$\hat{C}\delta_k < 1.$$

That is, the W-cycle algorithm converges uniformly with respect to the discretization parameters and the number of levels provided that m_1 and m_2 satisfy (3.88).

We can show that the **MG** operator B_{MG} can be bounded.

Lemma 3.3.4. *For predefined operators B_{MG} and A, there exist a constant $0 < \delta = \hat{C}\delta_k < 1$ as in (3.87), such that*

$$(1-\delta)(v, A^{-1}v) \leq (B_{MG}v, v) \leq (1+\delta)(v, A^{-1}v). \tag{3.89}$$

Proof. From Theorem 3.3.3 and (3.86) we have

$$\begin{aligned} \|\mathbb{E}_{k,m_1,m_2} z\|_1 &= \|(\mathrm{Id} - B_{MG}A_{h,k})z\|_1 \leq \sqrt{\delta}\|z\|_1 \\ &\Rightarrow \|(\mathrm{Id} - B_{MG}A_{h,k})\|_1 \leq \sqrt{\delta} \\ &\quad \text{by the norm-inner product equivalence} \\ &\Leftrightarrow (A(\mathrm{Id} - B_{MG}A_{h,k})v, v) \leq \delta(Av, v) \\ &\quad \text{and rearranging the terms within inner products} \\ &\Rightarrow (1-\delta)(v, A^{-1}v) \leq (B_{MG}v, v) \leq (1+\delta)(v, A^{-1}v) \end{aligned}$$

□

Lemma 3.3.5. *For predefined operators B_{MG} and A, there exist constants $0 < \delta_1 < \delta_2$, such that*

$$\delta_2^{-1}(Av, v) \leq (B_{MG}^{-1}v, v) \leq \delta_1^{-1}(Av, v). \tag{3.90}$$

Proof. From Lemma 3.3.4, setting $\delta_1 = 1-\delta$ and $\delta_2 = 1+\delta$, we have

$$\delta_1(A^{-1}u,u) \le (B_{MG}u,u) \le \delta_2(A^{-1}u,u)$$

From

$$\begin{aligned} \frac{(AB_{MG}Au,u)}{(Au,u)} & \qquad \text{setting } v = Au \\ &= \frac{(AB_{MG}v,A^{-1}v)}{(v,A^{-1}v)} \\ &= \frac{(B_{MG}v,v)}{(A^{-1}v,v)} \\ &= \frac{(A^{-1}B_{MG}Av,v)}{(A^{-1}v,v)} \end{aligned}$$

it follows that

$$\begin{aligned} \delta_1 \le \quad \lambda_{min}(B_{MG}A) = \quad \inf\frac{(AB_{MG}Au,u)}{(Au,u)}, \qquad \delta_1 \le \quad \lambda_{min}(AB_{MG}) = \quad \inf\frac{(A^{-1}AB_{MG}v,v)}{(A^{-1}v,v)} \\ \delta_2 \ge \quad \lambda_{max}(B_{MG}A) = \quad \sup\frac{(AB_{MG}Au,u)}{(Au,u)}, \qquad \delta_2 \ge \quad \lambda_{max}(AB_{MG}) = \quad \sup\frac{(A^{-1}AB_{MG}v,v)}{(A^{-1}v,v)} \end{aligned}$$

we then have

$$\begin{aligned} \frac{(B_{MG}^{-1}v,v)}{(Av,v)} & \qquad \text{setting } v = A^{-1}u \\ &= \frac{(B_{MG}^{-1}A^{-1}u,A^{-1}u)}{(u,A^{-1}u)} \\ &= \frac{(A^{-1}(AB_{MG})^{-1}u,u)}{(A^{-1}u,u)} \end{aligned}$$

and

$$\begin{aligned} \inf\frac{(A^{-1}(AB_{MG})^{-1}u,u)}{(A^{-1}u,u)} = \quad \lambda_{min}((AB_{MG})^{-1}) = \quad \lambda_{max}^{-1}(AB_{MG}) \ge \quad \delta_2^{-1} \\ \sup\frac{(A^{-1}(AB_{MG})^{-1}u,u)}{(A^{-1}u,u)} = \quad \lambda_{max}((AB_{MG})^{-1}) = \quad \lambda_{min}^{-1}(AB_{MG}) \le \quad \delta_1^{-1}. \end{aligned}$$

We obtain

$$\begin{aligned} \delta_2^{-1} \le \quad & \frac{(B_{MG}^{-1}v,v)}{(Av,v)} \le \quad \delta_1^{-1} \\ \delta_2^{-1}(Av,v) \le \quad & (B_{MG}^{-1}v,v) \le \quad \delta_1^{-1}(Av,v) \end{aligned}$$

This completes the proof. □

3.3.1 Computational complexity

We will discuss convergence of the **MG** method in the next section. Now we analyse the complexity and operation count for the presented in Algorithm 12 **MG** method.

We should outline that, although one of the main points of this work is the Multigrid method for uniform approximation polynomial degrees $p_{MG,k} = q = 1, \quad k = 0, \ldots, N_{MG}$, we still going to perform the theoretical estimations including general p_{MG}.

Introduce the number of coefficients, number of non-zero elements of matrices A_k, on each level $k = 0, \ldots, N_{MG}$ of the Multigrid process, for arbitrary chosen

$$r = (\frac{h}{H})^d < 1,$$

such that

$$\begin{aligned} N_0 &:= N_{mesh}(\frac{h}{H})^d \cdot q^d \\ N_k &:= r^{k-1} N_0. \end{aligned}$$

Adding up all algebraic operations in the algorithm 12, yields the number of computational operations:

$$\begin{aligned} K_{MG} &= 5N_{MG} + 4 \cdot m_k N_{MG} + 2 \cdot \nu N_{MG} \\ &+ \sum_{k=2}^{N_{MG}} \frac{N_k}{m_k} + 4 \cdot m_k \sum_{k=2}^{N_{MG}} \frac{N_k}{m_k} + 4 \cdot \nu \sum_{k=2}^{N_{MG}} \frac{N_k}{m_k} \\ &+ N_{MG}^2 \\ &= (5 + 4 \cdot m(k) + 2 \cdot \nu) N_{MG} + (1 + 4 \cdot m_k + 4 \cdot \nu) \sum_{k=2}^{N_{MG}} \frac{N_k}{m_k} + N_{MG}^2. \end{aligned} \tag{3.91}$$

We safely use the fact, that the matrices $A_k, k = 0, \ldots, N_{MG}$ arise from forms inherited from the bilinear form (3.80), which renders the aforementioned matrices sparse. Then term $\sum_{k=1}^{N_{MG}} \frac{N_k}{m_k}$ can be written as

$$N_0 \sum_{k=1}^{N_{MG}} \frac{r^k}{m_k},$$

and using geometric series, we find, for any N_{MG}:

$$\sum_{k=1}^{N_{MG}} \frac{N_k}{m_k} < N_0 \frac{1}{1-r}.$$

This allows to approximate the number of operations as:

$$T_{MG}(d) := (5 + 4 \cdot m_k + 2 \cdot \nu) N_{MG} + (1 + 4 \cdot m_k + 4 \cdot \nu) N_0 \frac{1}{1-r} + N_{MG}^2. \tag{3.92}$$

which in big-$\mathcal{O}$ notation yields:

$$\mathcal{O}_{MG}(N_{mesh}(\frac{h}{H})^d \cdot p_{MG}^{2d}).$$

Now, the described above modification for the Additive two-layer Schwarz method would require different number of operations. We are now analysing the Additive two-layer Schwarz method with Multigrid method as a coarse level solver. For (3.62), choosing

$p_{MG} = q = 1$, substituting the matrix inverse on coarse grid with **MG** algorithm

$$\begin{aligned} T_{ASM,MG,it} &:= N_S(N_{sub,f}^2 + 2N_{sub,f}) \\ &+ T_{MG} \end{aligned} \tag{3.93}$$

With parallel version:

$$\begin{aligned} T_{ASM,MG,it,par} &:= \frac{N_S(N_{sub,f}^2 + 2N_{sub,f})}{N_P} + (N_P - 1)(\frac{N}{N_P}) \\ &+ T_{MG} \end{aligned} \tag{3.94}$$

We are not considering any parallelisation for the **MG** method in this work.

3.4 Preconditioners' convergence analysis

3.4.1 The Abstract Additive Schwarz Method convergence analysis

We will now introduce the abstract additive Schwarz method as given in [52, Section 2.2].

We start with considering a suitable finite dimensional Hilbert space V, and symmetric, positive definite bilinear form

$$A(\cdot,\cdot) : V \times V \to \mathbb{R},$$

with associated stiffness matrix form $\mathbb{A}$. Also consider linear functional $f \in V$.

We can now state the problem, find $u \in V$, such that

$$A(u,v) = f(v), \quad v \in V. \tag{3.95}$$

Given a basis of V, we have the function $u \in V$ uniquely determined by the set of degrees of freedom. Applying the functional $f \in V$ to the basis of V we obtain the vector F, so-called "load vector", and we can write (3.95) in a matrix form

$$Au = F, \tag{3.96}$$

with the vector of unknowns, also called solution vector, u, and symmetric positive definite A.

Consider family of spaces

$$\{V_i, \quad i = 0, \ldots, N_S\},$$

and the interpolation operators

$$\begin{aligned} R_i^T &: V_i \to V, \\ R_i &: V \to V_i, \end{aligned}$$

often called prolongation\restriction operators, respectively. We also need the following decomposition of space V to hold

$$V = R_0^T V_0 + \sum_{i=1}^{N_S} R_i^T V_i. \tag{3.97}$$

Which allows for all $v \in V$ to be written, with $v_0 \in V_0$ and $v_i \in V_i, \quad i = 1, \dots, N_S$, as

$$v = R_0^T v_0 + \sum_{i=1}^{N_S} R_i^T v_i.$$

Introduce abstract bilinear forms on the subspaces

$$\bar{A}_i(\cdot,\cdot) : V_i \times V_i \to \mathbb{R}, \quad i = 1, \dots, N_S,$$

and the associated abstract local stiffness matrices

$$\bar{\mathbb{A}}_i : V_i \to V_i.$$

We also define projection-like operators

$$\tilde{P}_i : V \to V_i : \bar{A}_i(\tilde{P}_i u, v_i) = A(u, R_i^T v_i), \quad v_i \in V_i, \tag{3.98}$$

which is well defined since the local bilinear forms are coercive. Using the definition (3.98), define the Schwarz operators

$$P_i = R_i^T \tilde{P}_i : V \to R_i^T V_i \subset V, \quad i = 0, \dots, N_S.$$

Also in the case of exact local solvers

$$A(P_i u, R_i^T v_i) = A(u, R_i^T u_i), \quad v_i \in V_i.$$

According to [52, Lemma 2.1] P_i are self-adjoint with respect to the scalar product induced by the $A(\cdot,\cdot)$ and positive semi-definite, and the operator is given

$$P_i = R_i^T \bar{\mathbb{A}}_i^{-1} R_i \mathbb{A}, \quad i = 0, \dots, N_S. \tag{3.99}$$

If the local bilinear forms are given by (3.103), then P_i is a projection

$$P_i^2 = P_i.$$

We can now define the additive Schwarz preconditioner

$$P_{ad} = \sum_{i=0,\dots,N_S}^{N_S} P_i. \tag{3.100}$$

The additive Schwarz operator is symmetric and positive definite, and defined in matrix form as

$$P_{ad} = B_{ad} \mathbb{A},$$

with matrix B_{ad} given as

$$B_{ad} = \sum_{i=0}^{N_S} R_i^T \bar{\mathbb{A}}_i^{-1} R_i.$$

To check the convergence of the abstract Additive Schwarz method, we consider solution of the system, as in (3.96), for $u \in V$

$$P_{ad} u = F_{ad},$$

where $F_{ad} = B_{ad} f$, $f \in V'$, is a suitable right hand side. Immediately, we can give the condition number of P_{ad}

$$\kappa(P_{ad}) = \frac{\lambda_{max}(P_{ad})}{\lambda_{min}(P_{ad})}, \tag{3.101}$$

with

$$\lambda_{max}(P_{ad}) = \sup_{u \in V} \frac{A(P_{ad}u,u)}{A(u,u)}, \quad \lambda_{min}(P_{ad}) = \inf_{u \in V} \frac{A(P_{ad}u,u)}{A(u,u)}.$$

And now, to show the stability and convergence, we need to bound the Schwarz operator. To find and prove the bound of the Schwarz operator, we need to show the following assumptions, developed in [52, section 2.3]:

Assumption 3.4.1. *From [52, Assumption 2.2].* ***(Stable Decomposition)*** *There exists a constant C_0, such that every $u \in V$ admits a decomposition*

$$u = \sum_{i=0}^{N_S} R_i^T u_i, \quad u_i \in V_i, i = 0, \dots, N_S$$

that satisfies

$$\sum_{i=0}^{N_S} \bar{A}_i(u_i, u_i) \le C_0^2 A(u,u).$$

Assumption 3.4.2. *From [52, Assumption 2.4].* ***(Local stability)*** *There exist constants $\omega > 0$, such that*

$$A(R_i^T u_i, R_i^T u_i) \le \omega \bar{A}_i(u_i, u_i) \quad u_i \in V_i, \quad i = 0, \dots, N_S$$

Assumption 3.4.3. *From [52, Assumption 2.3].* ***(Strengthened Cauchy-Schwarz inequalities)*** *There exist constants $0 \le \epsilon_{ij} \le 1$, $i, j = 1, \dots, N_S$, such that*

$$|A(R_i^T u_i, R_j^T u_j)| \le \epsilon_{i,j} A(R_i^T u_i, R_i^T u_i)^{\frac{1}{2}} A(R_j^T u_j, R_j^T u_j)^{\frac{1}{2}}$$

for all $u_i \in V_i$, $u_j \in V_j$. Define $\rho(\mathscr{E})$ to be spectral radius of $\mathscr{E} = \{\epsilon_{ij}\}_{i,j=1,\dots,N_S}$.

Now, in order to find the bound for the condition number, according to [52], we need to introduce the following lemmas.

Lemma 3.4.4. *From [52, Lemma 2.5]. Let Assumption 3.4.1 be satisfied. Then,*

$$A(P_{ad}u, u) \ge C_0^{-2} A(u,u), u \in V \tag{3.102}$$

and consequently P_{ad} is invertible. In addition,

$$A(P_{ad}^{-1}u,u) = \min_{\substack{u_i \in V_i \\ u = \sum R_i^T u_i}} \sum_{i=0} \bar{A}_i(u_i,u_i).$$

Lemma 3.4.5. *From [52, Lemma 2.6]. Let Assumptions 3.4.2 and 3.4.3 be satisfied. Then for $i = 0,\dots,N_S$,*

$$||P_i||_A \le \omega.$$

In addition,

$$A(P_{ad}u,u) \le \omega(\rho(\mathscr{E})+1)A(u,u)$$

And combining those Lemmas 3.4.4 and 3.4.5, we get the [52, Theorem 2.7]

Theorem 3.4.6. *Let Assumptions 3.4.2,3.4.1 and 3.4.3 be satisfied. Then the condition number of the additive Schwarz operator P_{ad} defined in (3.100) satisfies*

$$\kappa(P_{ad}) \le C_0^2\omega(\rho(\mathscr{E})+1)$$

To comment on parameters we will refer to [52, pages 39-46]. We note that for the presented case with exact unscaled local solvers ω is equal to one. The spectral radius $\rho(\mathscr{E})$ is used to give an upper boundary for the $\lambda_{max}(P_{ad})$. The inequalities in Assumption 3.4.3 trivially hold for $\epsilon_{ij} = 1$. However, this will set $\rho(\mathscr{E})$ to be equal to N_{DOF}, which is a very poor bound. The best possible bound is obtained for orthogonal spaces $\{R_i^T V_i\}$, in which case $\epsilon_{ij} = 0$, for $i \neq j$, and $\rho(\mathscr{E}) = 1$.

3.4.2 The Additive two-level non-overlapping Schwarz Method

For the topic of this study, one of the preconditioner methods, which we use, is the Additive two-level non-overlapping Schwarz Method, first proposed in [2] and later developed for *hp* Discontinuous Galerkin methods [4]. For the sake of completeness, we cite main findings of the work which we are employing in the current thesis.

The set up and notation of the method are as in **3.2**. We note, that for the implementation, we choose the exact local solver, using the direct-inverse on both levels.

For the levels with operators $P_i, \quad i = 1,\dots,N_S$, we first compute the system matrix A_h (2.53), which arise from the primal bilinear form $B_h(\cdot,\cdot) : V_{hp} \times V_{hp} \to \mathbb{R}$ (2.2); then we locate the partitions $\{(A_h)_i\}$ to use later according to the form (3.50). For the coarse level, we are constructing the system matrix, using the primal bilinear form on a coarse partition $\mathscr{T}_H$, for given H.

For the *exact local solvers* the following holds [52, section 2.2]

$$A_i(v_i,w_i) = A(R_i^T v_i, R_i^T w_i), v_i, w_i \in V_i, \tag{3.103}$$

implying

$$A_i = R_i^T A R_i$$

To show the convergence, spectral bounds and stability of the method, we start by citing the analysis from [4, section 4] for the symmetric *hp* **DG** methods.

We derive the following.

Lemma 3.4.7. *From [4, Assumption 3]. **(Strengthened Cauchy-Schwarz inequalities)** There exist constants* $0 \leq \epsilon_{ij} \leq 1$, $i, j = 1, \dots, N_S$, *such that*

$$|A(R_i^T u_i, R_j^T u_j)| \leq \epsilon_{i,j} A(R_i^T u_i, R_i^T u_i)^{\frac{1}{2}} A(R_j^T u_j, R_j^T u_j)^{\frac{1}{2}}$$

for all $u_i \in V_i$, $u_j \in V_j$. *Define* $\rho(\mathscr{E})$ *to be spectral radius of* $\mathscr{E} = \{\epsilon_{ij}\}_{i,j=1,\dots,n}$.

Proof. Lemma (3.4.7) holds for $\epsilon_{i,i} = 1$, for $i = 1, \dots, N_S$, which is, again, due to the construction. As for $i \neq j$, it is noted [4, page 18] that $A_h(R_i^T u_i, R_j^T u_j) \neq 0$ only if $\partial\Omega_i \cap \partial\Omega_j \neq 0$, so $\epsilon_{ij} = 1$ in those cases, and $\epsilon_{ij} = 0$ otherwise. Then, $\rho(\mathscr{E})$ can be bounded by

$$\rho(\mathscr{E}) \leq \max_i \sum_j |\epsilon_{ij}| \leq 1 + N_H,$$

where N_H is the maximum number of adjacent subdomains that a given subdomain might have. □

Lemma 3.4.8. *From [4, Assumption 2]. **(Local stability)** There exist constant* $1 \leq \omega < 2$, *such that*

$$\begin{array}{rcll} A_h(R_i^T u_i, R_i^T u_i) & \leq & \omega A_i(u_i, u_i) & u_i \in V_{hp}^i, \quad i = 1, \dots, N_S \\ A_h(R_0^T u_0, R_0^T u_0) & \leq & \omega A_0(u_0, u_0) & u_0 \in V_{Hq}. \end{array} \tag{3.104}$$

Proof. Lemma 3.4.8 is trivial for the exact local solvers due to the construction with $\omega = 1$. For the case when $1 < \omega < 2$ inequalities (3.104) hold strictly with $<$, which can be observed in the reference. □

Theorem 3.4.9. *From [5, Theorem 5.1].**(Stable Decomposition)** Every* $v \in V_{hp}$ *admits a decomposition of the form* $v = \sum_{i=0}^{N_S} R_i^T u_i$, *with* $v_0 \in V_{Hq}$ *and* $v_i \in V_{hp}^i$, $\quad i = 1, \dots, N$, *which satisfies the bound*

$$\sum_{i=0}^{N_S} A_i(v_i, v_i) \leq C_{ASM}^2 A_h(v, v),$$

with

$$C_{ASM}^2 = \gamma \frac{H}{h} \frac{p^2}{q} \tag{3.105}$$

where constant $\gamma > 1$, *is an independent of the meshsize and the approximation order, which arises from the penalty term of the primal bilinear form (2.2).*

Now we can formulate the convergence theorem for the Additive two-level non-overlapping Schwarz Method.

Theorem 3.4.10. *From [5, Theorem 5.2]. The condition number of the additive Schwarz operator* P_{ad} *defined in (3.43) satisfies*

$$\kappa(P_{ad}) \leq C_{ASM}^2 (N_H + 2). \tag{3.106}$$

For C_{ASM} given as in (3.105), and N_H is the maximum number of adjacent subdomains that a given subdomain might have.

Proof. The estimate follows from the definition of the abstract additive Schwarz operator's condition number bound in Theorem 3.4.6 for parameters of the space decomposition provided in Lemmas 3.4.8,3.4.7 and Theorem 3.4.9. □

In the next section we are going to introduce the Schwarz Method non-exact local solver on the coarse level.

3.4.3 Multigrid method as a coarse level solver

The abstract convergence theory for Schwarz method was proven for the *exact local* solvers in the works of Toselli and Widlund [52], Smith, Bjørstad and Gropp [51], and Antonietti and Houston [4]. Abstract convergence theory is based on assumed existence of stable decomposition, stability of the local bilinear forms, and boundedness of the used operators in relation to employed bilinear form [52, Theorem **2.7**].

In the defined case (3.77), we first set a coarse grid bilinear form $\hat{A}_0 : V_{Hq} \times V_{Hq} \to \mathbb{R}$ to comply with

$$\hat{A}_0(u,v) = (B_{MG}^{-1}u, v), \forall u, v \in V_{Hq}.$$

Now, we require to show that it is possible to use the **MG** method as an *approximate local* solver for the Additive Schwarz method. For this, we first formulate the assumptions which allow stable decomposition, local stability, and strengthened Cauchy-Schwarz inequality for the chosen fine grid decompositions.

We start with

Lemma 3.4.11. *For predefined operators B_{MG} and A_0, there exist constants $\omega_1, \omega_2 > 0$, such that*

$$\omega_1(A_0u,u) \le (B_{MG}^{-1}u,u) \le \omega_2(A_0u,u). \tag{3.107}$$

Proof. Proof follows from Lemmas 3.3.4 and 3.3.5. □

Lemma 3.4.12. ***(Stable decomposition)*** *There exists a minimum constant $C_0 > 0$, such that for all $u \in V_{hp}$ there exists a decomposition*

$$u = \sum_{i=0}^{N} R_i^t u_i,$$

with $u_0 \in V_{Hq}$, $\quad u_i \in V_{hp}^i$, $\quad i = 1,\dots,N$, and that satisfies

$$\left(\sum_{i=1}^{N} A_i(u_i,u_i)\right) + \hat{A}_0(u_0,u_0) \le \max(1,\omega_2)C_0^2 A_h(u,u).$$

Proof. From Theorem 3.4.9, we have $\forall u \in V_{hp}$ a decomposition

$$u = \sum_{i=0} R_0^T u_i, \quad u_0 \in V_{Hq}, \quad u_i \in V_{hp}^i, \quad i = 1,\dots,N_S$$

with

$$\sum_{i=0}^{N_S} A_i(u_i,u_i) \le C_0^2 A_h(u,u).$$

Now with Lemma 3.4.11, we have

$$\hat{A}_0(u_0,u_0) = (B_{MG}^{-1}(u_0,u_0) \le \omega_2 A_0(u_0,u_0).$$

Consequently, we have

$$\begin{aligned}\sum_{i=1}^{N_S} A_i(u_i,u_i) + \hat{A}_0(u_0,u_0) &\le \sum_{i=1}^{N_S} A_i(u_i,u_i) + \omega_2 A_0(u_0,u_0)\\ &\le \max(1,\omega_2)\sum_{i=0}^{N_S} A_i(u_i,u_i)\\ &\le \max(1,\omega_2) C_0^2 A_h(u,u)\end{aligned}$$

□

Lemma 3.4.13. ***(Strengthened Cauchy-Schwarz inequalities)*** *There exist minimum constants $0 \le \varepsilon_{ij} \le 1, 1 \le i,j \le N$, that satisfy*

$$|A(R_i^t u_i, R_j^t u_j)| \le \varepsilon_{i,j} A(R_i^t u_i, R_i^t u_i)^{\frac{1}{2}} A(R_j^t u_j, R_j^t u_j)^{\frac{1}{2}}$$

for all $u_i \in V_{hp}^i$, $u_j \in V_{hp}^j$. $\rho(\mathscr{E})$ is a spectral radius of $\mathscr{E} = \{\varepsilon_{ij}\}_{i,j=1,\dots,N_S}$, such that

$$\rho(\mathscr{E}) \le 1 + N_H,$$

where N_H is the maximum number of adjacent subdomains that a given subdomain might have.

Proof. The Strengthened Cauchy-Schwarz inequalites holds for $\epsilon_{i,i} = 1$, for $i = 1,\dots,N_S$, which is, again, due to the construction. As for $i \ne j$, it is noted [4, page 18] that $A_h(R_i^T u_i, R_j^T u_j) \ne 0$ only if $\partial\Omega_i \cap \partial\Omega_j \ne 0$, so $\epsilon_{ij} = 1$ in those cases, and $\epsilon_{ij} = 0$ otherwise. Then, $\rho(\mathscr{E})$ can be bounded by

$$\rho(\mathscr{E}) \le \max_i \sum_j |\epsilon_{ij}| \le 1 + N_H,$$

as it is identical to proven 3.4.7. □

Lemma 3.4.14. ***(Local stability)*** *When the assumptions of lemma 3.4.11 hold, let $\tilde{\omega} = \max(\omega, \frac{\omega}{\omega_1})$ be such that*

$$A(R_i^T u_i, R_i^T u_i) \le \tilde{\omega} A_i(u_i,u_i) \quad u_i \in V_{hp}^i, \quad i = 1,\dots,N \tag{3.108}$$

$$A(R_0^T u_0, R_0^T u_0) \le \tilde{\omega}(B_{MG}^{-1} u_0, u_0) \quad u_0 \in V_{Hq}. \tag{3.109}$$

Proof. From Lemma 3.4.8 we have

$$A_h(R_i^T u_i, R_i^T u_i) \le \omega A_i(u_i,u_i) \quad u_i \in V_{hp}^i, \quad i = 1,\dots,N_S,$$

which proves (3.108).

Using Lemma 3.4.11 together with (3.104), for all $u_0 \in V_{Hq}$ we have

$$\begin{aligned} A_h(R_0^T u_0, R_0^T u_0) &\leq \omega A_0(u_0, u_0) \\ &\leq \frac{\omega}{\omega_1}(B_{MG}^{-1} u_0, u_0) \\ &\leq \frac{\omega}{\omega_1}\hat{A}_0(u_0, u_0), \end{aligned}$$

which proves (3.109).

This completes the proof. □

Theorem 3.4.15. *Let assumptions in Lemmas 3.4.12, 3.4.14 and 3.4.13 hold. Then the condition number of the modified additive Schwarz operator with **MG** method $P_{ASM,MG}$ as an approximate local solver on a coarse grid V_{Hq}, satisfies*

$$\kappa(P_{ASM,MG}) \leq C_{ASM,MG}^2 (N_H + 2),$$

for $C_{ASM,MG} = \tilde{\omega} \cdot \gamma \frac{H}{h} \frac{N_{pd}^2}{q}$, $\tilde{\omega} = \max(\omega_2, \omega_1)$.

Proof. The bound on condition number follows by Lemmas 3.4.12, 3.4.14 and 3.4.13 being satisfied, and taking the bound for $\rho(\mathscr{E})$ as in lemma 3.4.7. □

Remark 1. *Theorem 3.4.15 and Lemma 3.3.4 allows us to detail the constant $C_{ASM,MG} = \tilde{\omega} \cdot \gamma \frac{H}{h} \frac{N_{pd}^2}{q}$. For $\tilde{\omega}$*

$$\tilde{\omega} = \max(\omega_2, \omega_1) = \max(\frac{1}{1-\delta}, \frac{1}{1+\delta}) = \frac{1}{1-\delta} = \frac{1}{1-\hat{C}\delta_k}.$$

We can bound the number of iterations for the Additive two-level non-overlapping Schwarz Method with Multigrid method as a coarse level solver. For the modified Additive two-layer Schwarz method with Multigrid method as a coarse level solver the constant $C_{ASM,MG}$ is growing exponentially with the growth in the polynomial approximation degree, due to the simple fact that $q = 1$. This is only counterweighted with $\tilde{\omega} \leq 1$.

Theorem 3.4.16. *The number of iterations k of the Conjugate Gradients method, with the Additive two-level non-overlapping Schwarz Method with Multigrid method as a coarse level solver to achieve an accuracy of ϵ can be bounded by*

$$k = \lceil \ln(\frac{2}{\epsilon}) \frac{(C_{ASM,MG}\sqrt{(N_H + 2)} + 1)}{2} \rceil, \tag{3.110}$$

for N_H being a maximum number of adjacent partitions that any given subdomain of fine level partition might posses.

Proof. Considering the Lemma 3.1.1, by analogy with (3.66), number of iterations k can be bound with (3.33). Choosing the constant corresponding to the method, we get (3.110). □

We can now analyse the complexity and give the time estimations for the modified additive Schwarz method, with multigrid method as an *approximate local solver* on a coarse partition, as a preconditioner for the Conjugate Gradient method as an iterative solver.

3.5 Complexity. Time estimations

The method's time consumption can be defined, in terms of big-$\mathcal{O}$ notation, as:

$$\mathcal{O}_{main} := \mathcal{O}_{GSA} + \mathcal{O}_{CG},$$

where $\mathcal{O}_{GSA}$ is a time consumption per Galerkin System assembly and $\mathcal{O}_{CG}$ is a time consumption per Conjugate-Gradient method based solver. The conventional **CG** solver complexity can be defined as:

$$\mathcal{O}_{CG} := N_{it} \cdot (\mathcal{O}_{MV} + \mathcal{O}_{P} + \mathcal{O}_{LO} + \mathcal{O}_{SP}),$$

where $\mathcal{O}_{MV}$ is a matrix-vector multiplication operations, parallel or serial, $\mathcal{O}_{P}$ is a preconditioner's contribution, zero if we apply no preconditioner, $\mathcal{O}_{LO}$ is a linear operations' contribution, lastly $\mathcal{O}_{SP}$ is a scalar operations' contribution. The Galerkin system assembly can be defined as:

$$\mathcal{O}_{GSA} := \mathcal{O}_{MS} + \mathcal{O}_{MAP} + \mathcal{O}_{GM} + \mathcal{O}_{A} + \mathcal{O}_{RHS},$$

where $\mathcal{O}_{MS}$ is a mesh and geometry setup, $\mathcal{O}_{MAP}$ is a mapping between geometry information and local-elements and their neighbours' information, $\mathcal{O}_{GM}$ is a Global matrix assembly, $\mathcal{O}_{A}$ is a local matrices assembly, which depends on a primal bilinear form, and $\mathcal{O}_{RHS}$ is a right hand side calculation time. The global matrix assembly time would depend on a primal bilinear form complexity. Local matrices timing depends on how many and which type of integrals we have in a chosen primal bilinear form, as well as the choice of the basis functions.

Now, we show the complexity and time estimations for the **CG** method with **ASM** as preconditioner and **ASM** with **MG** as a coarse level solver.

Summing over the computational complexity per iteration for **CG** method with **ADLP** basis functions (3.25) with number of operations per iteration for optimised **ASM** (3.62) yields number of operations per iteration for Conjugate-Gradient method with additive

Schwarz method:

$$\begin{aligned}
T^{sel,ADLP}_{CG,ASM,it,par} &:= \frac{9N_{mesh}\cdot N_{pd}^{d}}{N_P}+\frac{(2d+1)N_{mesh}\cdot N_{pd}^{2d}}{N_p}+N_p\cdot N_{mesh}^{1/d}N_{pd}^{d}\\
&+ 3N_P+2+5(N_P-1)+(N_P-1)\cdot(N_{mesh}\cdot N_{pd}^{d})\\
&+ \frac{(N_{mesh}\cdot N_{pd}^{d})^2}{N_PN_S}+\frac{2N_{mesh}\cdot N_{pd}^{d}}{N_P}\\
&+ (\frac{N_{mesh}}{N_S}(\frac{h}{H})^d)^2+\frac{N_{mesh}(\frac{h}{H})^d\cdot N_{mesh}\cdot N_{pd}^{d}}{N_P}\\
&+ (N_P-1)(\frac{N_{mesh}\cdot N_{pd}^{d}}{N_P})+2(N_P-1)\frac{N_{mesh}(\frac{h}{H})^d}{N_P}. \qquad (3.111)
\end{aligned}$$

And for the **C1** basis functions **CG** algorithm (3.27) with **ASM**, yields

$$\begin{aligned}
T^{sel,C1}_{CG,ASM,it,par} &:= \frac{9N_{mesh}\cdot N_{pd}^{d}}{N_P}+\frac{N_{mesh}\cdot N_{pd}^{2d}+2d\cdot N_{mesh}\cdot N_{pd}^{d}}{N_p}+N_p\cdot N_{mesh}^{1/d}N_{pd}^{d-1}\\
&+ (N_P-1)\cdot(N_{mesh}\cdot N_{pd}^{d}+3)+5N_P\\
&+ \frac{(N_{mesh}\cdot N_{pd}^{d})^2}{N_PN_S}+\frac{2N_{mesh}\cdot N_{pd}^{d}}{N_P}\\
&+ (\frac{N_{mesh}}{N_S}(\frac{h}{H})^d)^2+\frac{N_{mesh}(\frac{h}{H})^d\cdot N_{mesh}\cdot N_{pd}^{d}}{N_P}\\
&+ (N_P-1)(\frac{N_{mesh}\cdot N_{pd}^{d}}{N_P})+2(N_P-1)\frac{N_{mesh}(\frac{h}{H})^d}{N_P}. \qquad (3.112)
\end{aligned}$$

Now, by summing expressions (3.25) with number of operations per modified additive Schwarz method (3.60) with multigrid method as an *approximate local* solver for coarse grid (3.92):

$$\begin{aligned}
T^{sel,ADLP}_{CG,ASM,it,par} &:= \frac{9N_{mesh}\cdot N_{pd}^{d}}{N_P}+\frac{(2d+1)N_{mesh}\cdot N_{pd}^{2d}}{N_p}+N_p\cdot N_{mesh}^{1/d}N_{pd}^{d}\\
&+ 3N_P+2+5(N_P-1)+(N_P-1)\cdot(N_{mesh}\cdot N_{pd}^{d})\\
&+ \frac{(N_{mesh}\cdot N_{pd}^{d})^2}{N_PN_S}+\frac{2N_{mesh}\cdot N_{pd}^{d}}{N_P}\\
&+ \Big[(5+4\cdot m_k+2\cdot v)N_{MG}\\
&+ (1+4\cdot m_k+4\cdot v)\frac{N_{mesh}(\frac{h}{H})^d}{1-r}+N_{MG}^2\Big]. \qquad (3.113)
\end{aligned}$$

And for the **C1** basis functions (3.27) i

$$\begin{aligned}
T^{sel,C1}_{CG,ASM,MG,it,par} &:= \frac{9N_{mesh}\cdot N^d_{pd}}{N_P} + \frac{N_{mesh}\cdot N^{2d}_{pd} + 2d\cdot N_{mesh}\cdot N^d_{pd}}{N_p} + N_p\cdot N^{1/d}_{mesh}N^{d-1}_{pd} \\
&+ (N_P - 1)\cdot(N_{mesh}\cdot N^d_{pd} + 3) + 5N_P \\
&+ \frac{(N_{mesh}\cdot N^d_{pd})^2}{N_P N_S} + \frac{2N_{mesh}\cdot N^d_{pd}}{N_P} \\
&+ [(5 + 4\cdot m_k + 2\cdot \nu)N_{MG} \\
&+ (1 + 4\cdot m_k + 4\cdot \nu)\frac{N_{mesh}(\frac{h}{H})^d}{1-r} + N^2_{MG}\Bigg]. \qquad (3.114)
\end{aligned}$$

Now, to perform the speed gain analysis, we first formulate the Amdahl's law [1], for parallel execution time ratio T_{N_p} of a total execution time T_T

$$T_{N_p} := \varpi + \frac{1-\varpi}{N_P}, \qquad (3.115)$$

with ϖ being a fraction of a serial code, and $1-\varpi$ is a fraction of a parallel code. This translates into speed up gain S_{N_p}

$$S_{N_p} := \frac{T_T}{T_N} = \frac{N_P}{N_P\varpi + 1 - \varpi}. \qquad (3.116)$$

Speed up gain is not linear, as it can be seen from the (3.116).

Now, in order to compute the serial or parallel code fraction we can to compare the operation count for parallel and for serial algorithms. For instance for the **CG** algorithm with Additive two-layer non-overlapping Schwarz method, theoretical fraction of the serial code of the **ASM** part itself, can be obtained by dividing the operation count for parallel execution (3.60) by the operation count for serial execution (3.55)

$$\varpi_{ASM,sel,it} := \frac{T^{sel}_{ASM,it,par}}{T^{sel}_{ASM,it}}.$$

Using the formulas (3.112) and (3.114) we can separate the coarse level solver and compare the operation count for both. We want to determine how much **MG** is faster than **ASM** on the coarse level for V_{Hq}. We are comparing the **MG** not only with non-parallel version of **ASM** an a coarse grid, but also the **ASM** with effect of the parralelization. We are performing comparison for V-cycle **MG** method. Both methods are analysed for the approximation polynomial degree set to be equal to 1 on coarse level. We use the **C1**-polynomials as a basis functions for all comparisons, as the optimal choice for the investigated problem.

The comparison performed by considering from (3.53,3.92) the following

$$\Theta := \frac{T^{coarse}_{ASM,it,par}}{T_{MG}}. \qquad (3.117)$$

All computations are made for **C1** basis functions as a considered optimal choice, as it will be demonstrated in the next chapter.

Table 3.2: Comparison of the Multigrid method with the direct inverse for the Additive Schwarz Method on a coarse level. $\frac{h}{H} = \frac{1}{2}$. 2D. Numbers are from (3.117).

				Number of Processes			
h^{-1}	N_{mesh}	N_{pd}	DoF	1	2	4	8
4	16	3	256	17.4	8.8	4.5	2.3
4	16	4	400	30.9	15.5	7.9	4.0
4	16	10	1936	192.8	96.5	48.3	24.3
8	64	3	1024	19.0	9.5	4.8	2.5
8	64	4	1600	33.7	16.9	8.5	4.3
8	64	10	7744	210.0	105.0	52.6	26.3
16	256	3	4096	19.4	9.7	4.9	2.5
16	256	4	6400	34.4	17.2	8.7	4.4
16	256	10	30976	214.8	107.4	53.7	26.9
32	1024	3	16384	19.5	9.8	4.9	2.5
32	1024	4	25600	34.6	17.3	8.7	4.4
32	1024	10	123904	216.0	108.0	54.1	27.1
64	4096	3	65536	19.5	9.8	4.9	2.5
64	4096	4	102400	34.7	17.4	8.7	4.4
64	4096	10	495616	216.3	108.2	54.1	27.1

Table 3.3: Comparison of the Multigrid method with the direct inverse for the Additive Schwarz Method on a coarse level. $\frac{h}{H} = \frac{1}{4}$. 2D. Numbers are from (3.117).

				Number of Processes			
h^{-1}	N_{mesh}	N_{pd}	DoF	1	2	4	8
4	16	3	256	4.3	2.2	1.1	0.6
4	16	4	400	7.7	3.9	2.0	1.0
4	16	10	1936	48.2	24.1	12.1	6.0
8	64	3	1024	4.7	2.4	1.2	0.6
8	64	4	1600	8.4	4.2	2.1	1.1
8	64	10	7744	52.5	26.2	13.1	6.6
16	256	3	4096	4.8	2.4	1.2	0.6
16	256	4	6400	8.6	4.3	2.1	1.1
16	256	10	30976	53.7	26.8	13.4	6.7
32	1024	3	16384	4.9	2.4	1.2	0.6
32	1024	4	25600	8.6	4.3	2.2	1.1
32	1024	10	123904	54.0	27.0	13.5	6.7
64	4096	3	65536	4.9	2.4	1.2	0.6
64	4096	4	102400	8.7	4.3	2.2	1.1
64	4096	10	495616	54.1	27.0	13.5	6.8

Table 3.4: Comparison of the Multigrid method with the direct inverse for the Additive Schwarz Method on a coarse level. $\frac{h}{H} = \frac{1}{2}$. 3D. Numbers are from (3.117).

				Number of Processes			
h^{-1}	N_{mesh}	N_{pd}	DoF	1	2	4	8
4	16	3	256	26.0	13.1	6.6	3.3
4	16	4	400	61.7	30.9	15.5	7.8
4	16	10	1936	963.6	481.8	241.0	120.5
8	64	3	1024	28.4	14.2	7.1	3.6
8	64	4	1600	67.2	33.6	16.8	8.4
8	64	10	7744	1049.6	524.8	262.4	131.2
16	256	3	4096	29.0	14.5	7.3	3.6
16	256	4	6400	68.7	34.4	17.2	8.6
16	256	10	30976	1073.5	536.8	268.4	134.2
32	1024	3	16384	29.2	14.6	7.3	3.7
32	1024	4	25600	69.1	34.6	17.3	8.7
32	1024	10	123904	1079.7	539.8	269.9	135.0
64	4096	3	65536	29.2	14.6	7.3	3.7
64	4096	4	102400	69.2	34.6	17.3	8.7
64	4096	10	495616	1081.2	540.6	270.3	135.2

Table 3.5: Comparison of the Multigrid method with the direct inverse for the Additive Schwarz Method on a coarse level. $\frac{h}{H} = \frac{1}{4}$. 3D. Operation count for parallel versions $N_p = 2, 4, 8$ includes data transfer operation count. Numbers are from (3.117).

				Number of Processes			
h^{-1}	N_{mesh}	N_{pd}	DoF	1	2	4	8
4	16	3	256	3.3	1.6	0.8	0.4
4	16	4	400	7.7	3.9	1.9	1.0
4	16	10	1936	120.4	60.2	30.1	15.1
8	64	3	1024	3.5	1.8	0.9	0.4
8	64	4	1600	8.4	4.2	2.1	1.1
8	64	10	7744	131.2	65.6	32.8	16.4
16	256	3	4096	3.6	1.8	0.9	0.5
16	256	4	6400	8.6	4.3	2.1	1.1
16	256	10	30976	134.2	67.1	33.5	16.8
32	1024	3	16384	3.6	1.8	0.9	0.5
32	1024	4	25600	8.6	4.3	2.2	1.1
32	1024	10	123904	135.0	67.5	33.7	16.9
64	4096	3	65536	3.6	1.8	0.9	0.5
64	4096	4	102400	8.6	4.3	2.2	1.1
64	4096	10	495616	135.2	67.6	33.8	16.9
64	4096	4	512000	62.31	31.15	15.58	7.79
64	4096	10	5451776	15211.84	7605.92	3802.96	1901.48

Results in the tables (3.2,3.3,3.4,3.5) show, that direct inverse on the coarse level for Additive Schwarz Method is being slower than the non-parallel Multigrid method. Direct inverse on the coarse level of Additive Schwarz Method with full effect of the parallel computation still might outperform the non-parallel Multigrid version when applied to a problem with lower polynomial degree or when executed massively parallel. Although massive parallelization of the aforementioned **ASM** hypothetically would allow to solve the problem with higher polynomial degree basis faster than non-parallel **MG** method, we should also consider that the data transfer for the **MPI** implementation would surge significantly and might negate the parallel effect. As for the **OpenMP** massive parallelization does not require data transfer, and the number of parallel threads in **OpenMP** is currently only limited by the integer data type in the software implementation. On the other hand, at the moment largest massive multi-core and multi-chip devices are limited to hundreds, hence for the problem with $N_{pd} = 10$ shown in (3.3), even for the smallest mesh size, it would require at least $N_P = 1024$ to outperform the presented Multigrid method, which is currently impossible to reach using only one of the analysed platforms. We can observe similar trends for all data presented.

Chapter 4

Numerical experiments

In this chapter we are presenting numerical results in detail, provide the validation of the estimates using problems with known, as well as unknown, smooth solution, present approximate error estimates, convergence rate for different penalty terms, mesh sizes and polynomial degrees. First we need to give specification of the hardware to set some common ground in case of future experiments, or improvements for different types of (non-) parallel executions.

We are going to demonstrate results for the the model problem introduced in (2.1) for different spatial dimensions (2,3), right hand sides, initial parameters, boundary conditions. We are going to compare the methods with Anti-Derivatives of Legendre Polynomial (**ADLP**) functions (2.20) and C1 (b-spline) functions (2.21), as basis functions.

4.1 Model problems in 2D

We choose the following model problems to solve using Symmetric Interior Penalty and Local Discontinuous methods. In order to compare the results of the **CG** method with **ASM** as a preconditioner with **MG** as the coarse level solver, with the ones shown in [2, Section 7], we introduce the following model problem of the Laplacian on the square $\Omega = [0,1]^2$ in two spatial dimensions:

$$\begin{aligned} -\Delta u &= f \text{ in } \Omega \\ u &= u_0 \text{ on } \Gamma = \partial\Omega. \end{aligned}$$

Problem C. Exact solution given as

$$u(x,y) = e^{xy},$$

with

$$u(0,y) = u(x,0) = e^0 = 1,$$

and right hand side given as:

$$f(x,y) = e^{xy}(x^2+y^2).$$

Here, the penalty term is chosen $\alpha := \gamma h^{-1}$, for $\gamma = 10$.

For all of the experiments we are using meshes with rectangular elements (squares) with uniform element radius h, and basis functions with polynomial degree p.

4.2 Model problem in 3D

As it was observed before, polynomial explosion in Discontinuous Galerkin method can heavily influence the time required by the solver. In order to show how presented solvers would behave for 3D problem, we choose simple model problem.

Consider, homogenous Dirichlet problem of the Laplacian on the cube $\Omega = [-1,1]^3$ in three spatial dimensions

$$\begin{aligned} -\Delta u &= f \text{ in } \Omega \\ u &= 0 \text{ on } \Gamma = \partial\Omega. \end{aligned}$$

Problem A3. Right hand side

$$f(x,y,z) = 3\pi^2 \sin(\pi x)\sin(\pi y)\sin(\pi z).$$

As the penalty term α we choose,

$$\alpha := \gamma p^2 h^{-1},$$

with sub-problems for $\gamma = 10$ and $\gamma = 100$. The exact solution of the model problem is known to be:

$$u(x,y,z) = \sin(\pi x)\sin(\pi y)\sin(\pi z).$$

4.3 Parallel solvers.

We are using four solvers in order to solve the matrix form of the model problems. We designate them as follows:

Solver SA. Conventional Conjugate Gradients method described in chapter **3**, section 1.

Solver SB. Conjugate Gradients method with Additive Schwarz Method as a preconditioner, described in chapter **3**, section 2.

Solver SC. Conjugate Gradient method with Additive Schwarz Method as a preconditioner and Multigrid method as a coarse level solver for **ASM** described in chapter **3**, section 4.

4.4 Used hardware and software

All executions were run on two clusters **Milet** with four hardware machines and **Heron** with twelve hardware machines. For experiments' purity and consistency, all *OpenMP* executions were run on **Milet** machines, whereas *MPI* executions were run on **Heron** cluster, without *OpenMP* parallelism. The hardware specifications shown in the table below.

Table 4.1: Cluster machines' specification. Each machine within cluster has identical specification.

Description	Parameter	
Architecture:	x86_64	
CPU op-modes:	32-bit, 64-bit	
CPUs:	24	40
On-line CPUs:	0-23	0-39
Threads per core:	2	
Cores per socket:	6	10
Sockets:	2	
NUMA nodes:	2	
CPU family:	6	
Model:	45	79
Vendor name:	Intel(R) Xeon(R) CPU	
Model name:	E5-2640 @ 2.50GHz	E5-2640 @ 2.40GHz
CPU revision:	0	v4
Stepping:	7	
CPU MHz:	2000	1317.681
BogoMIPS:	5004.97	4800.3
Virtualization:	VT-x	
L1d cache:	32K	
L1i cache:	32K	
L2 cache:	256K	
L3 cache:	15360K	25600K
NUMA node0 CPUs:	0-5,12-17	0-38:even
NUMA node1 CPUs:	6-11,18-23	1-39:odd
Nodes in cluster:	4	12
	Milet	Heron

GNU Fortran (GCC) version 6.2.0 was used to compile the source code. OpenRTE version 2.0.1 and Open MPI version 3.1 were used for parallel program execution with **MPI**. For **OMP** parallelism Open MP libraries version 4.0 were used for the compilation of the program.

4.5 Validation of the results

Another error estimation arise from the **CG** method itself. Vector of residuals r_k, for iteration k, is being derived from Lanczos algorithm [46, Proposition 6.20], as a direct difference between the right hand side vector b and a product of a Galerkin matrix A and the k-th approximation of the solution c_k^u (*i.e.* $r_k = b - Ac_k^u$). Vector of residuals is conventionally used as a convergence criterion in most implementations of the **CG** algorithm, typically [2, Section 7],[50, Section 11.2], for some $\epsilon = 10^{-12}$, and Euclidean norm $||\cdot||_2$, the following is used:

$$||r_k||_2 \leq \epsilon ||r_0||_2.$$

This allows to use the value of $||r_k||_2$ at the final k-th iteration as an indicator of convergence. In this work we are using the tolerance regulator

$$\epsilon := 10^{-12},$$

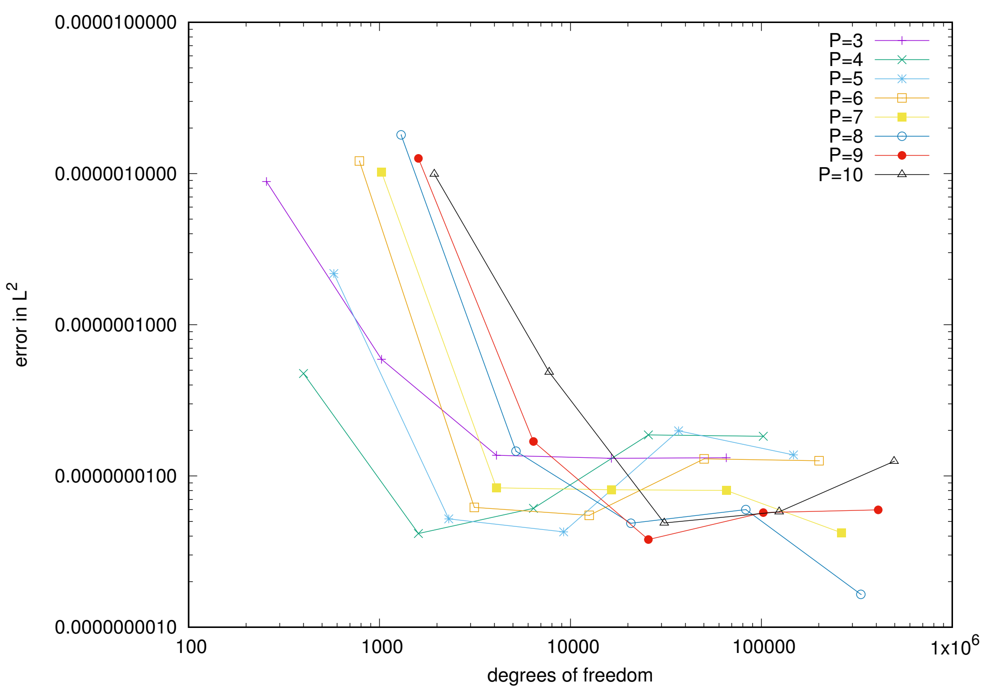
and stopping criterion for the solver **SA**,

$$||r_k||_2 \leq \epsilon \frac{||b - Ac_k^u||_2}{||b||_2}.$$

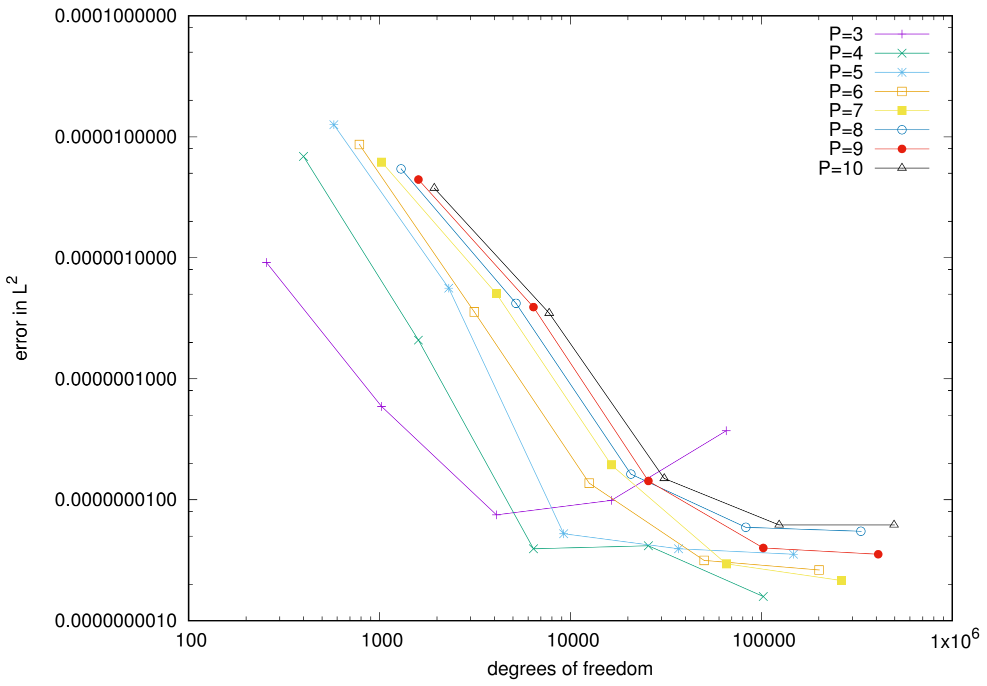
Convergence criterion for the solvers **SB,SC** with preconditioner $\mathbb{B}$ is chosen as

$$||z_k||_2 \leq \epsilon \frac{||\mathbb{B}b - \mathbb{B}Ac_k^u||_2}{||\mathbb{B}b||_2}.$$

We should note, that the error estimators shown, unless labelled otherwise, both in L^2 and H_0^1 norms, obtained by the **CG** solver **SA**. We are allowed to use those as the error estimators for all three solvers, providing the solvers are used for the same problem with the same parameters, boundary and initial conditions. This is due to the fact that preconditioned **CG** solvers, if implemented correctly, demonstrate values which lie in the ϵ neighbourhood of the conventional **CG**. The difference in absolute values amongst the error estimators of the solvers arise from rounding and cancellations, and for parallel implementation - precision loss during distributed computations.



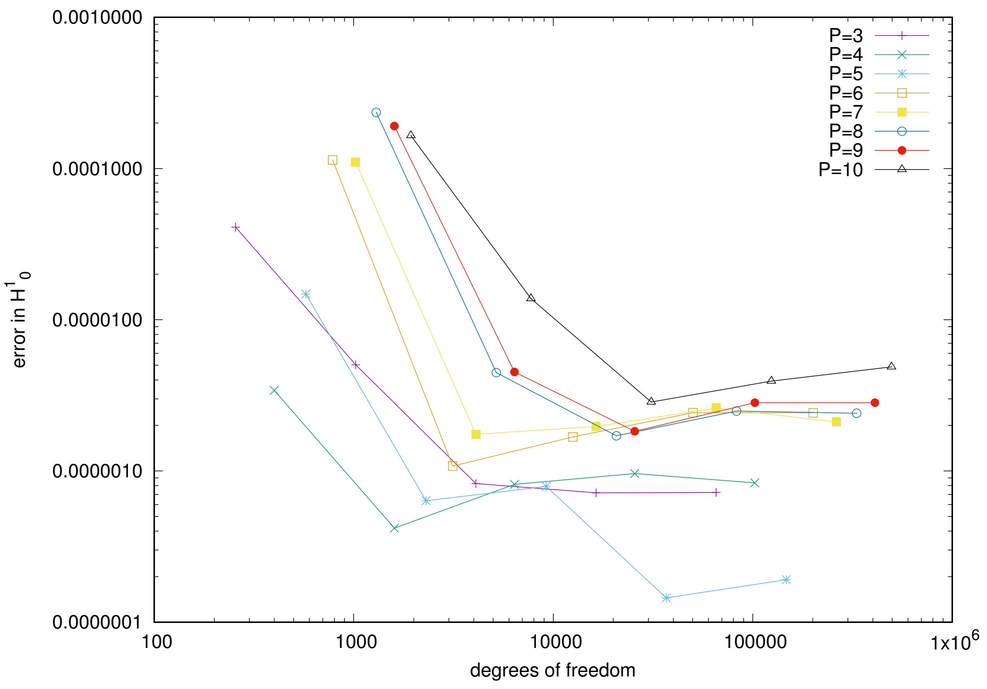
ADLP basis functions.



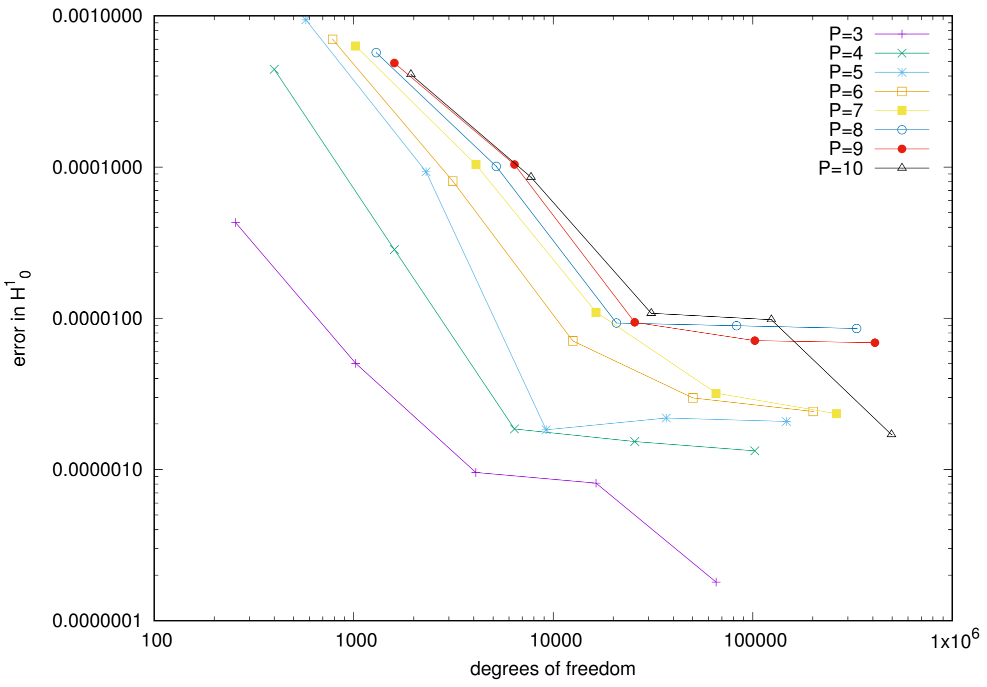
C1 basis functions.

Figure 4.1: Error in L^2, for problem **C**. Logarithmic scale on all axes.

In the figures 4.1 we can observe the convergence as both $1/h$ and p grow. Additionally one can observe that for the surge in computable elements the error indicator rises, this can be attributed to the numerical noise due to rounding and cancellations. The same trend is observed on the figures 4.2.



ADLP basis functions.



C1 basis functions.

Figure 4.2: Error in H_0^1, for problem **C**. Logarithmic scale on all axes.

4.5.1 Condition number estimates

Condition number of the global system matrix is the best measure for convergence analysis. It can be used to estimate the total number of iterations required by the iterative solvers in theorem 3.1.1. Condition number is a relation (3.29) of the smallest λ_{min} and largest

λ_{max} eigenvalues of the system matrix $\mathbb{A}$:

$$\kappa(\mathbb{A}) := \frac{\lambda_{max}}{\lambda_{min}}. \tag{4.1}$$

In this subsection we will compute the condition number. We first present required tables, and then derive the figures for simplicity of the data observations. In the following tables h and H are the element radii of the fine and coarse grids. N_{pd} is the polynomial approximation degree. $\Delta\kappa$ - exponential change in condition numbers.

Table 4.2: hp-Condition numbers, for problem **C**, solver **SB**. ADLP basis functions. IPG.

	h/H									
N_{pd}	4/4	4/2	8/8	8/4	16/16	16/8	32/32	32/16	64/64	64/32
3	205.0703	601.3363	207.6635	608.9406	202.5874	594.0557	201.7494	591.5982	201.7483	591.595
4	644.451	1889.751	627.2358	1839.27	615.5926	1805.128	615.4207	1804.624	615.4461	1804.699
5	1539.28	4513.697	1495.071	4384.059	1466.962	4301.636	1467.102	4302.045	1458.299	4276.231
6	3150.925	9239.59	3060.737	8975.128	2987.407	8760.099	2987.385	8760.036	2969.044	8706.254
7	5801.572	17012.2	5576.844	16353.21	5450.114	15981.6	5453.15	15990.5	5452.875	15989.7
8	9843.642	28864.93	9535.086	27960.14	9258.768	27149.88	9266.062	27171.27	9232.235	27072.07
9	15720	46096.42	15030	44073.11	14770	43310.7	14710	43134.76	14660	42988.14
10	23920	70141.63	22980	67385.23	22510	66007.03	22350	65537.85	22340	65508.53
$\Delta\kappa$	2.033523	2.501233	2.012614	2.91829	2.014369	3.283421	2.014581	3.102454	2.013927	2.660397

Table 4.3: hp-Condition numbers, for problem **C**, solver **SC**. **ADLP** basis functions. IPG.

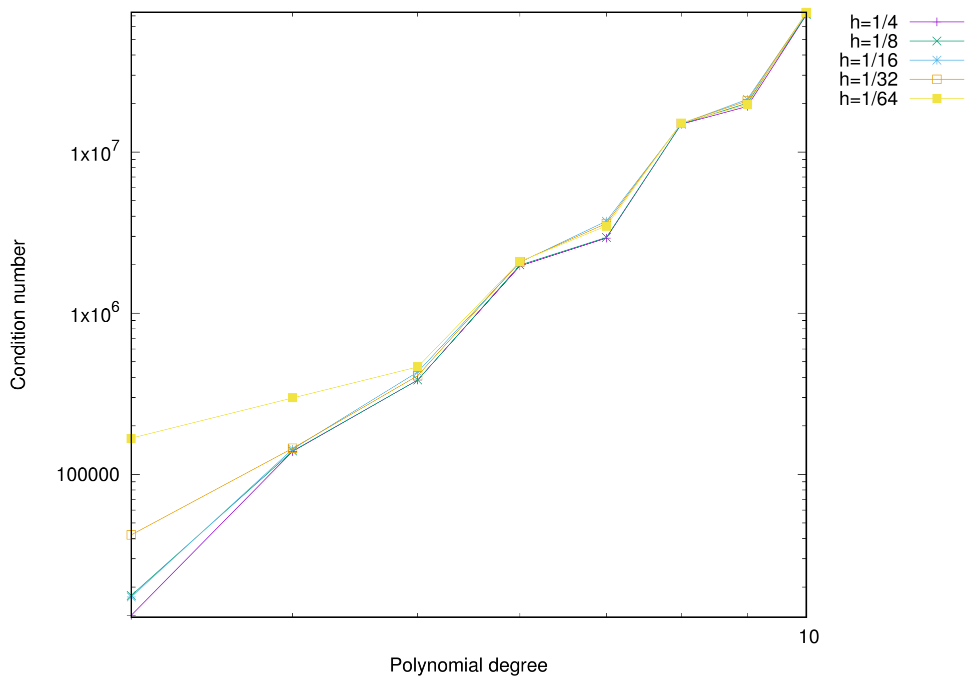
	h/H									
N_{pd}	4/4	4/2	8/8	8/4	16/16	16/8	32/32	32/16	64/64	64/32
3	206.9302	606.7903	211.4273	619.9771	210.4307	617.0548	206.3902	605.2066	201.6311	591.2515
4	615.5809	1805.094	623.7803	1829.137	627.614	1840.379	1031.398	3024.411	1202.806	3527.04
5	1439.853	4222.141	1368.382	4012.564	1447.067	4243.296	1561.546	4578.989	1817.572	5329.743
6	2915.382	8548.898	2738.401	8029.93	2869.712	8414.977	2918.728	8558.708	2745.355	8050.319
7	5368.311	15741.73	5006.741	14681.48	5136.175	15061.02	5236.943	15356.51	5173.325	15169.96
8	9099.765	26683.63	8441.179	24752.43	8610.033	25247.56	8700.354	25512.42	8668.88	25420.12
9	14500	42518.96	13500	39586.62	13700	40173.09	13800	40466.32	13740	40290.38
10	22020	64570.18	20500	60113.02	20770	60904.75	20910	61315.28	20830	61080.69
$\Delta\kappa$	1.998164	2.457742	1.968193	2.853879	1.97781	3.223831	2.133916	3.286231	2.235431	2.953004

Table 4.4: hp-Condition numbers, for problem **C**, solver **SB**. **C1** basis functions. IPG.

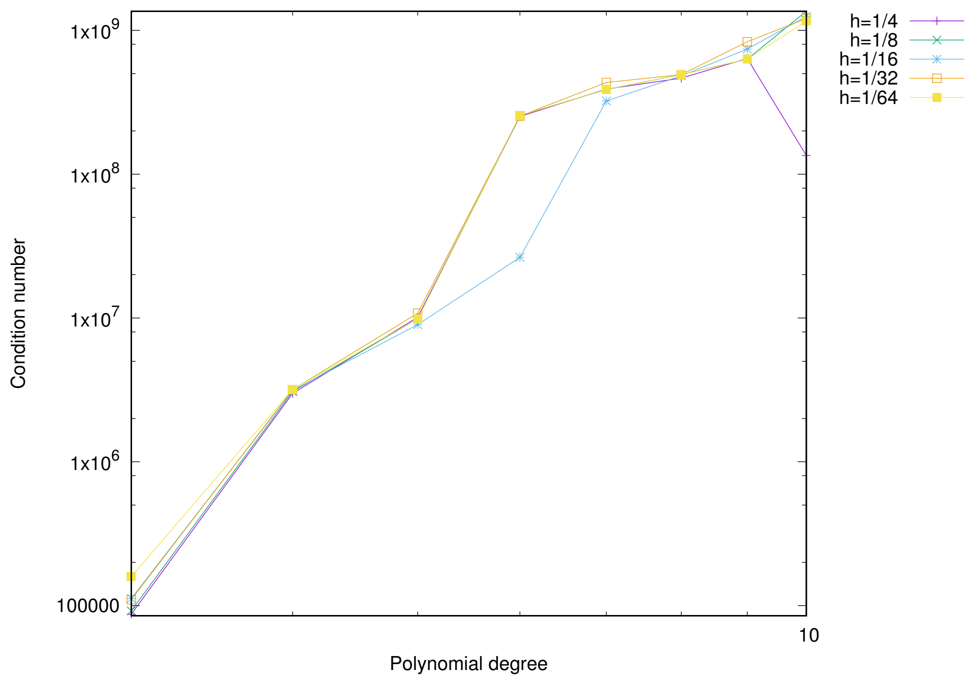
	h/H									
N_{pd}	4/4	4/2	8/8	8/4	16/16	16/8	32/32	32/16	64/64	64/32
3	207.778	588.4984	202.8065	574.4175	204.5225	579.2777	203.6565	576.825	203.51	576.4101
4	640.7941	1814.948	618.4604	1751.692	624.3017	1768.236	623.0997	1764.832	622.6609	1763.589
5	1533.116	4342.31	1475.976	4180.47	1490.435	4221.421	1487.038	4211.799	1485.402	4207.166
6	3114.823	8822.244	3015.881	8542.006	3063.251	8676.174	3042.755	8618.123	3038.327	8605.583
7	5736.502	16247.74	5541.658	15695.87	5639.416	15972.76	5587.241	15824.98	5603.816	15871.93
8	9725.399	27545.66	9415.213	26667.11	9588.498	27157.91	9496.831	26898.28	9476.506	26840.71
9	15580	44127.89	15060	42655.08	15340	43448.13	15150	42909.99	15120	42825.02
10	23720	67183.16	22840	64690.7	23280	65936.93	23100	65427.11	22910	64888.96
$\Delta\kappa$	2.024248	2.469583	2.018854	2.967715	2.021973	3.376695	2.021181	3.153042	2.019547	2.847561

Table 4.5: hp-Condition numbers, for problem **C**, solver **SC**. **C1** basis functions. IPG.

	h/H									
N_{pd}	4/4	4/2	8/8	8/4	16/16	16/8	32/32	32/16	64/64	64/32
3	207.214	586.901	210.9337	597.4365	215.6292	610.7356	253.0361	716.6848	261.8289	741.5891
4	615.6056	1743.606	629.7971	1783.801	624.5743	1769.008	1195.287	3385.461	1365.637	3867.95
5	1441.015	4081.447	1453.751	4117.522	1433.384	4059.834	1807.576	5119.674	2062.666	5842.176
6	2929.568	8297.538	2901.207	8217.21	2853.06	8080.843	2921.589	8274.94	2916.753	8261.244
7	5375.771	15226.02	5254.794	14883.38	5161.558	14619.3	5259.991	14898.09	5198.585	14724.17
8	9114.306	25814.83	8902.985	25216.3	8667.906	24550.48	8826.287	24999.07	8799.64	24923.59
9	14540	41182.26	14130	40021	13780	39029.68	13900	39369.56	13960	39539.5
10	22090	62566.44	21380	60555.48	20850	59054.34	21090	59734.1	21000	59479.19
$\Delta\kappa$	1.998524	2.438199	1.98506	2.918038	1.967603	3.285897	2.060405	3.214231	2.100854	2.962205



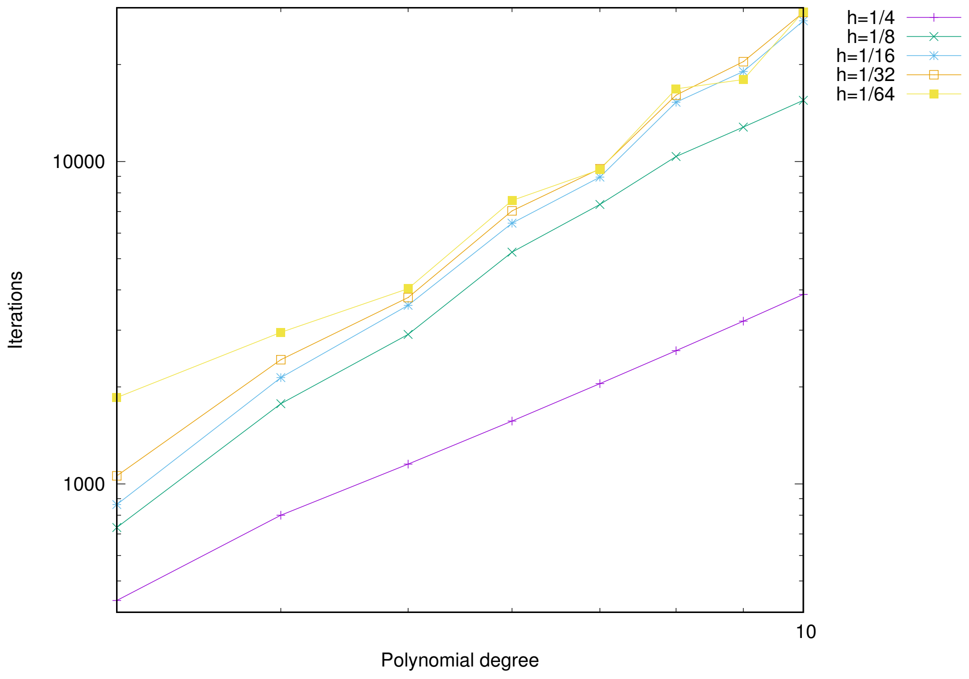
ADLP basis functions.



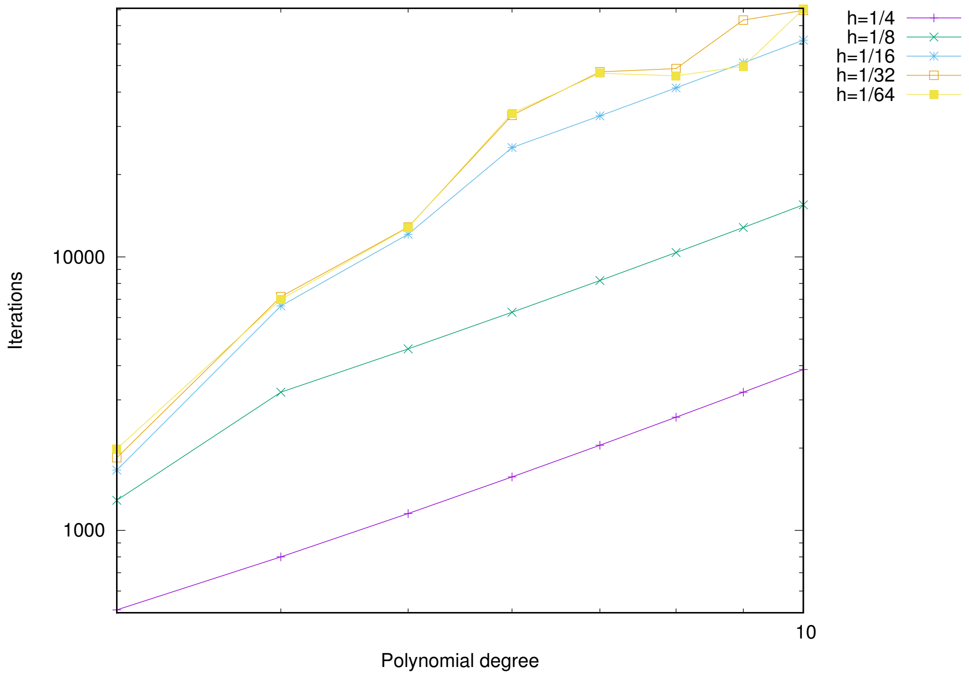
C1 basis functions.

Figure 4.3: Condition number estimates, for problem **C**, with CG solver **SA**. Logarithmic scale on all axes.

In the case of conventional CG solver **SA**, on figures 4.3 we observe that condition number with **C1** basis functions are slightly higher than with **ADLP**. It is caused mostly by the fact that with **C1** we have much sparser system matrix and need more iterations to solve the problem. Iteration numbers are shown on the following figures.

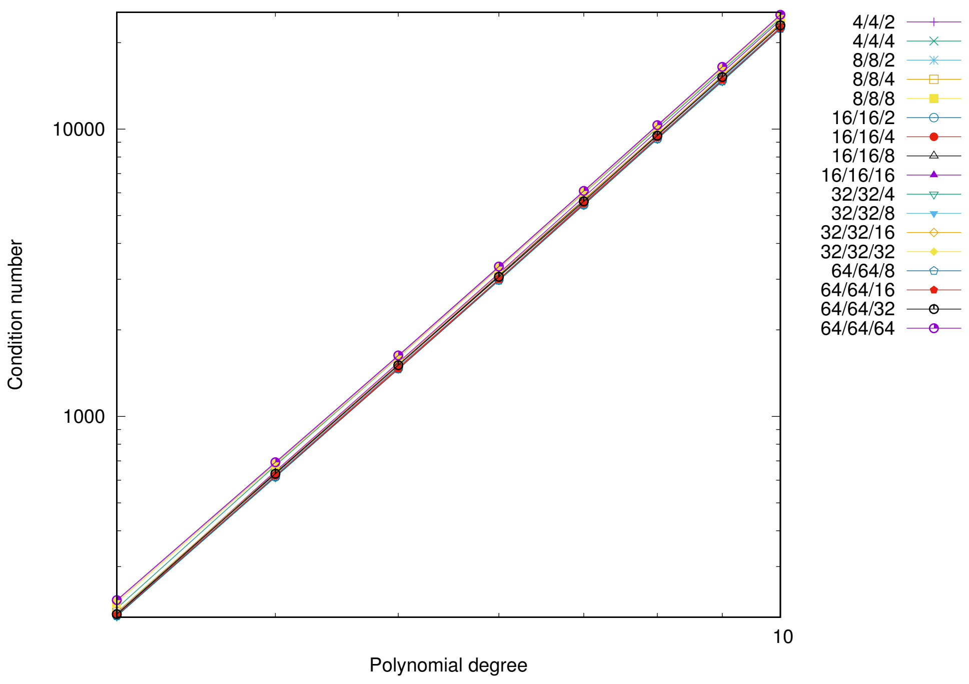


ADLP basis functions.

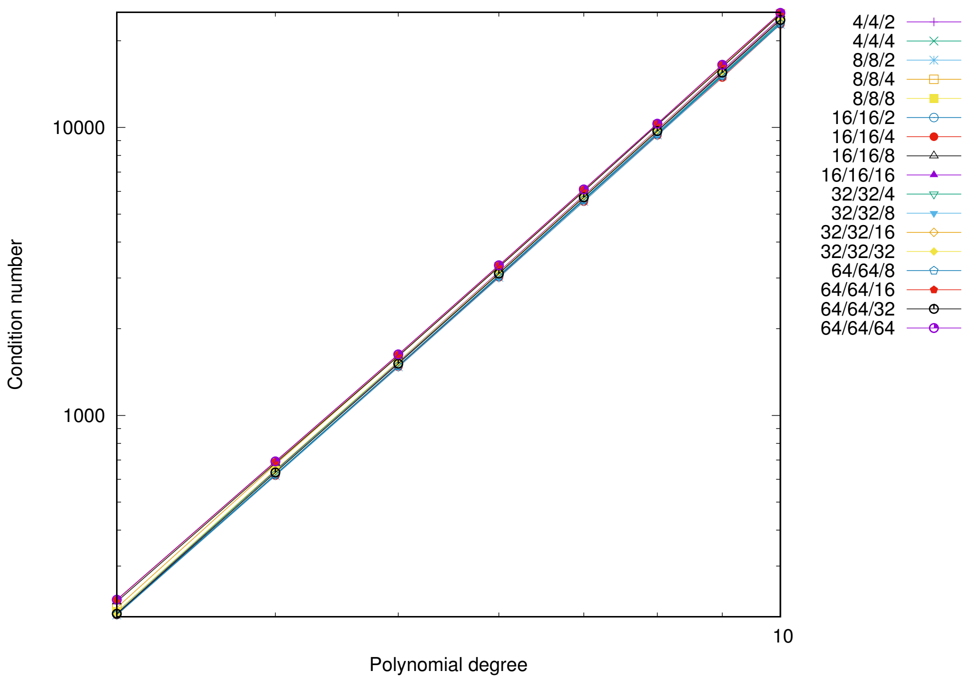


C1 basis functions.

Figure 4.4: Iteration number, for problem **C**, with CG solver **SA**. Logarithmic scale on all axes.



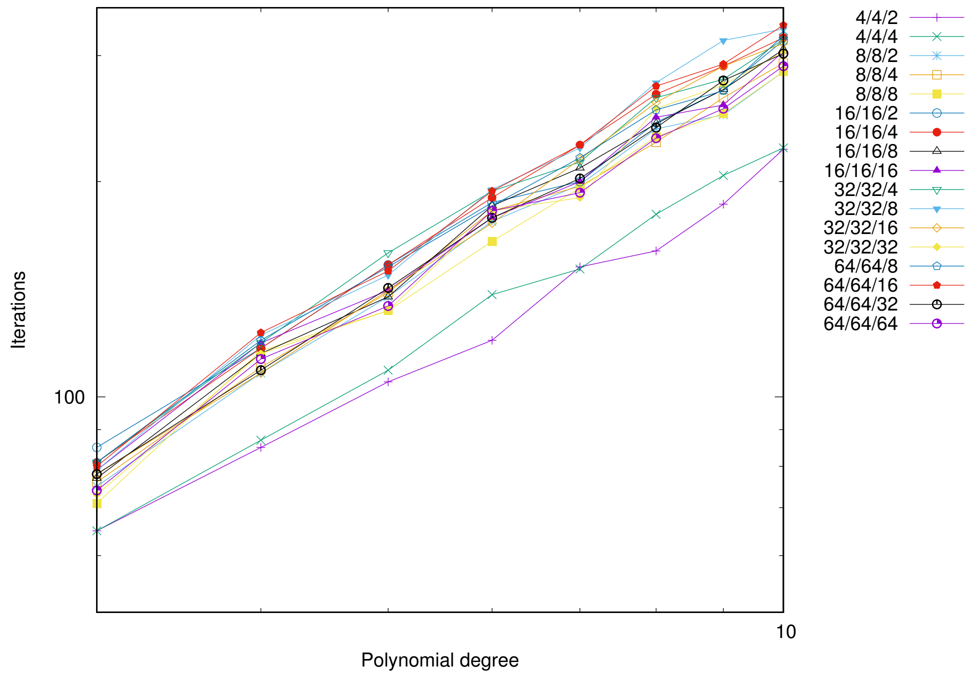
ADLP basis functions.



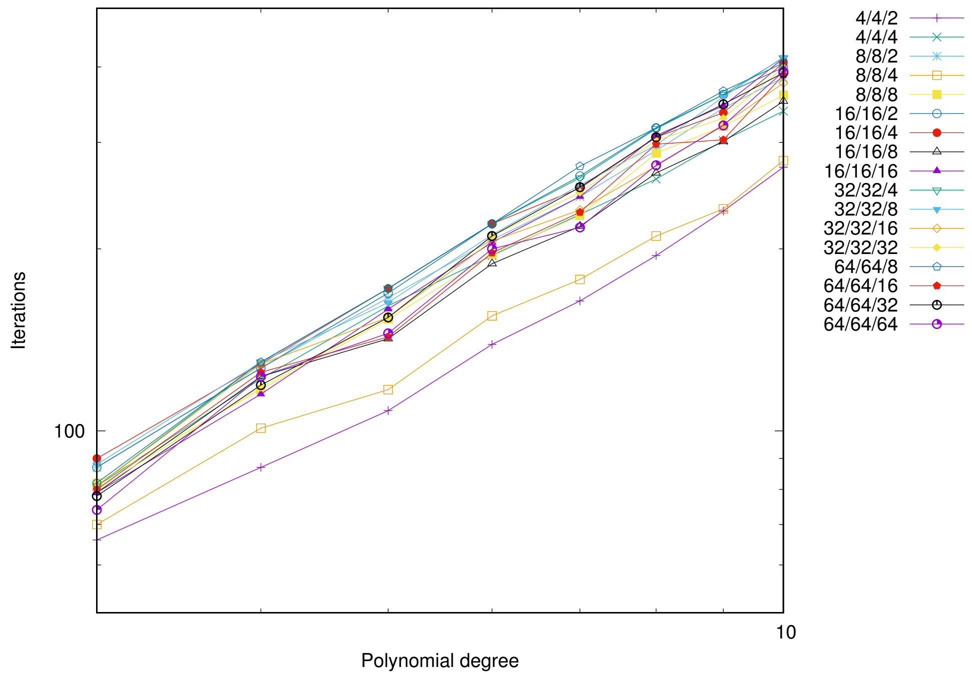
C1 basis functions.

Figure 4.5: Condition number estimates, for problem **C**, with solver **SB**. Logarithmic scale on all axes.

Here, on the figures 4.5 we observe that the condition numbers of the preconditioned system are very similar to each other. Ideally the condition numbers for the preconditioned system matrix should be exactly the same.

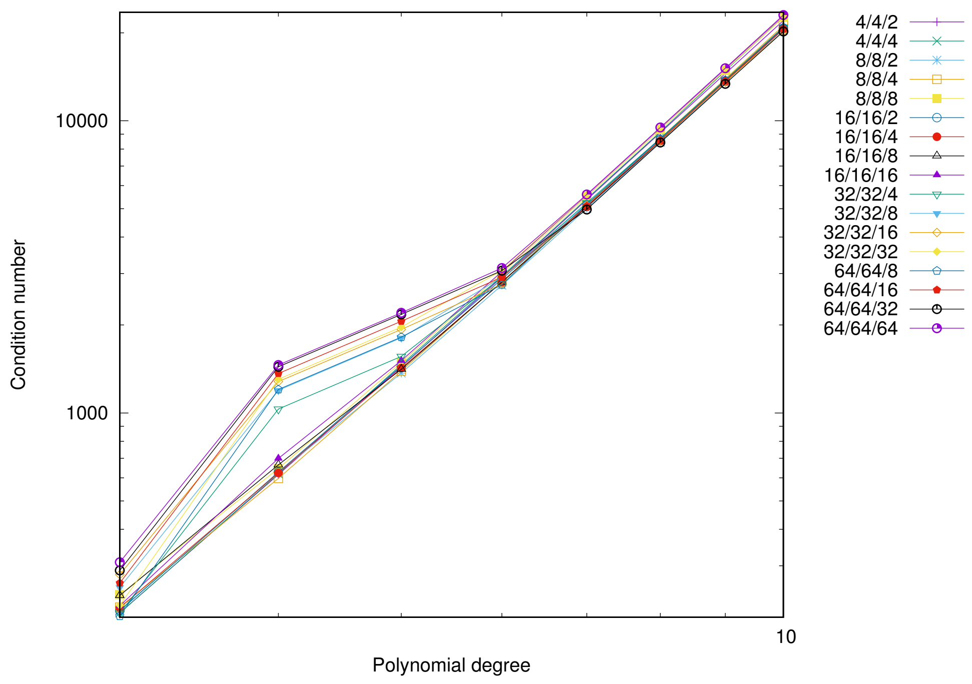


ADLP basis functions.

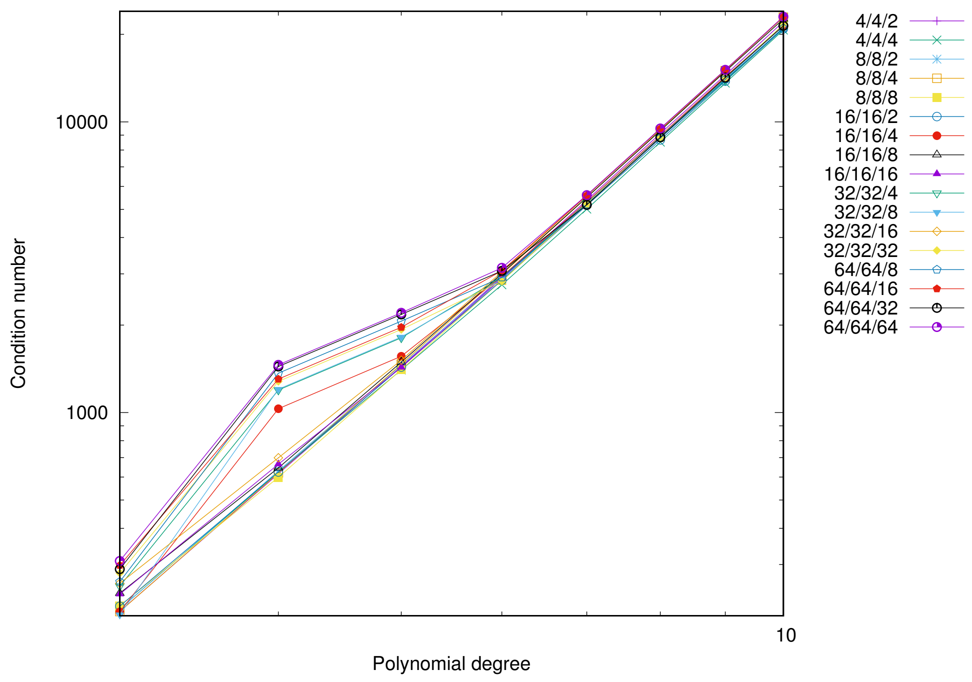


C1 basis functions.

Figure 4.6: Iteration number, for problem **C**, with solver **SB**. Logarithmic scale on all axes.

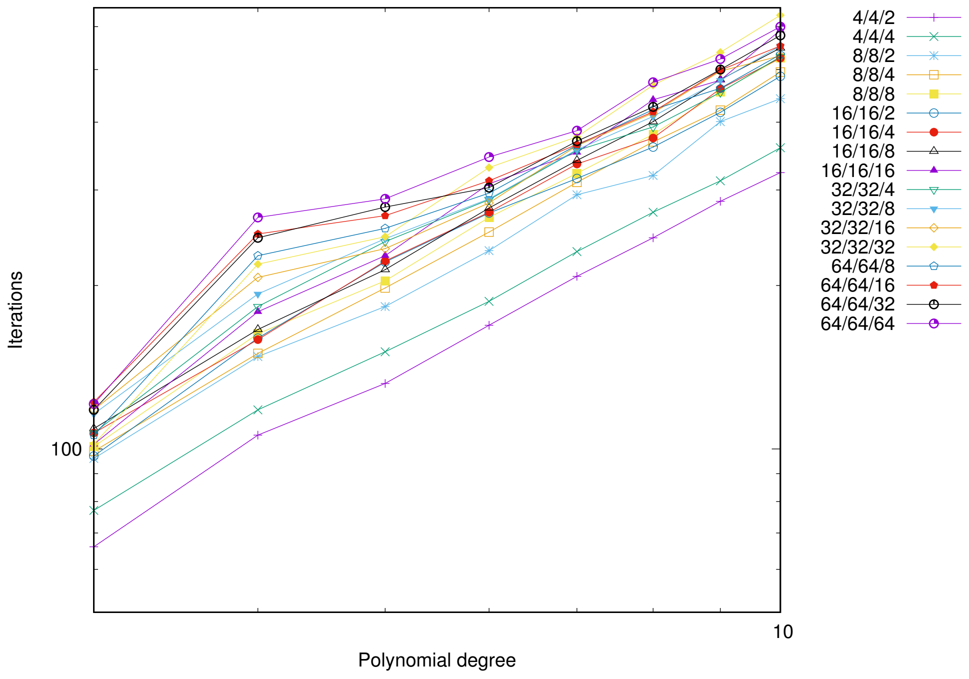


ADLP basis functions.

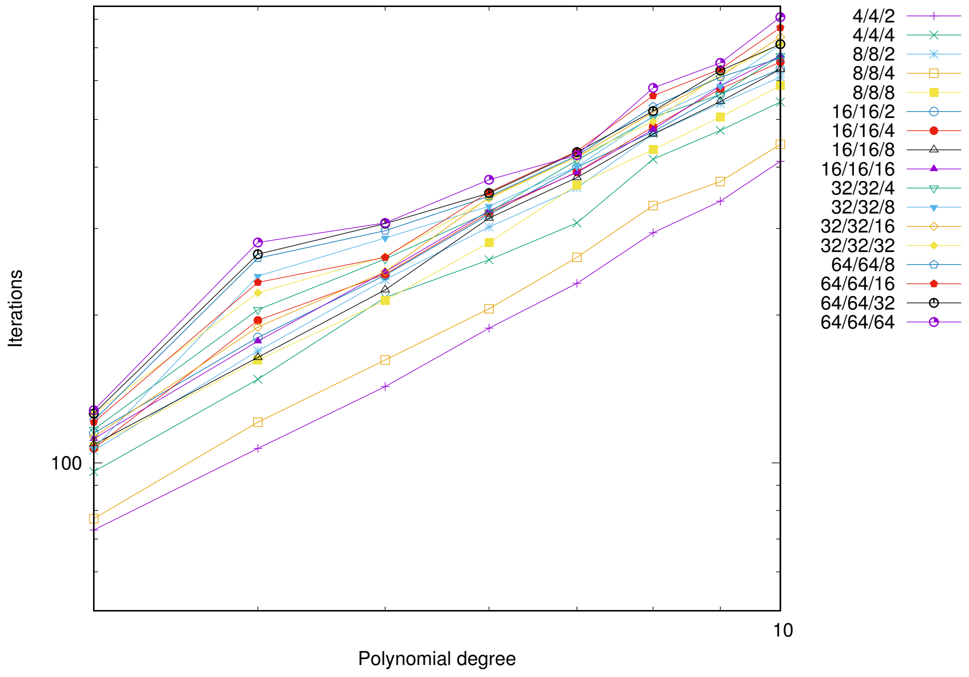


C1 basis functions.

Figure 4.7: Condition number estimates, for problem **C**, with solver **SC**. Logarithmic scale on all axes.



ADLP basis functions.



C1 basis functions.

Figure 4.8: Iteration number, for problem **C**, with solver **SC**. Logarithmic scale on all axes.

Figures 4.5 and 4.7 show that the solvers **SB** and **SC** are spectrally equivalent, as theorems 3.4.10 and 3.4.15 show.

4.6 Choice of the basis function effect

Choice of the basis function can give considerable gain in terms of the computation speed. It can be demonstrated by performing the following analysis. Taking the **CG** algorithm's computational complexity per iteration applied for **ADLP** based system $T^{ADLP}_{CG,IT,par}$ (3.25) and the one for **C1** $T^{C1}_{CG,IT,par}$

$$\Theta := \frac{T^{ADLP}_{CG,IT,par}}{T^{C1}_{CG,IT,par}}, \tag{4.2}$$

which we will call "efficiency gain". The value of the efficiency gain Θ represents the ratio of the times between that of using **ADLP** basis functions and with using **C1** basis functions. It shows how much faster the conventional parallel CG solver **SA** performed an iteration with **C1** basis function against itself with **ADLP** basis function.

The results of the formula (4.2) are shown on the figure 4.9 .

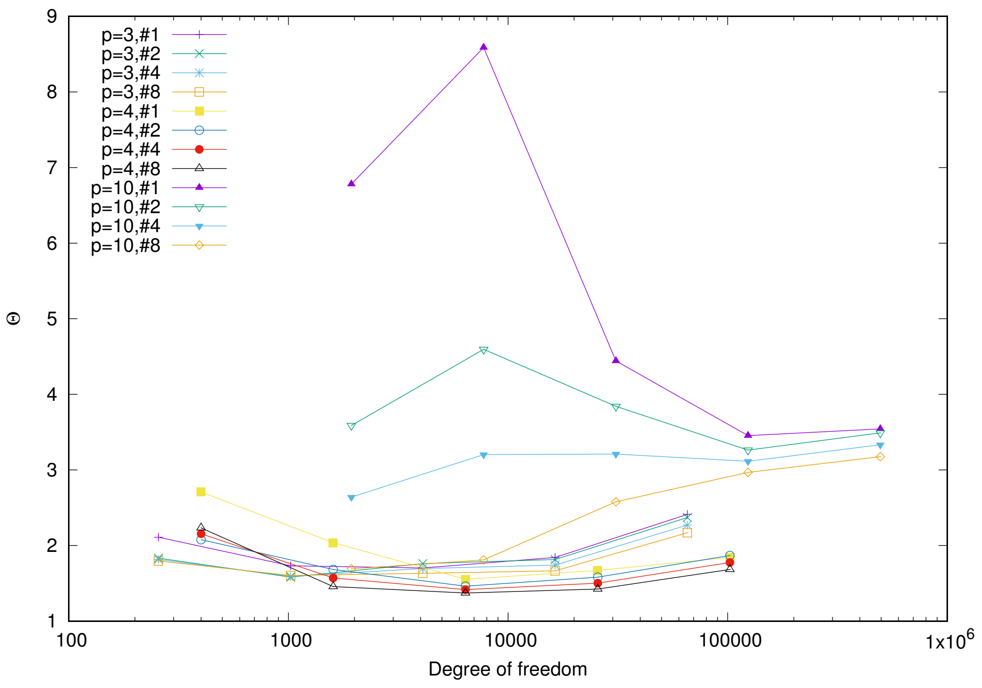


Figure 4.9: Efficiency gain for problem **C** with solver **SA**. Comparison between **ADLP** and **C1** basis functions. Logarithmic scale on the Degree of Freedom axis. **MPI** execution.

In figure 4.9 where comparison is done between the time required by the **CG** solver **SA** for the system which uses **ADLP** basis functions and **C1** basis functions. Comparison shown is for the polynomial degrees $p = 3, 4, 10$, for $1, 2, 4$, and 8 threads. It can be observed that, the efficency gain Θ is changing the order with the polynomial approximation degree. Additionally, we can conclude, that there's a slight growth in Θ with higher degrees of freedom.

Additionally, on the same figure, we observe that reduction for eight processes is less than the one for four processes. This is observed for all parallel executions. This is due to the fact, that solver with **ADLP** basis functions also improves for parallel execution (requires less time per iteration for more parallel processes). This in turn slightly decreases the effect from the use of **C1** basis functions.

4.7 Solver timing comparison

As it was argued in section **Multigrid method as a coarse level solver, 3.4**, solver **SC**, which uses constant polynomial degree $q = 1$ would require less operations (3.92) to solve the coarse level problem. This can be observed for both **ADLP** basis functions and **C1** basis functions in figures 4.10 and 4.11 correspondingly.

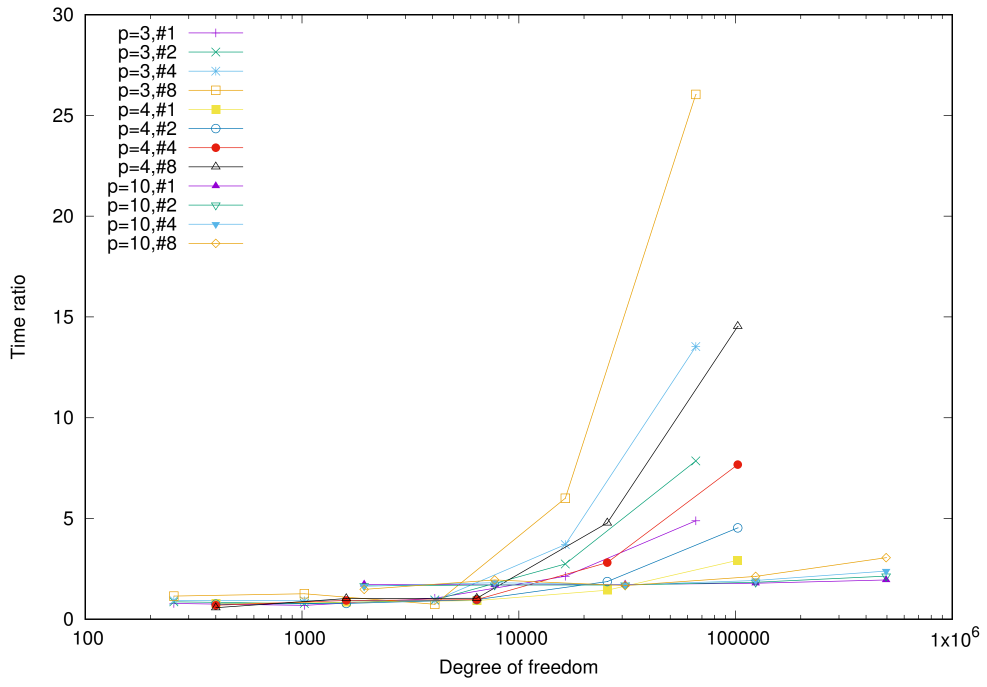


Figure 4.10: Efficiency gain for the solution of the problem C by the solvers **SC** and **SB**. Logarithmic scales on the Degree of Freedom axis. **MPI** execution. ADLP basis functions.

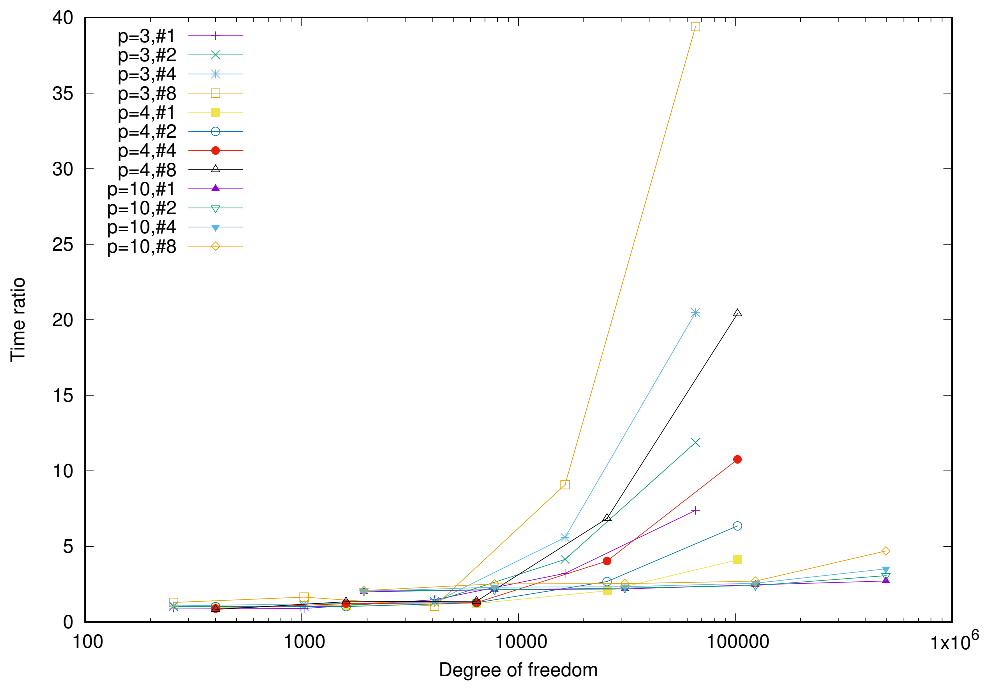


Figure 4.11: Efficiency gain for the solution of the problem C by the solvers **SC** and **SB**. Logarithmic scales on the Degree of Freedom axis. **MPI** execution. C1 basis functions.

In figures 4.10 and 4.11 we observe the ratio of the times between that of using the parallel solver with Additive two-layer non-overlapping Schwarz method as a preconditioner (solver **SB**) and that of using Multigrid method as a coarse level solver for the aforementioned preconditioner (solver **SC**).

4.8 Effect of parallelization

In this section we are going to analyse the effect of the parallel implementation of the problems.

The results shown below demonstrate the speed up and general gain per number of threads. Although both solver use parallel data structure distributed amongst processes for the execution, solver **SC** demonstrates higher gain in time reduction. Results for the time required by the preconditioning show stability. It can be observed, that, generally, solver **SC** tends to the exact parallelism on the fine grid part for higher approximation polynomial degrees and bigger mesh, e.g. speed up scales with degree of freedom of the problem, for higher degrees of freedom.

4.8.1 Numerical results in 2d

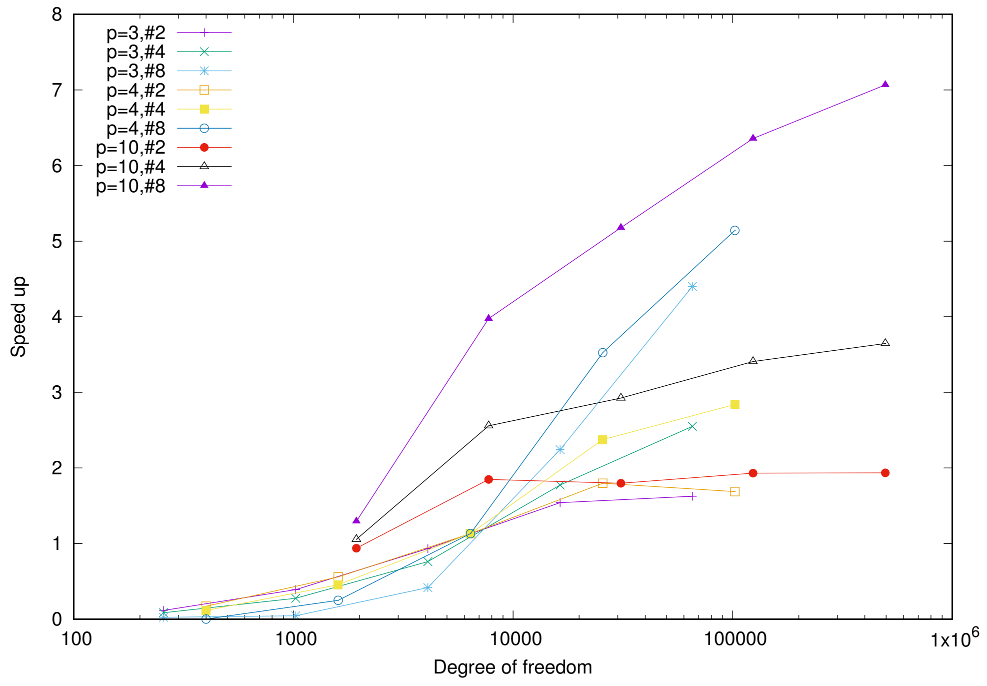
We compare the non-parallel execution time with the results obtained with the parallel version. It is clearly seen that the **C1** polynomials yield mostly exact parallelization for the higher approximation polynomial degree. Reducing the size of the coarse grid level reduces the execution time.

Figures 4.12-4.14 demonstrate the schematic comparison of the speed up gained for parallel executions for different solvers, such as conventional CG **SA** and preconditioned ones with ASM **SB** and ASM+MG **SC**. It can be observed, that the speed up for the CG solver with ASM **SB** does not reach full parallelization for higher polynomial degrees. It can be explained by the fact that preconditioned part of the solver is also not embarrassingly parallel, as it contains the coarse level part which grows with growth of the problem size. This can be seen on figure 4.15.

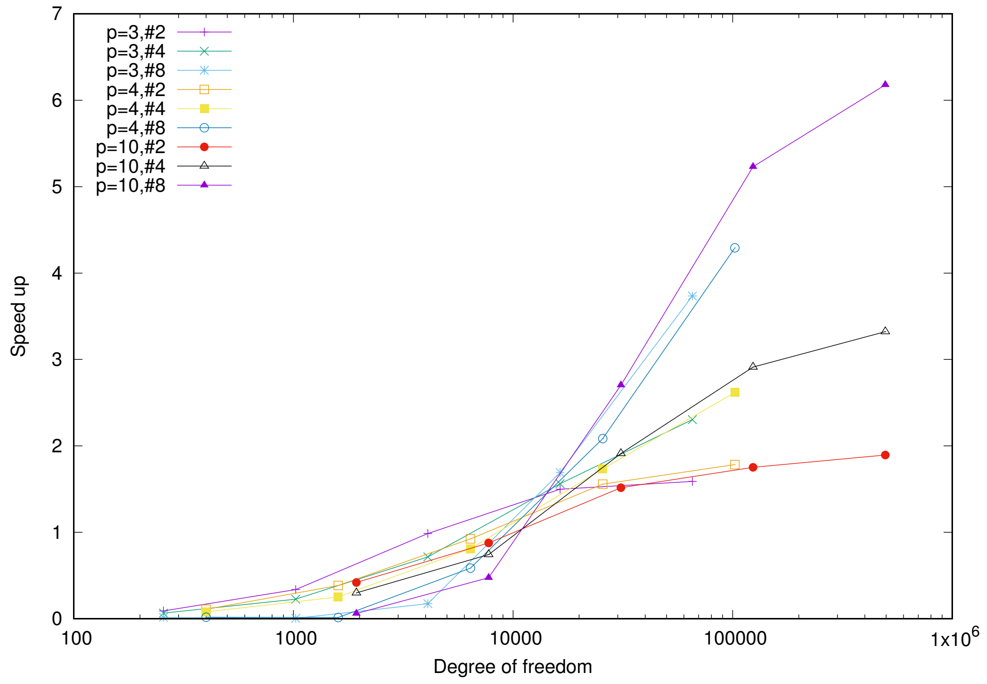
At the same time, on figure 4.16 we observe that for higher degree of freedom total preconditioning time speed up for ASM+MG solver **SC** becomes embarrassingly parallel.

Figures 4.17 demonstrate speed up on the fine grid of the additive two layer non-overlapping Schwarz method. This is true for both preconditioners **SB** and **SC** as they have the same fine grid solver.

On figures 4.18-4.26 we can clearly observe, that full speed up is achievable for higher polynomial degrees. Parallel version is slower when the degree of freedom is not so high.

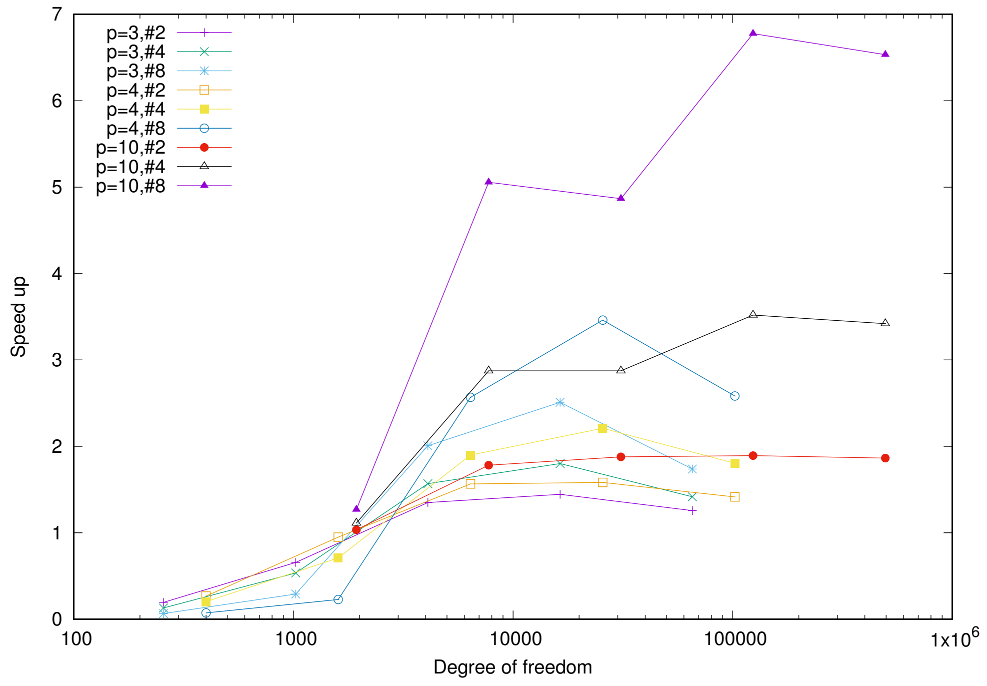


ADLP basis functions.

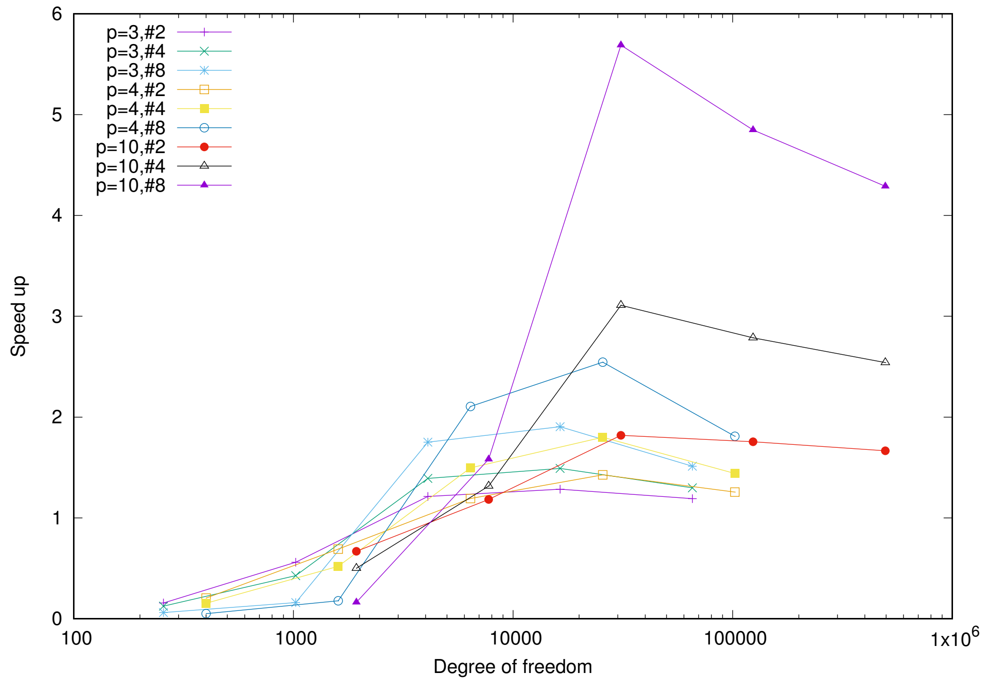


C1 basis functions.

Figure 4.12: CG solver **SA** speed up with increase in number of threads. Logarithmic scale on Degree of Freedom axis. **MPI** execution.

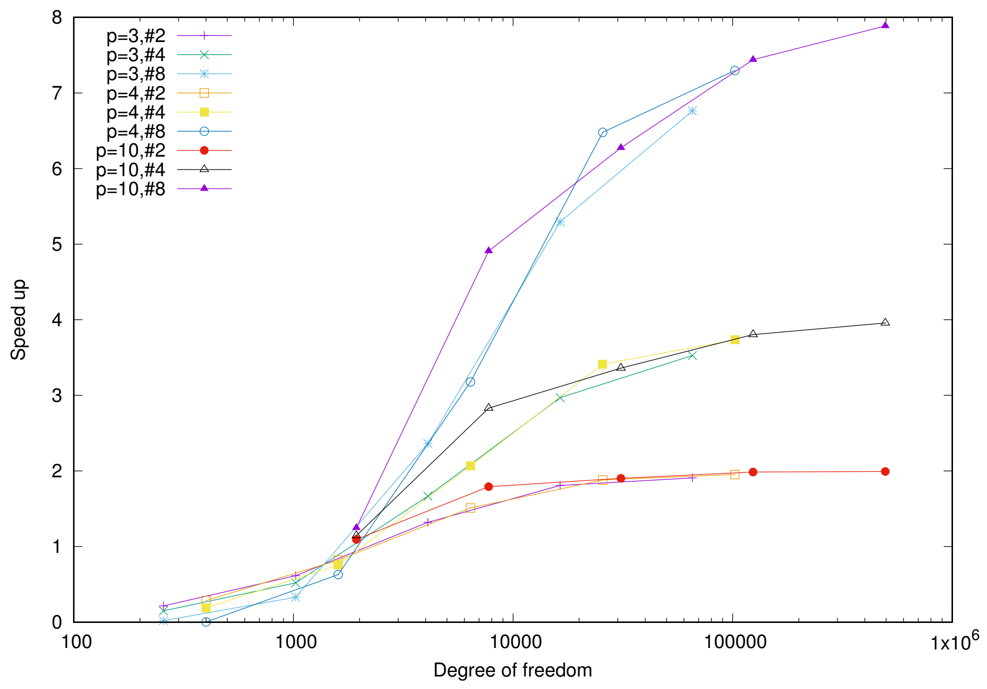


ADLP basis functions.

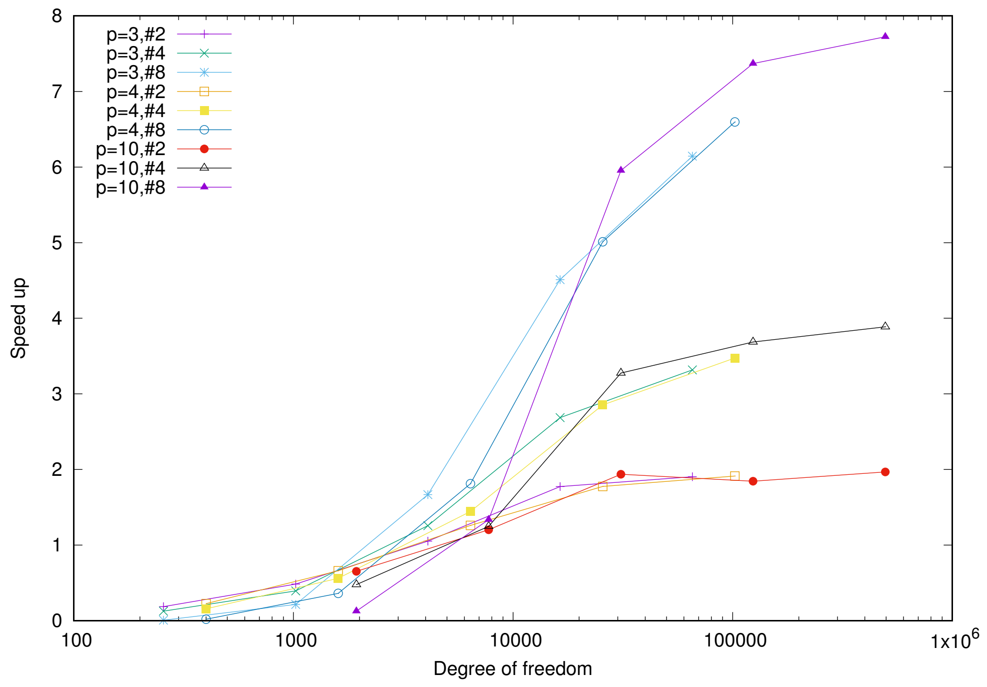


C1 basis functions.

Figure 4.13: CG solver with ASM **SB** speed up with increase in number of threads. Logarithmic scale on Degree of Freedom axis. **MPI** execution.

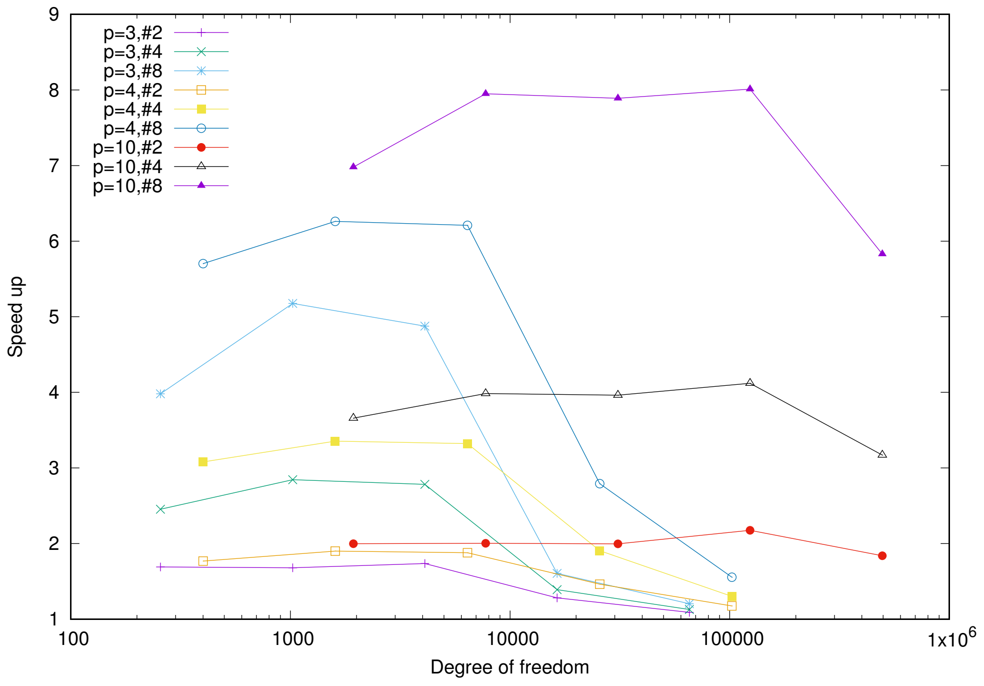


ADLP basis functions.

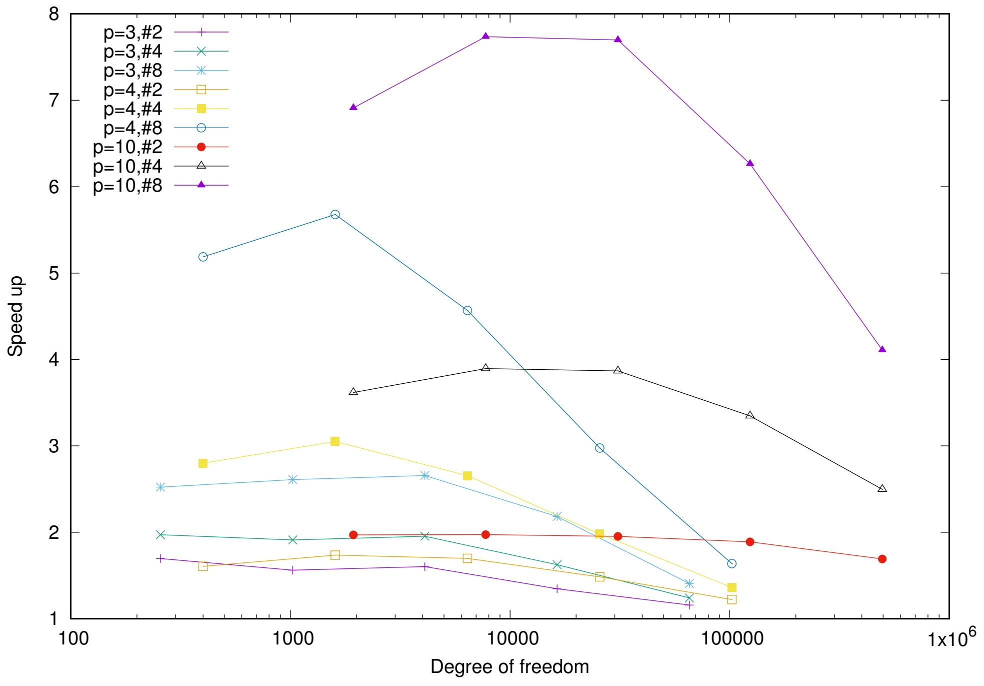


C1 basis functions.

Figure 4.14: CG solver with ASM+MG **SC** speed up with increase in number of threads. Logarithmic scale on Degree of Freedom axis. **MPI** execution.

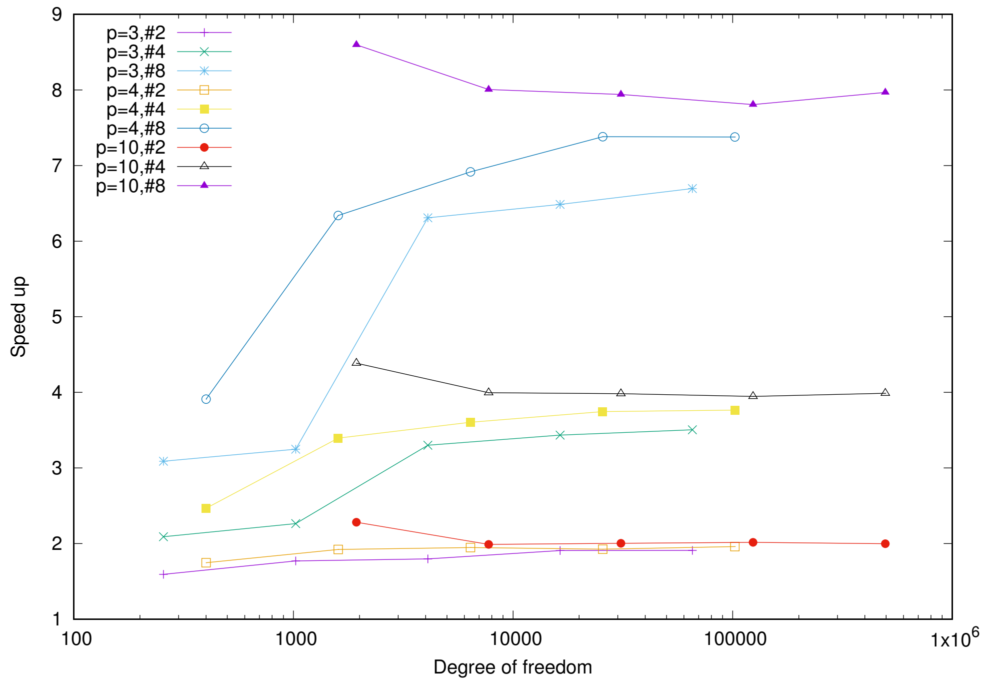


ADLP basis functions.

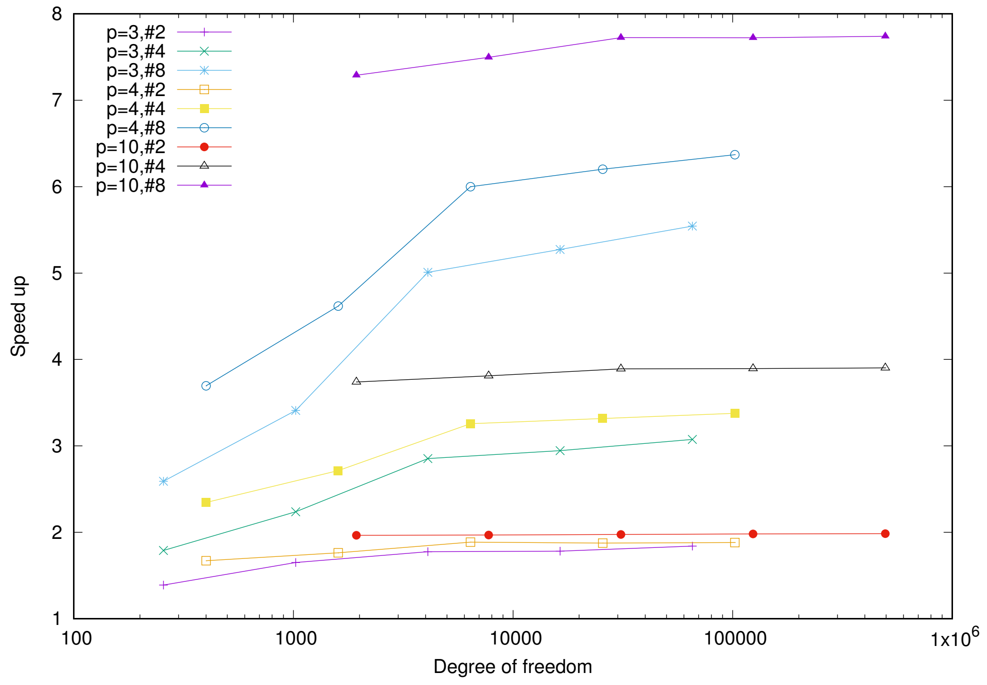


C1 basis functions.

Figure 4.15: Speed up in total preconditioning time with increase in number of threads for Additive Schwarz Method for solver **SB** . Logarithmic scale on Degree of Freedom axis. **MPI** execution.

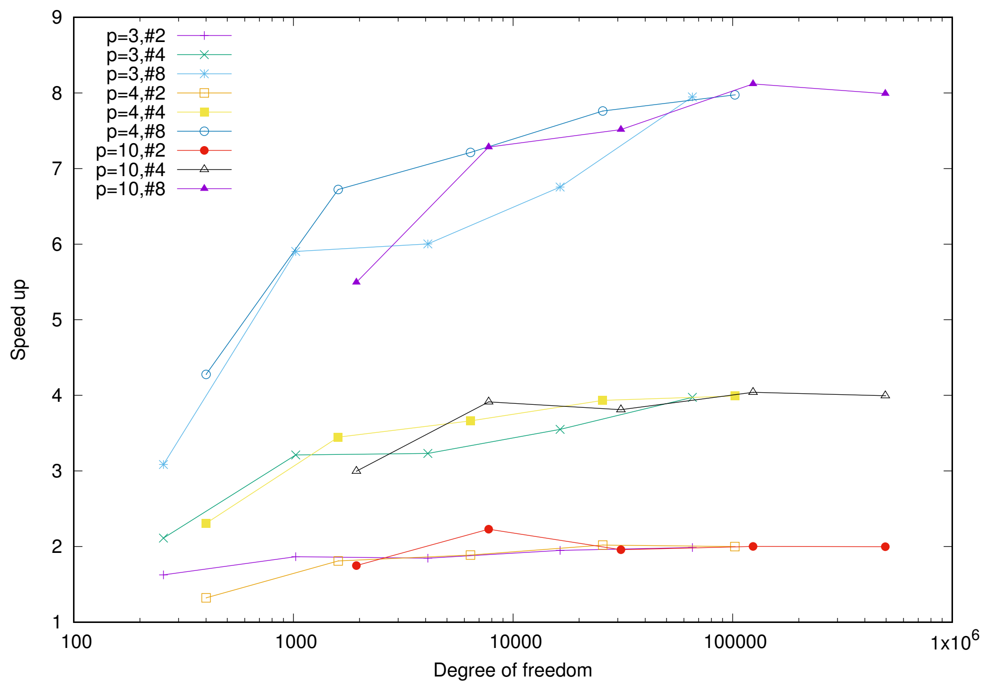


ADLP basis functions.

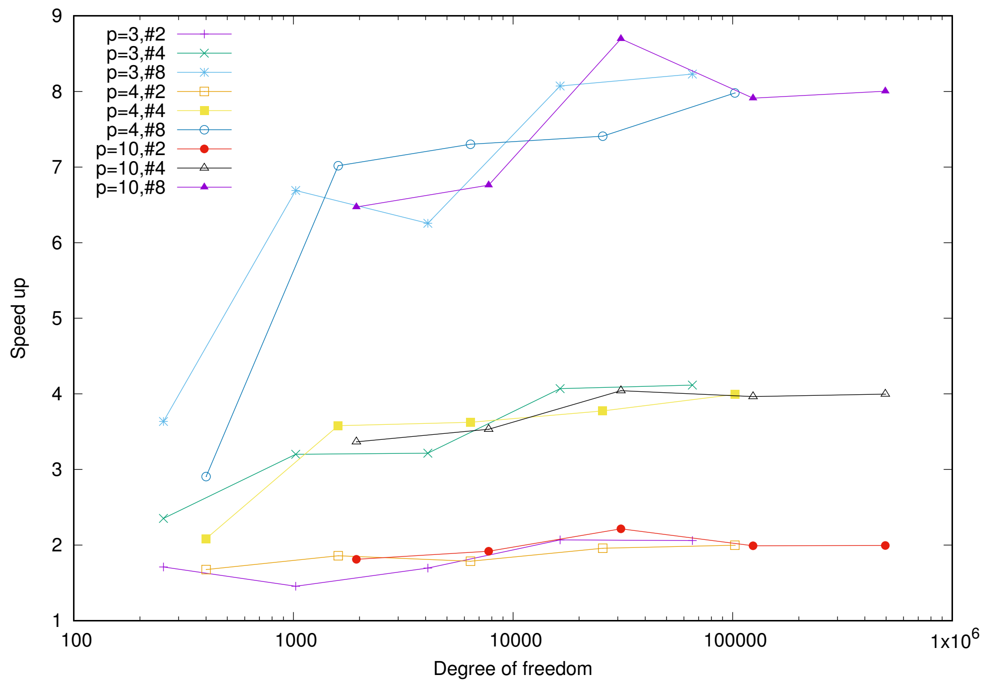


C1 basis functions.

Figure 4.16: Speed up in total preconditioning time with increase in number of threads for **ASM** with Multigrid Method as a coarse level solver **SC** . Logarithmic scale on Degree of Freedom axis. **MPI** execution.

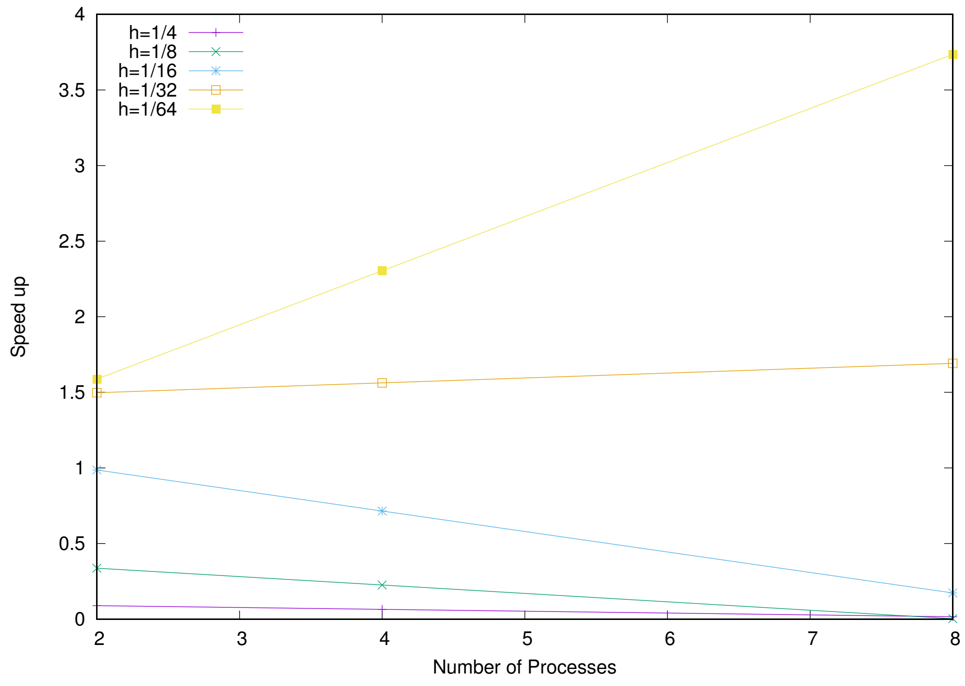


ADLP basis functions.

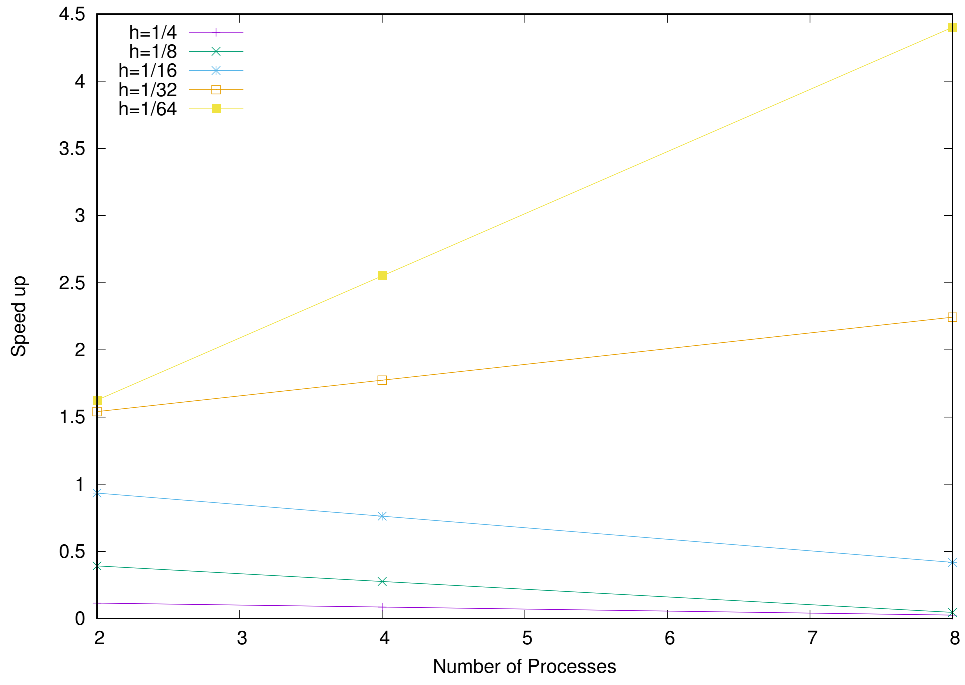


C1 basis functions.

Figure 4.17: CG solver with ASM **SB** and ASM+MG **SC** speed up in fine grid problem with increase in number of threads. Logarithmic scale on Degree of Freedom axis. **MPI** execution.

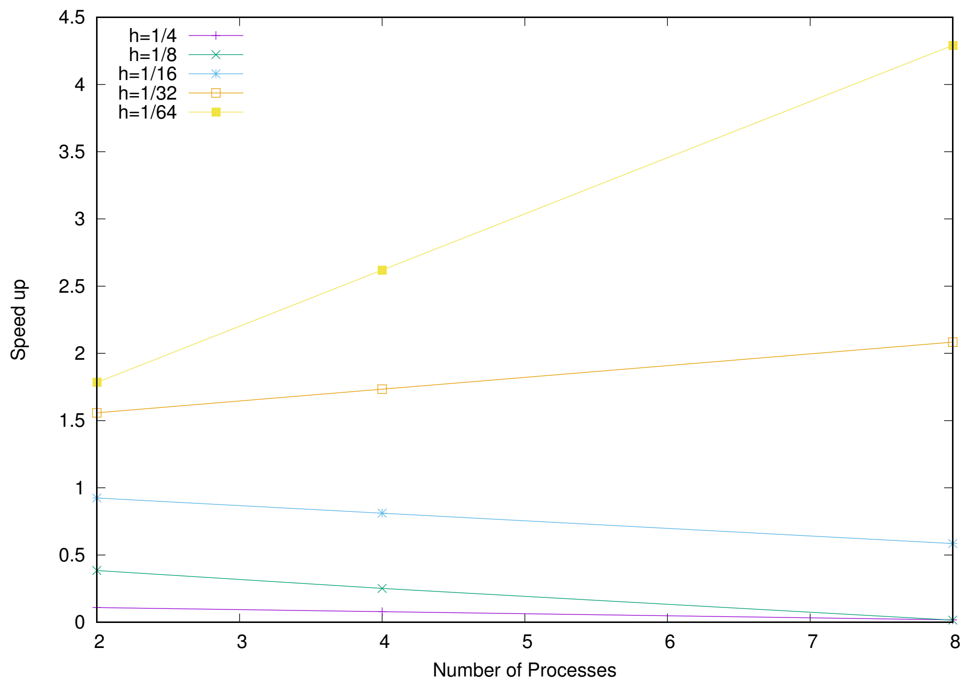


ADLP basis functions.

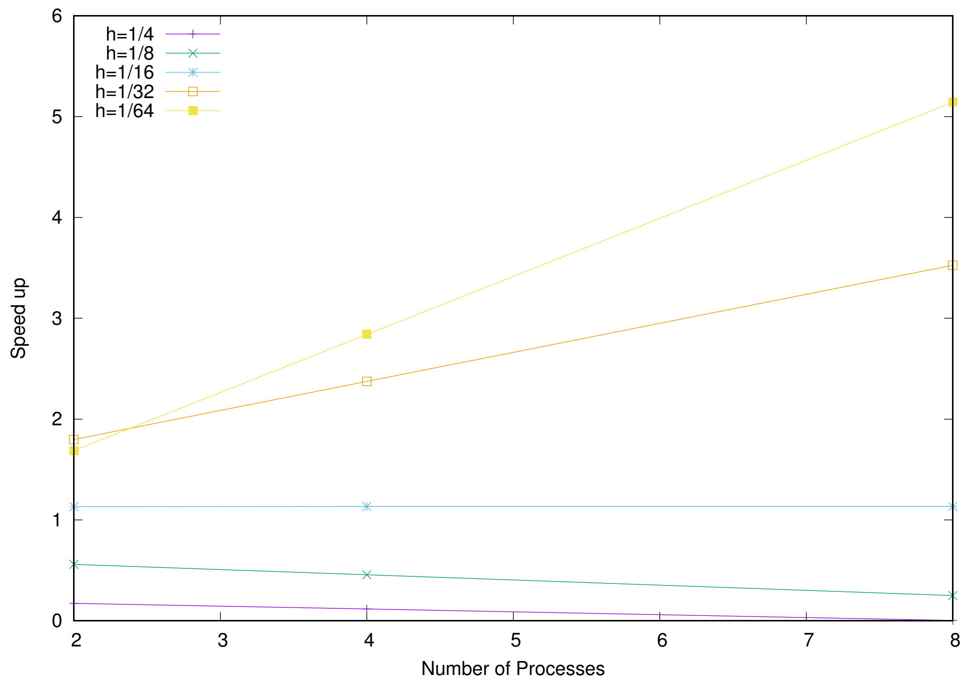


C1 basis functions.

Figure 4.18: CG solver **SA** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 3$. **MPI** execution.

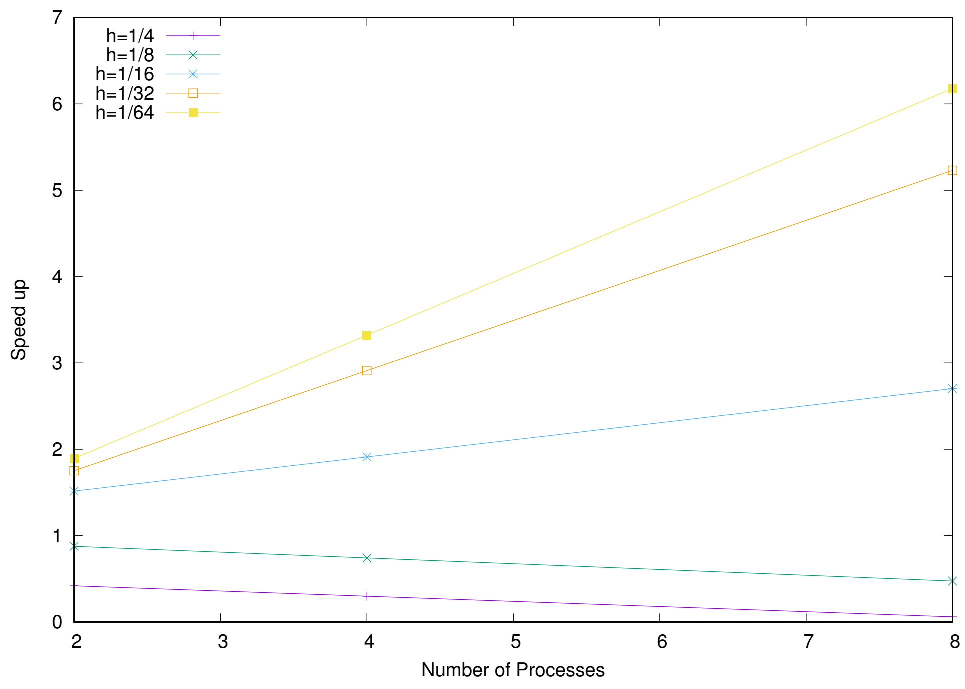


ADLP basis functions.

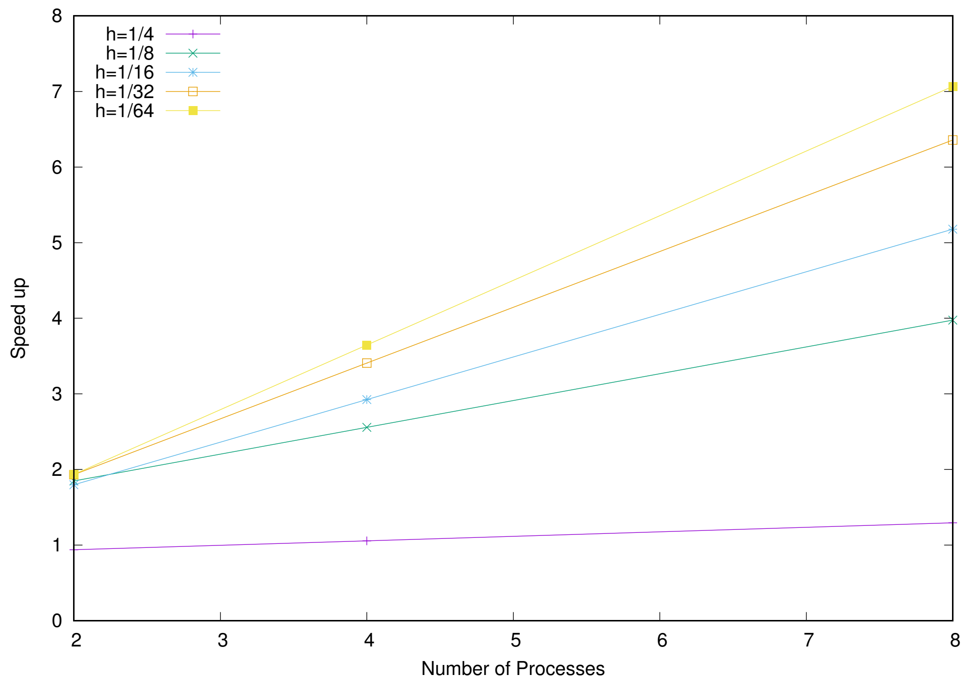


C1 basis functions.

Figure 4.19: CG solver **SA** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 4$. **MPI** execution.

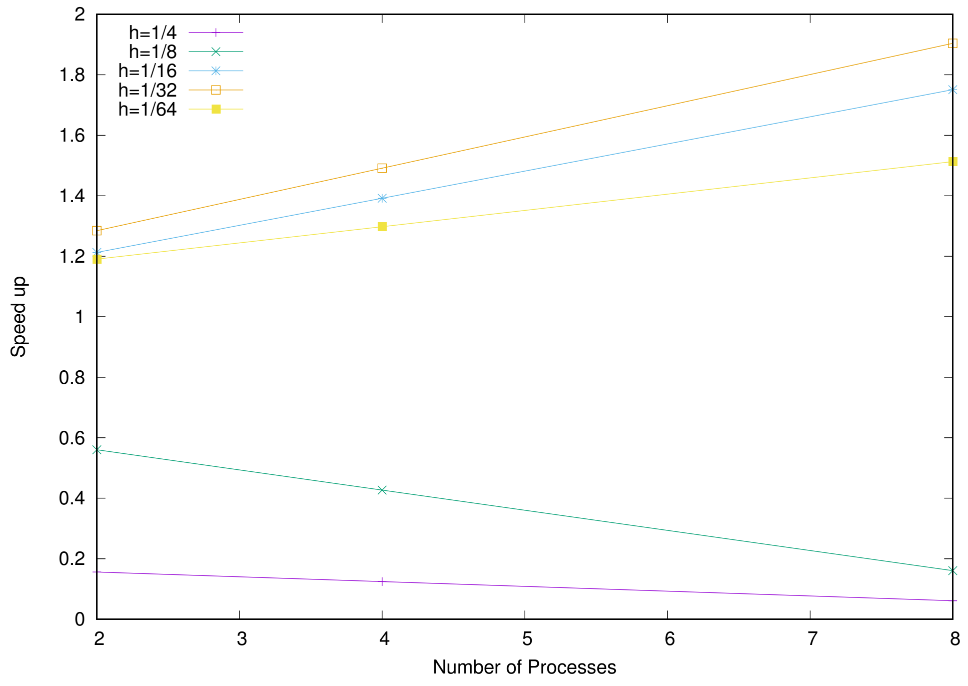


ADLP basis functions.

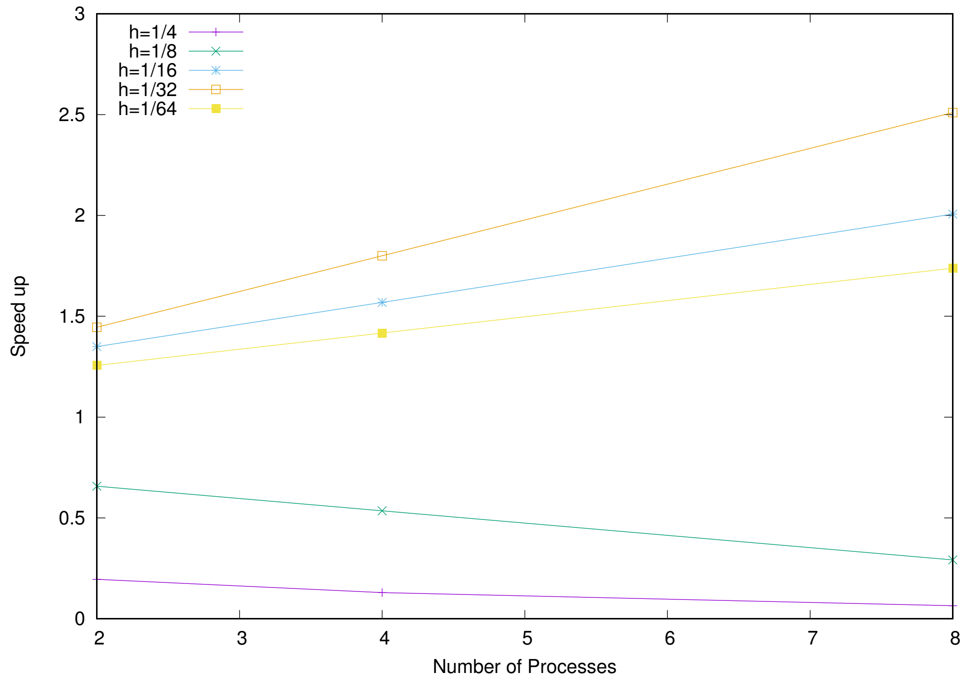


C1 basis functions.

Figure 4.20: CG solver **SA** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 10$. **MPI** execution.

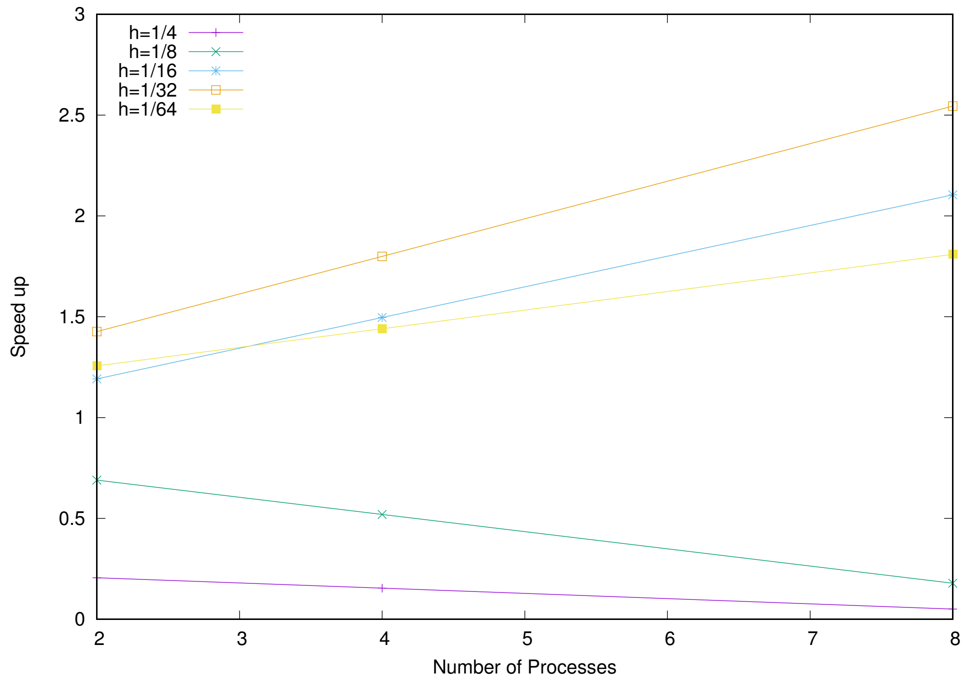


ADLP basis functions.

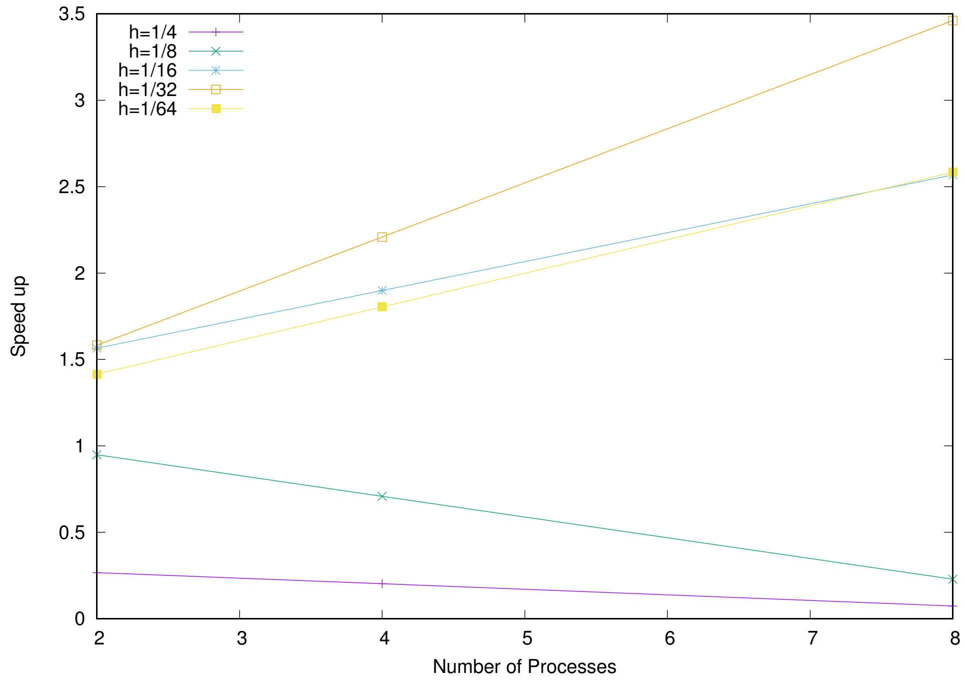


C1 basis functions.

Figure 4.21: CG solver **SB** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 3$. **MPI** execution.

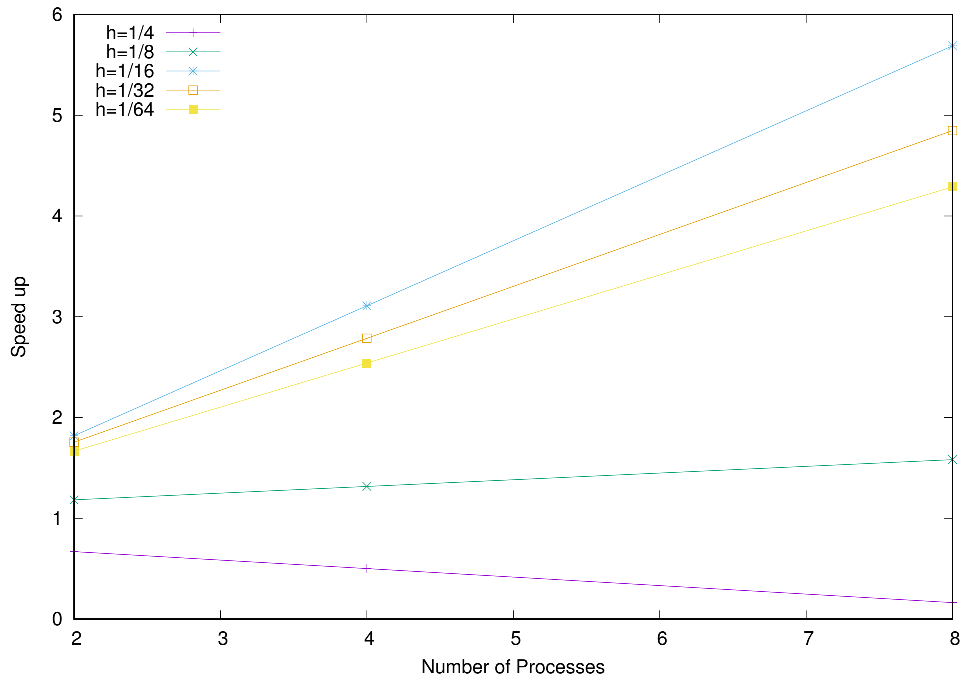


ADLP basis functions.

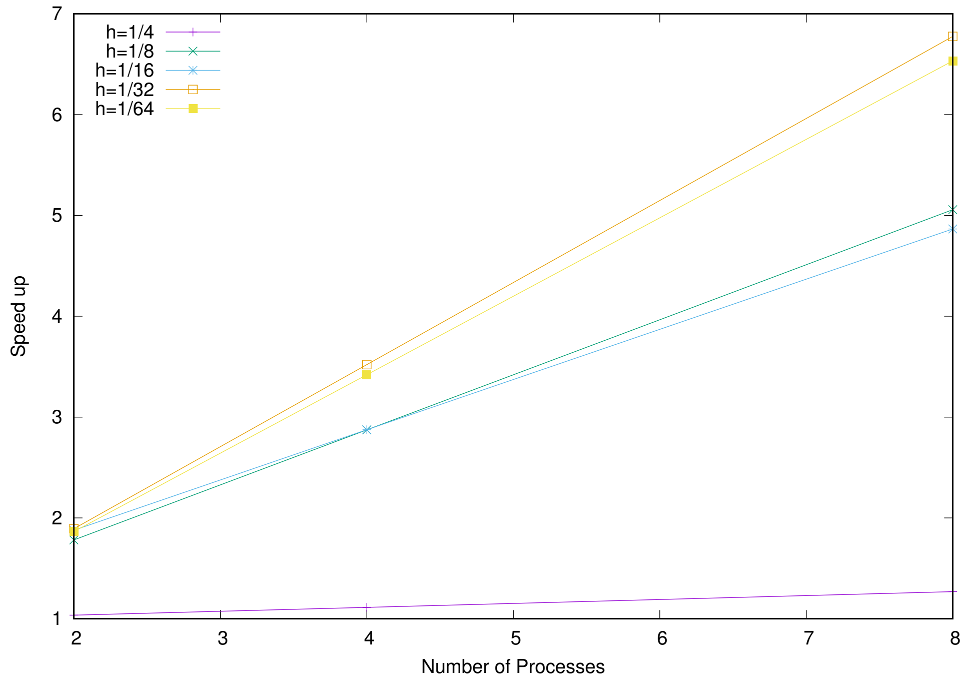


C1 basis functions.

Figure 4.22: CG solver **SB** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 4$. **MPI** execution.

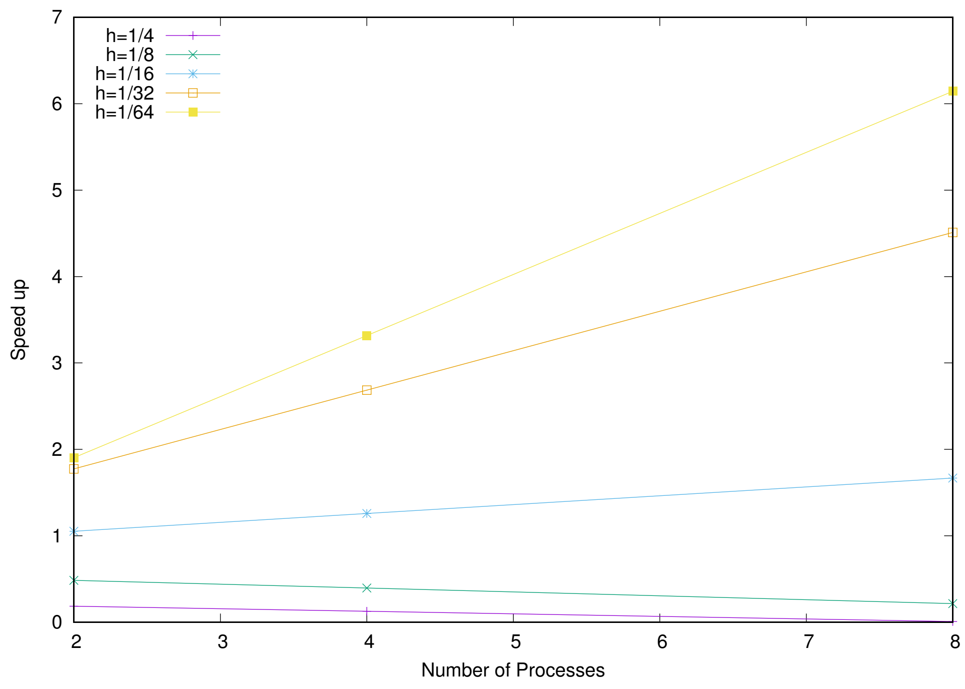


ADLP basis functions.

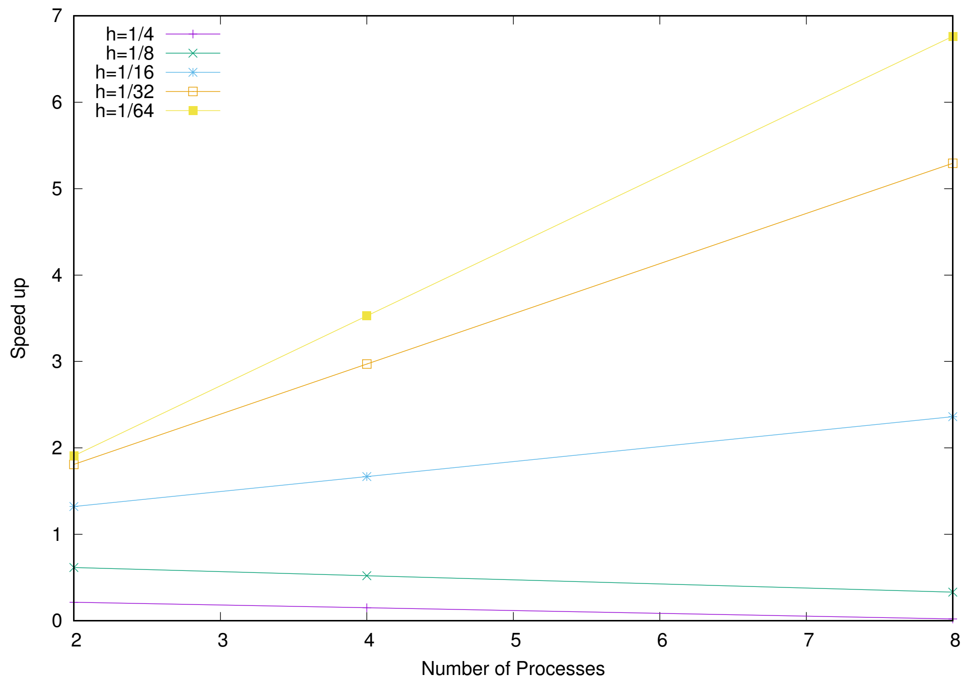


C1 basis functions.

Figure 4.23: CG solver **SB** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 10$. **MPI** execution.

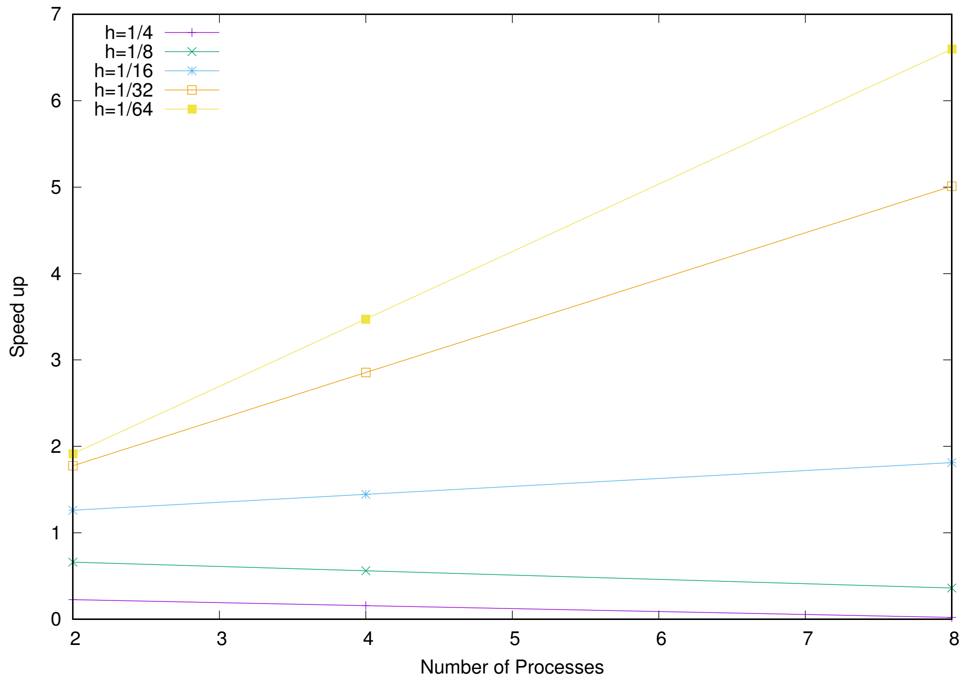


ADLP basis functions.

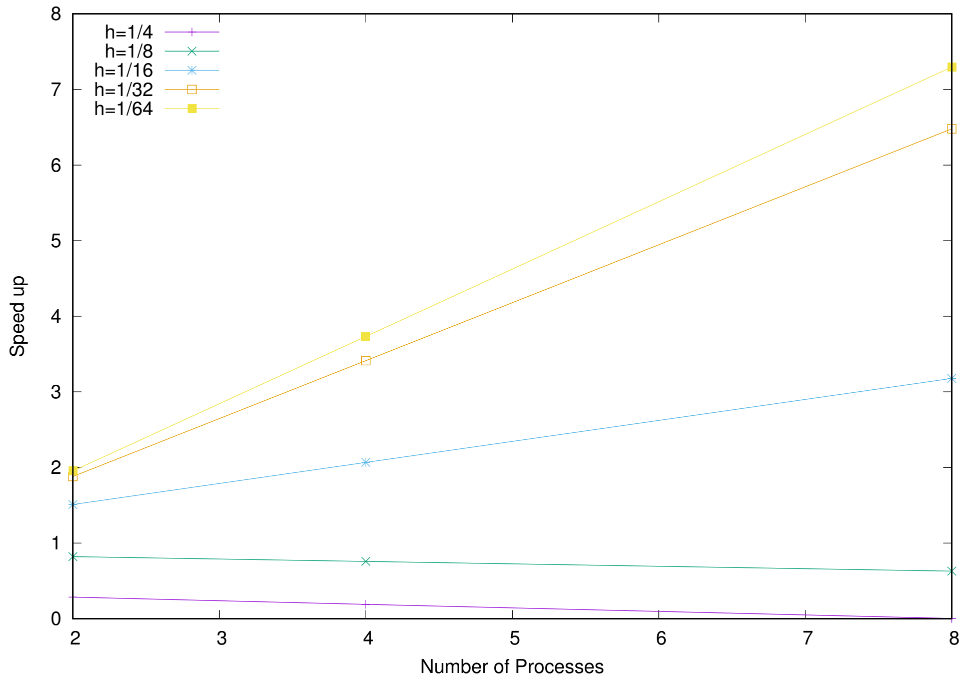


C1 basis functions.

Figure 4.24: CG solver **SC** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 3$. **MPI** execution.

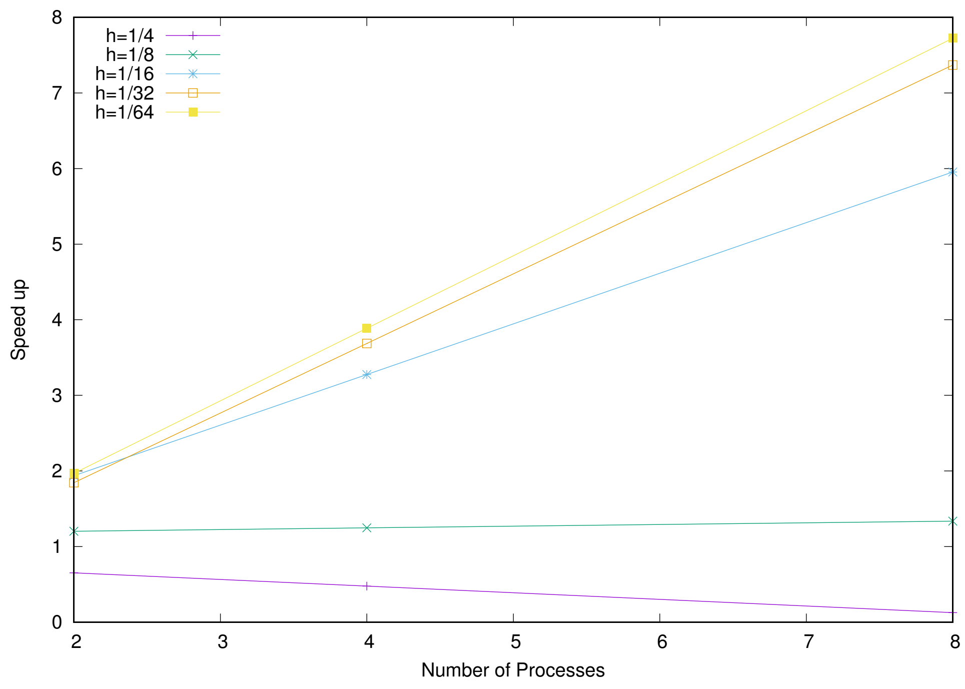


ADLP basis functions.

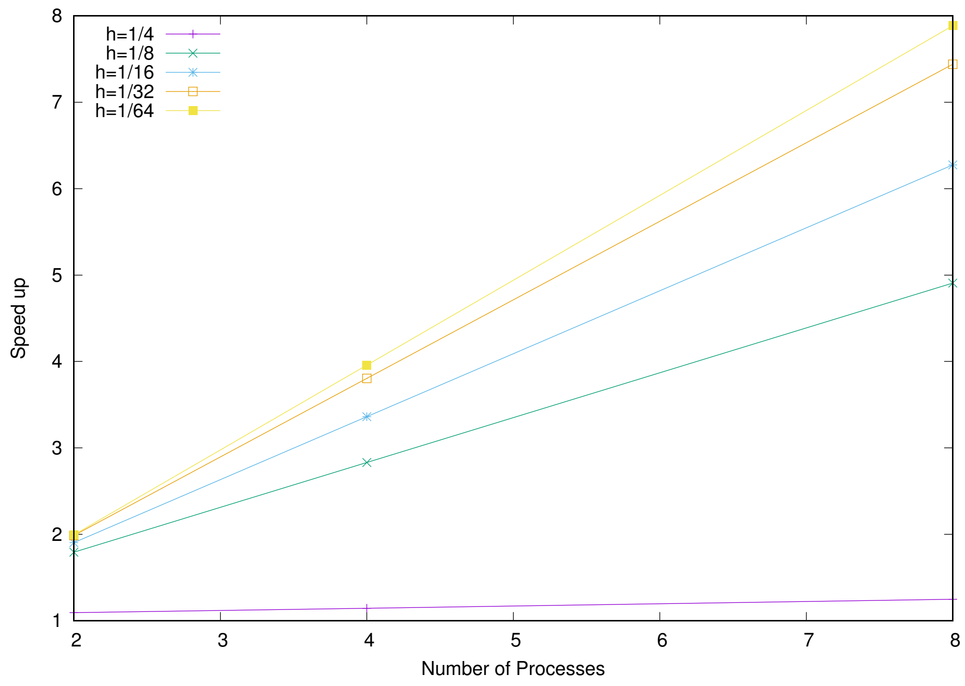


C1 basis functions.

Figure 4.25: CG solver **SC** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 4$. **MPI** execution.



ADLP basis functions.



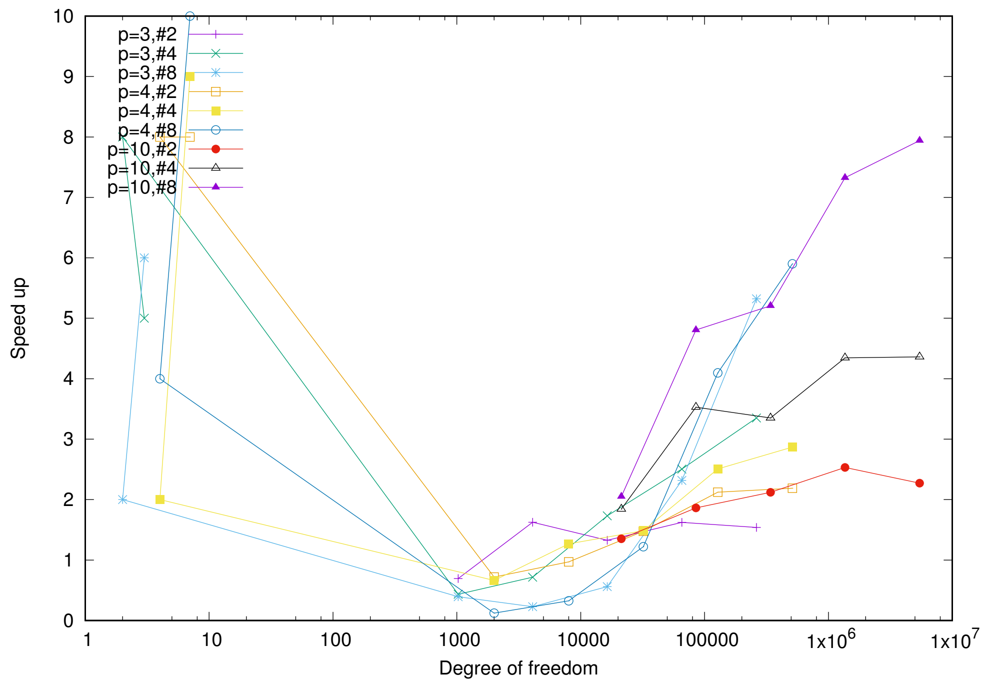
C1 basis functions.

Figure 4.26: CG solver **SC** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 3$. **MPI** execution.

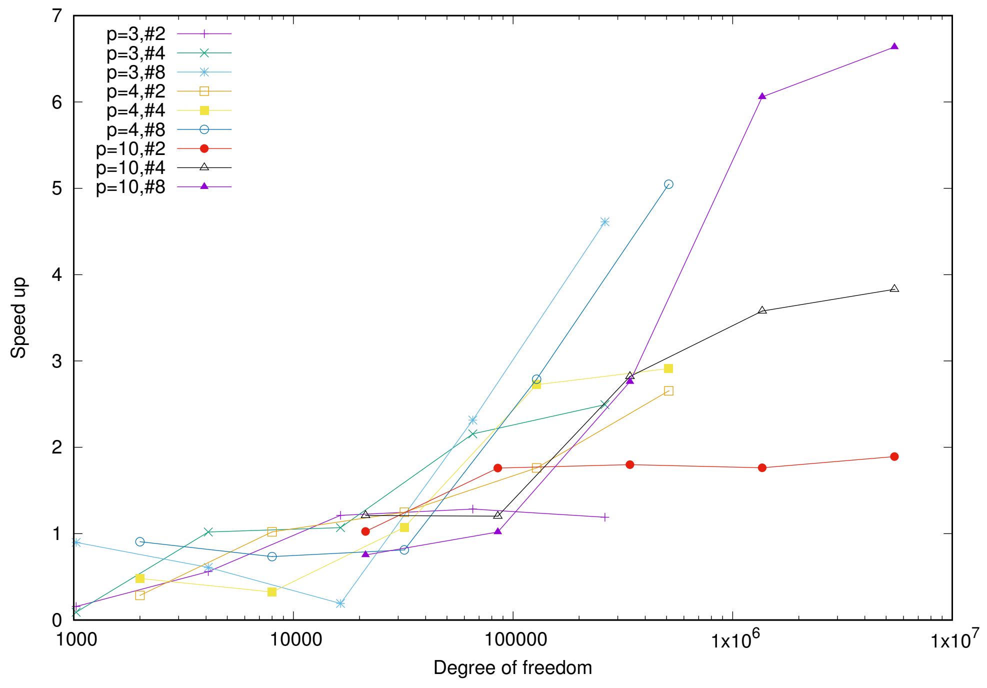
4.8.2 Numerical results in 3d

Results for the three-dimensional problem **A3** show that coarse level solver heavily impacts the execution time. This can be seen by observing the speed up in the figures 4.33,4.34 and 4.35 which show the speed up gained by the solver **SB**. And on figures 4.36,4.37 and 4.38 which show the results for solver **SC**.

On figures 4.27-4.29 we observe the same effect on the speed up for the higher polynomial approximation degrees. It is trivial that there is not much gain in parallelization for problems with smaller degree of freedom.

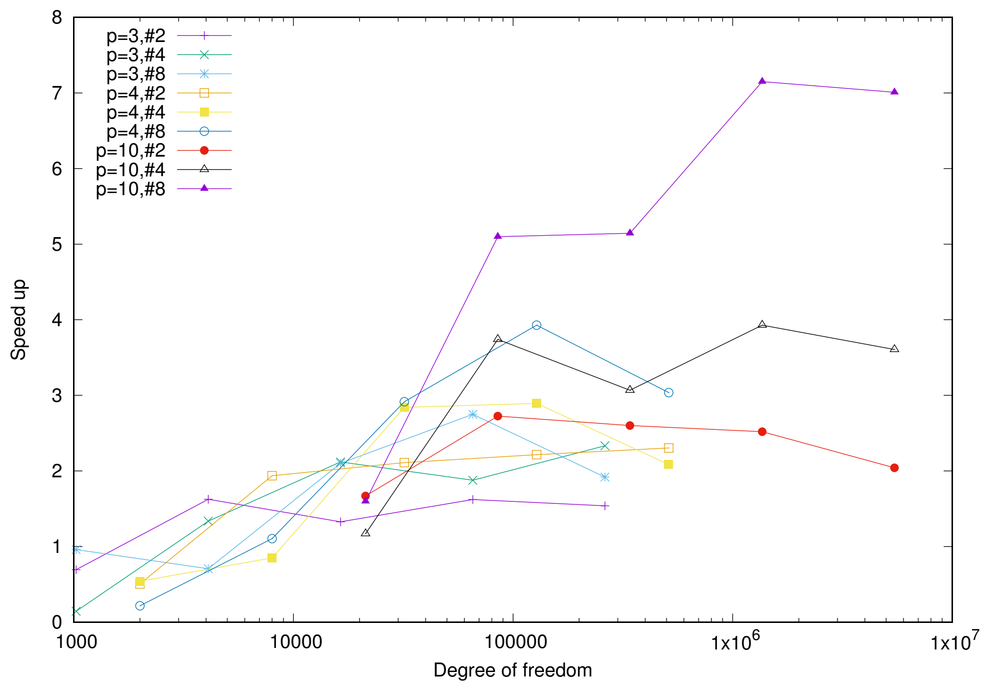


ADLP basis functions.

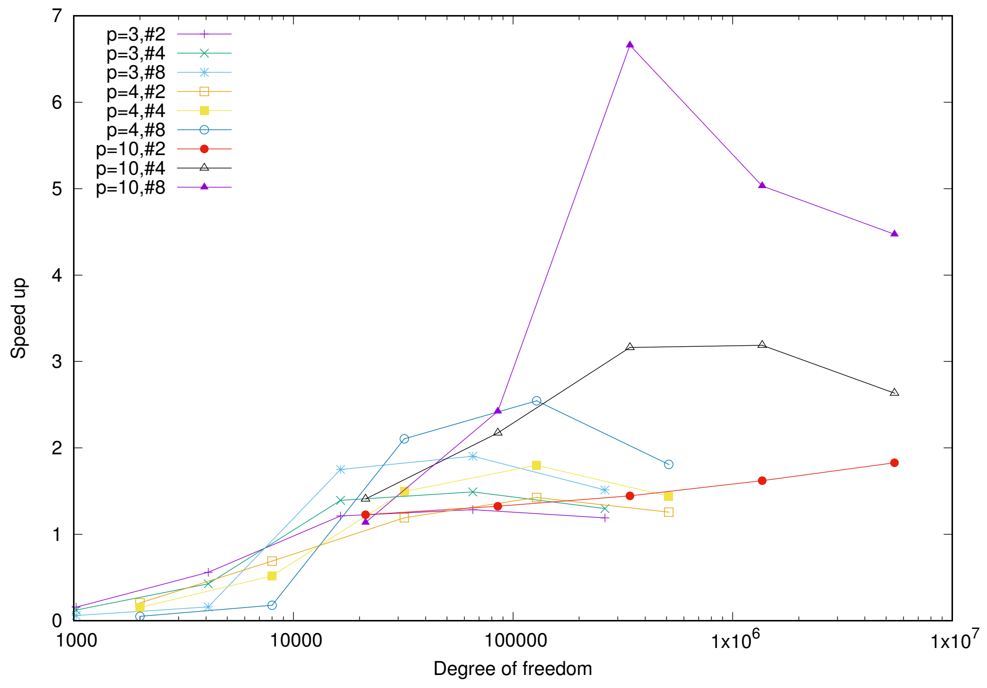


C1 basis functions.

Figure 4.27: CG solver **SA** speed up with increase in number of threads. Logarithmic scale on Degree of Freedom axis. **MPI** execution. Problem **A3**.

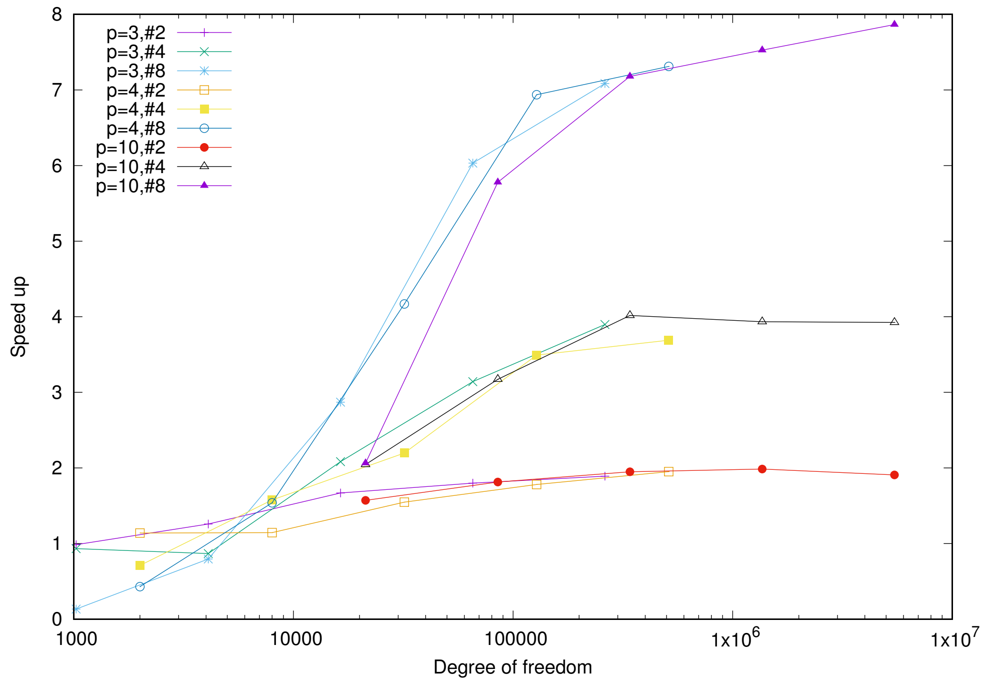


ADLP basis functions.

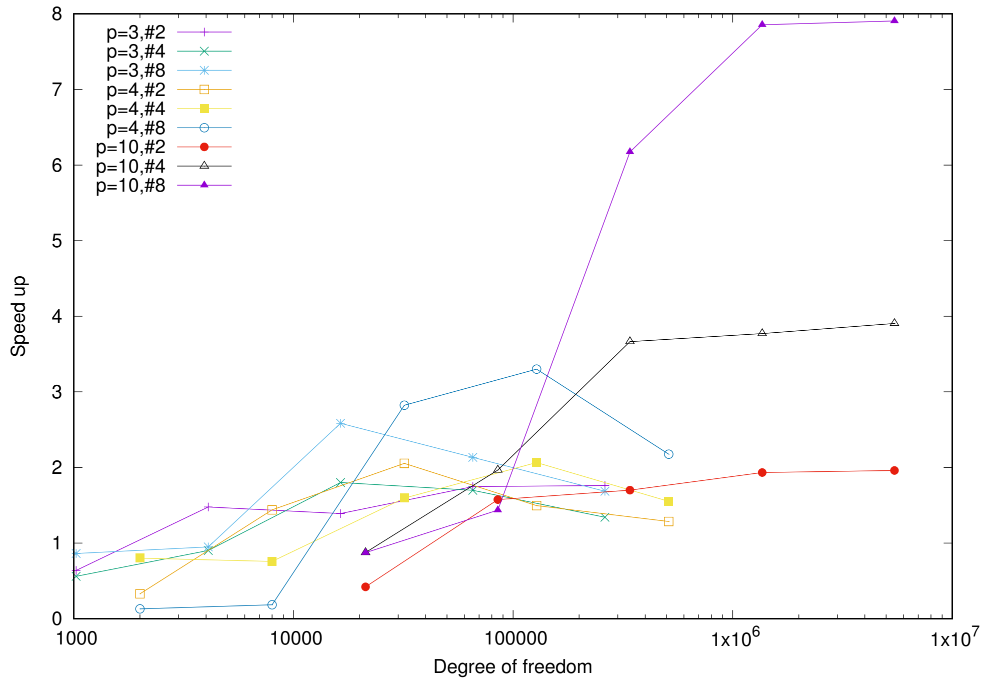


C1 basis functions.

Figure 4.28: CG solver with ASM **SB** speed up with increase in number of threads. Logarithmic scale on Degree of Freedom axis. **MPI** execution. Problem **A3**.

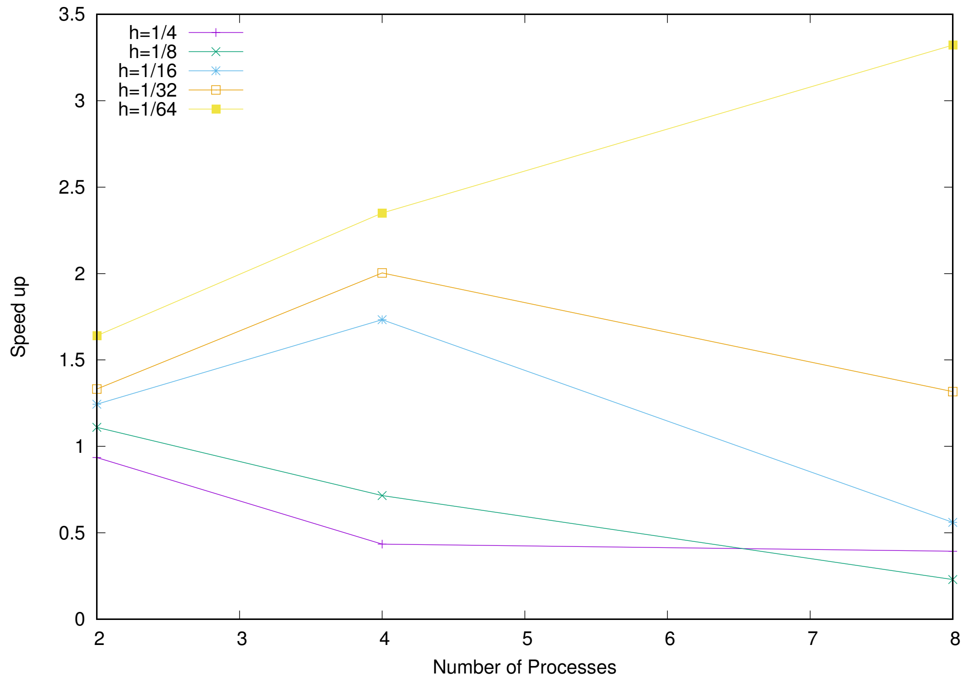


ADLP basis functions.

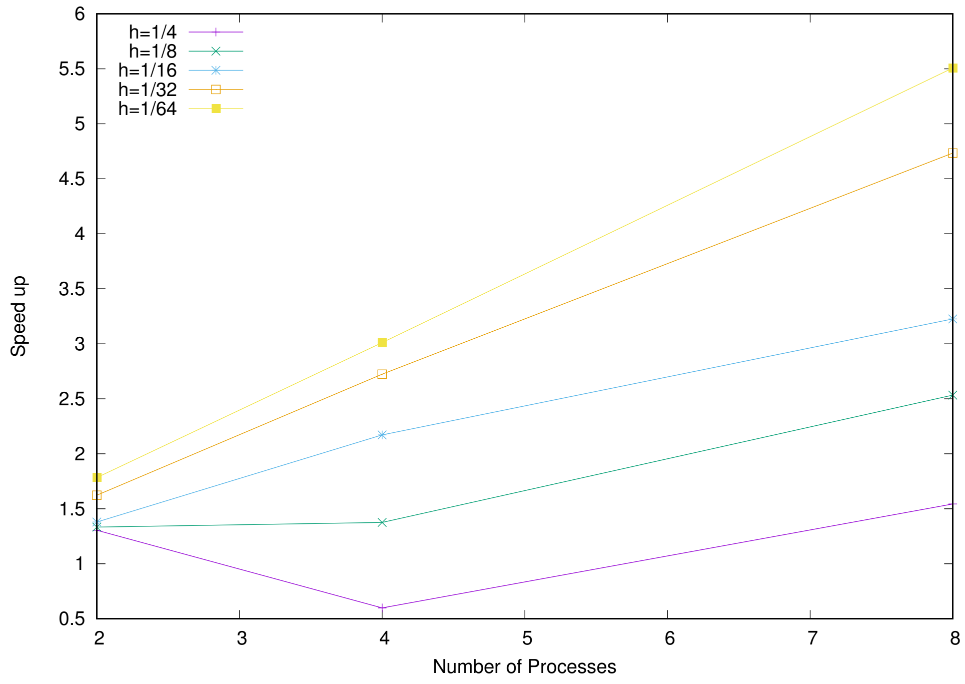


C1 basis functions.

Figure 4.29: CG solver with ASM+MG **SC** speed up with increase in number of threads. Logarithmic scale on Degree of Freedom axis. **MPI** execution. Problem **A3**.

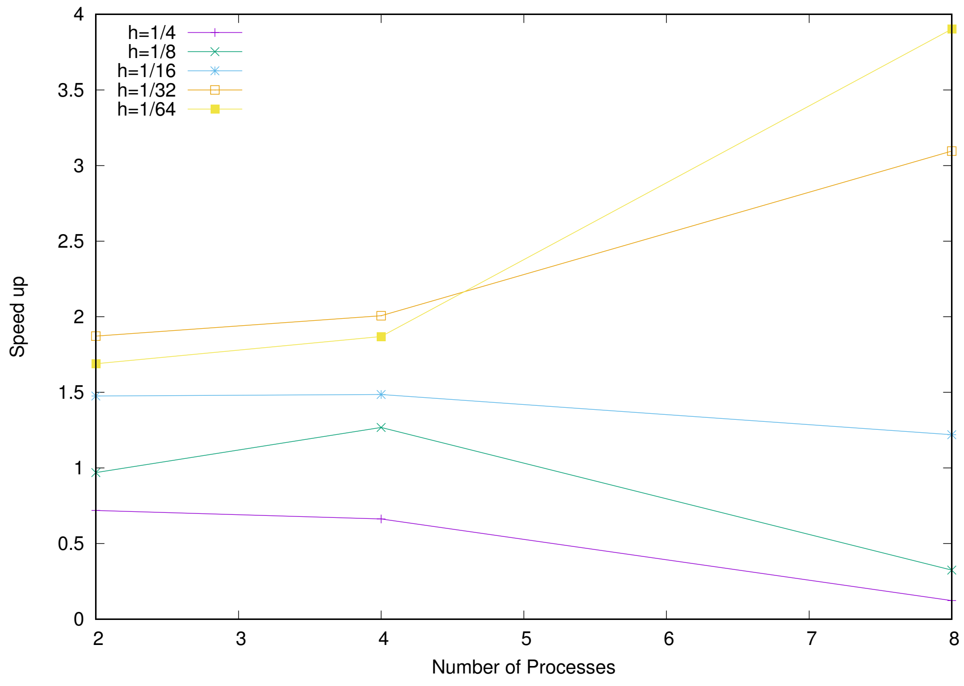


ADLP basis functions.

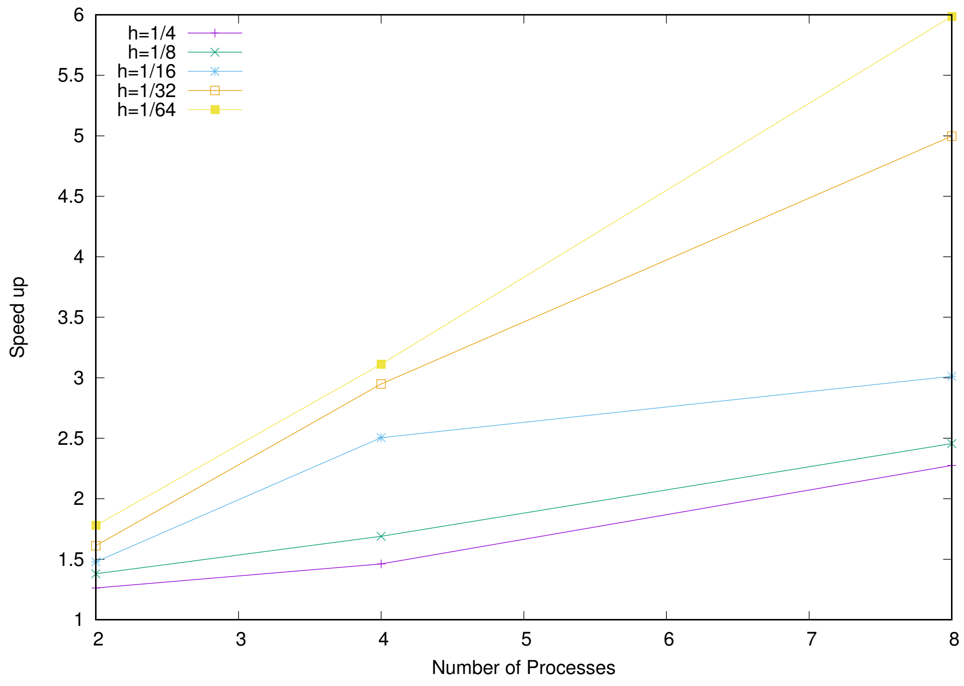


C1 basis functions.

Figure 4.30: CG solver **SA** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 3$. **MPI** execution.

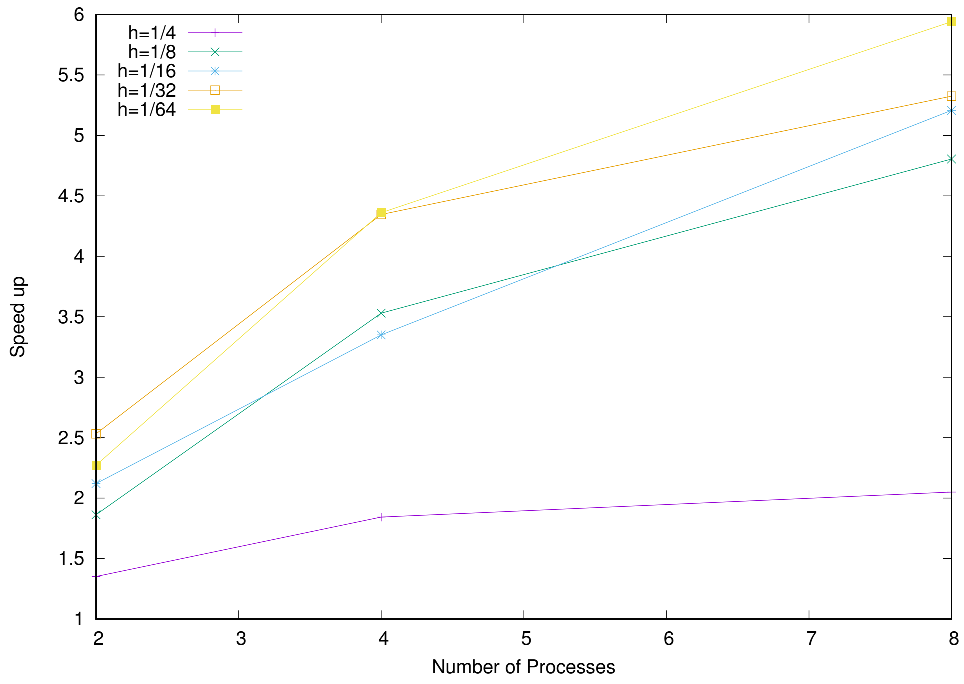


ADLP basis functions.

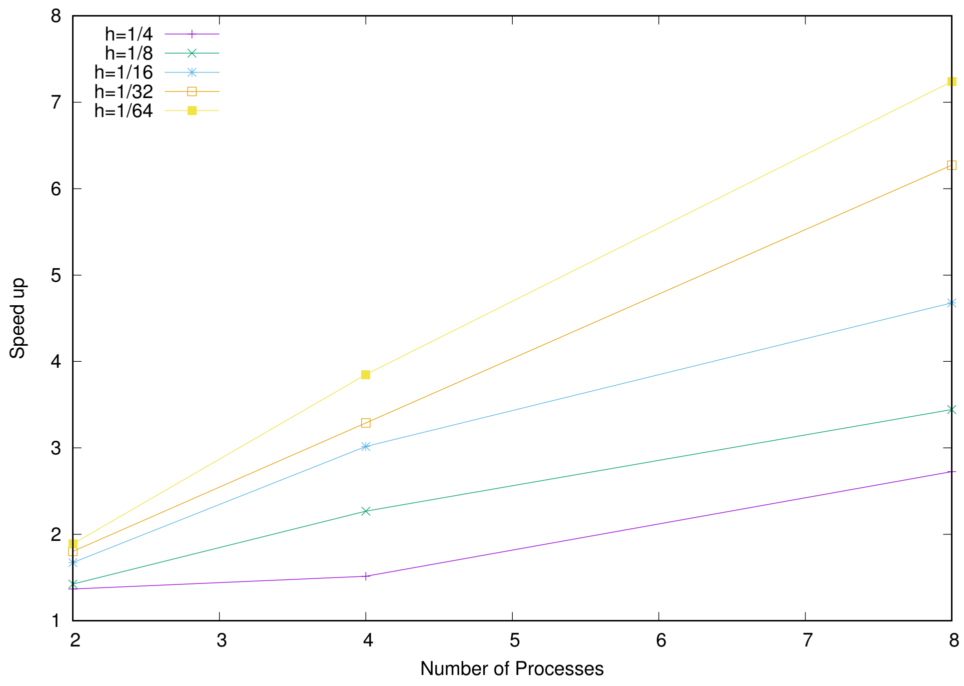


C1 basis functions.

Figure 4.31: CG solver **SA** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 4$. **MPI** execution.

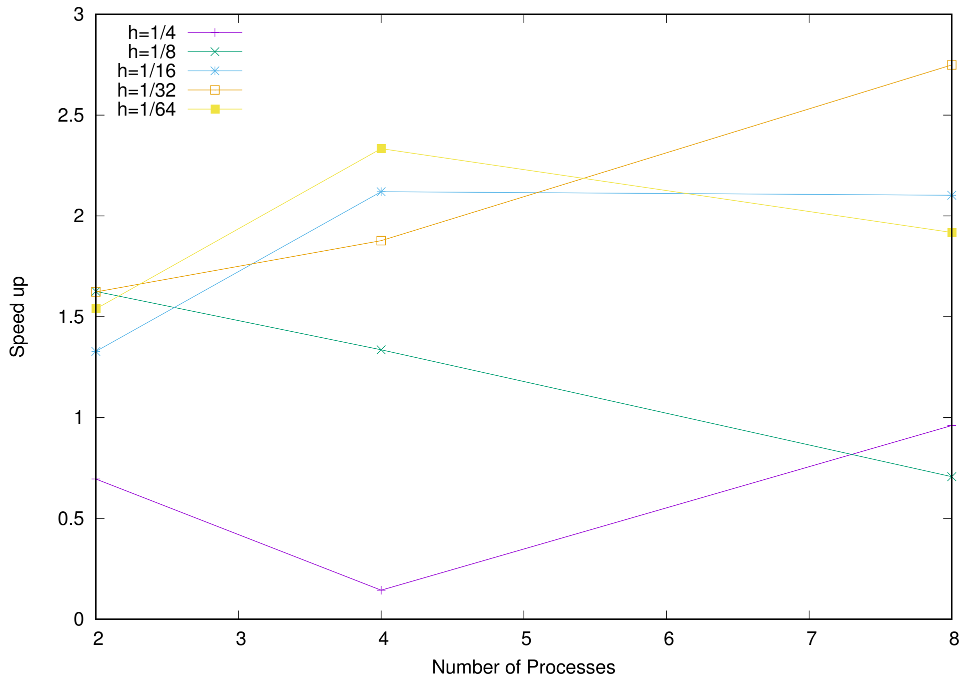


ADLP basis functions.

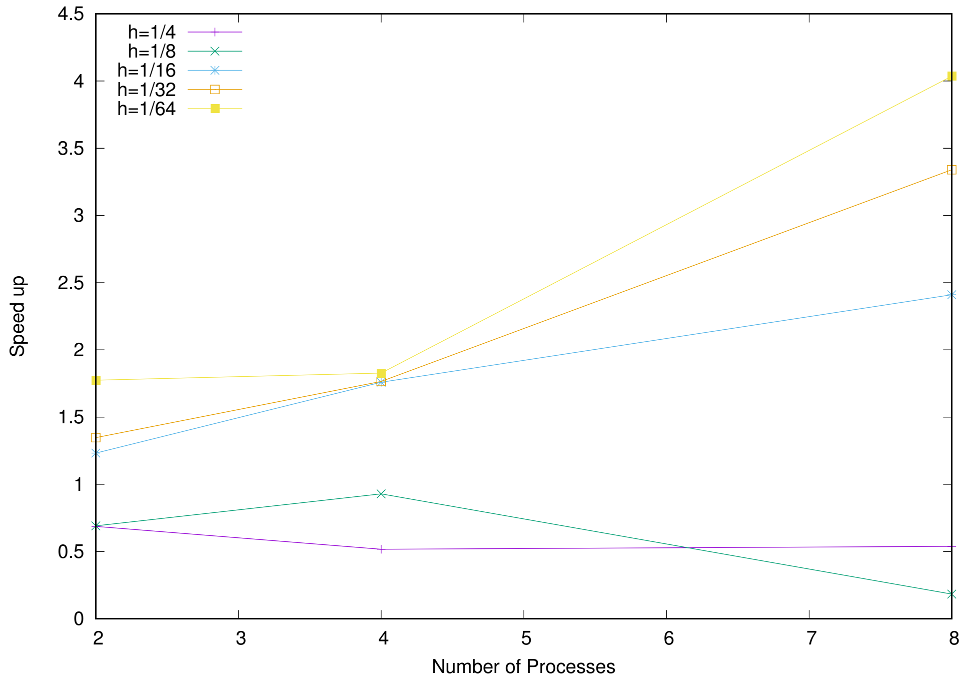


C1 basis functions.

Figure 4.32: CG solver **SA** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 10$. **MPI** execution.

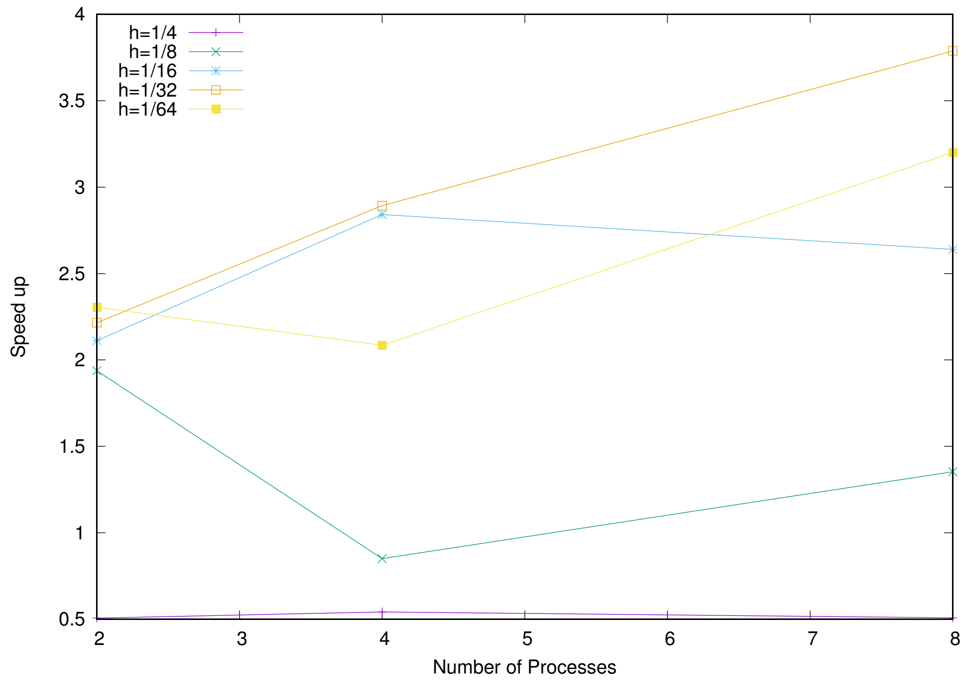


ADLP basis functions.

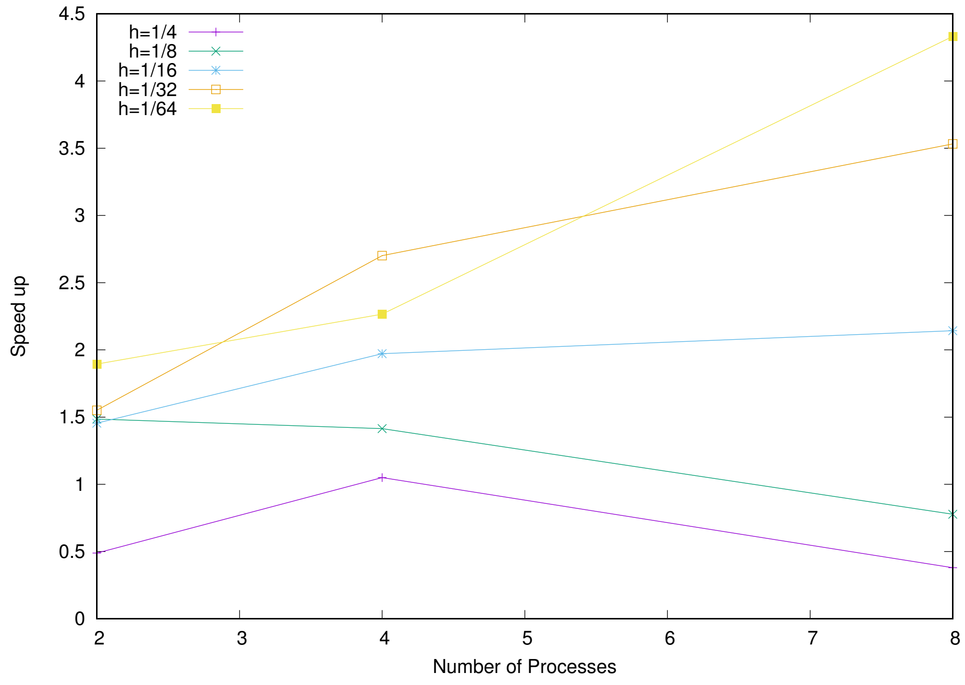


C1 basis functions.

Figure 4.33: CG solver **SB** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 3$. **MPI** execution.

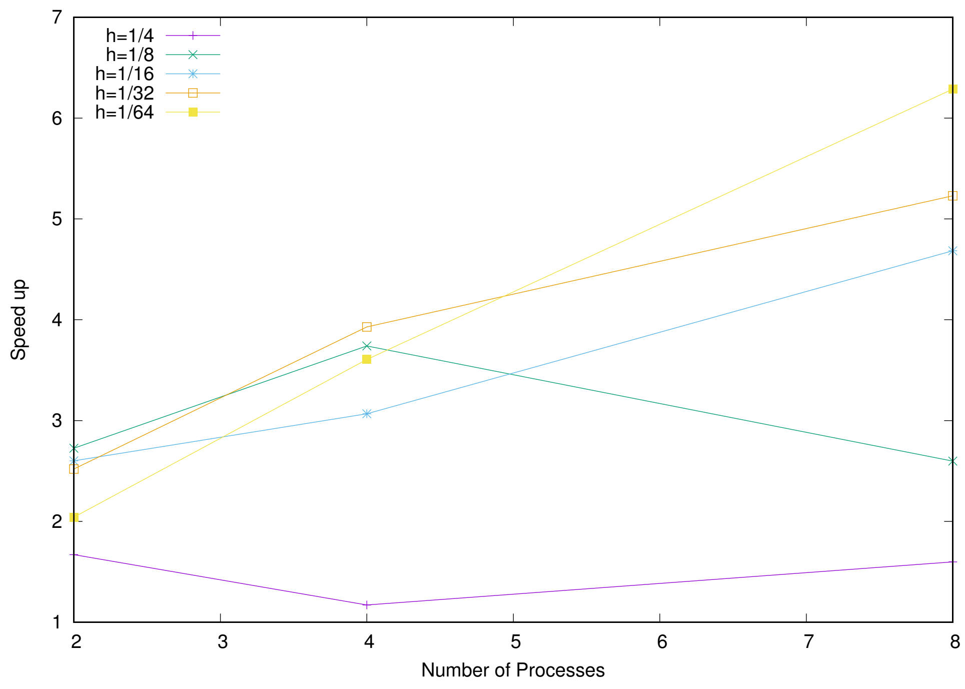


ADLP basis functions.

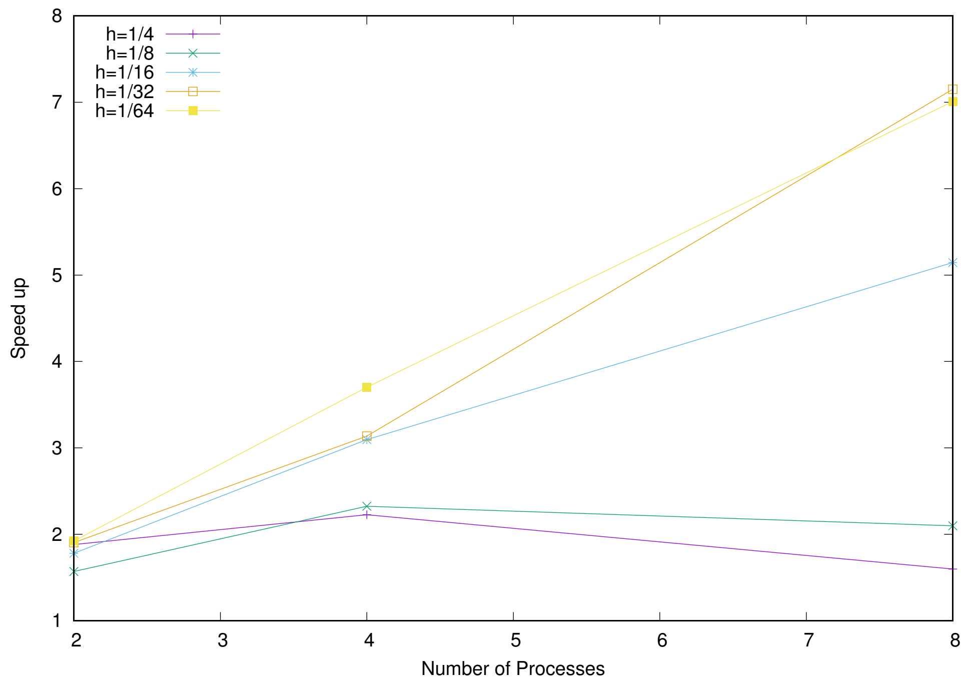


C1 basis functions.

Figure 4.34: CG solver **SB** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 4$. **MPI** execution.

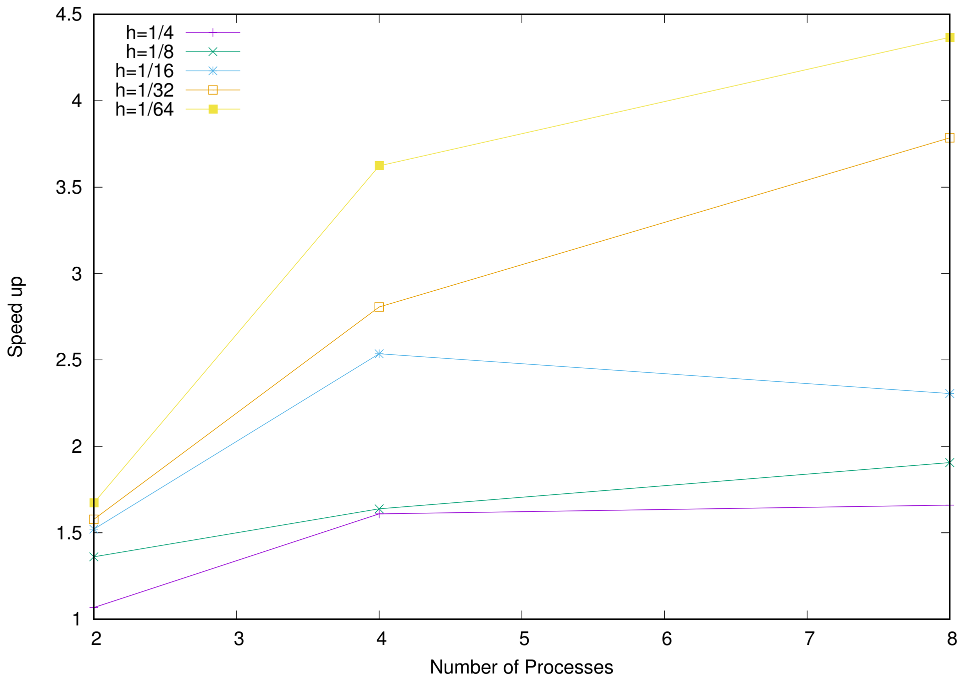


ADLP basis functions.

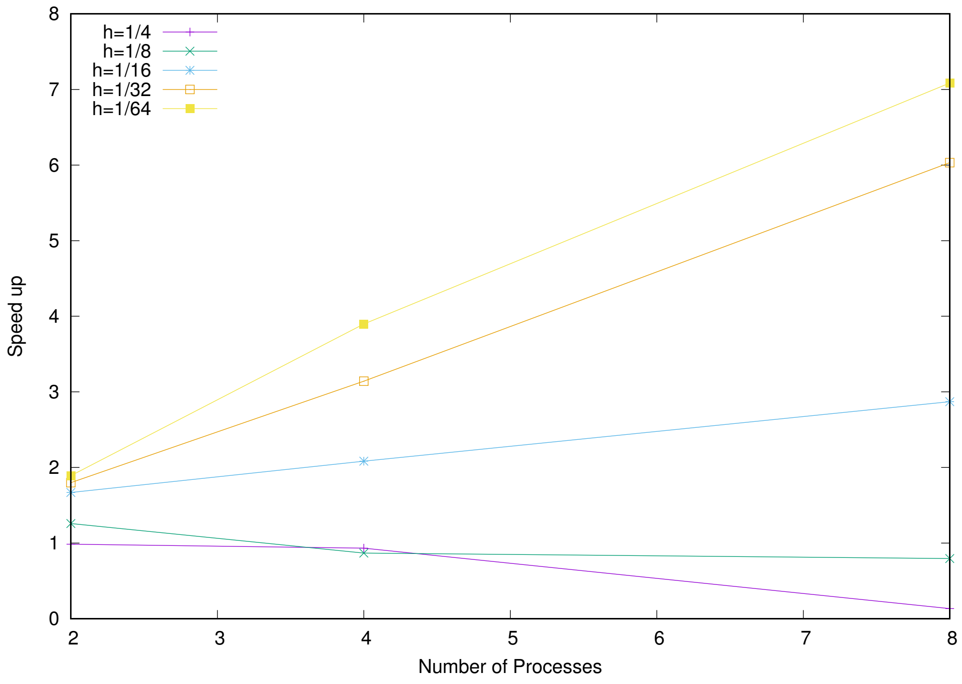


C1 basis functions.

Figure 4.35: CG solver **SB** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 10$. **MPI** execution.

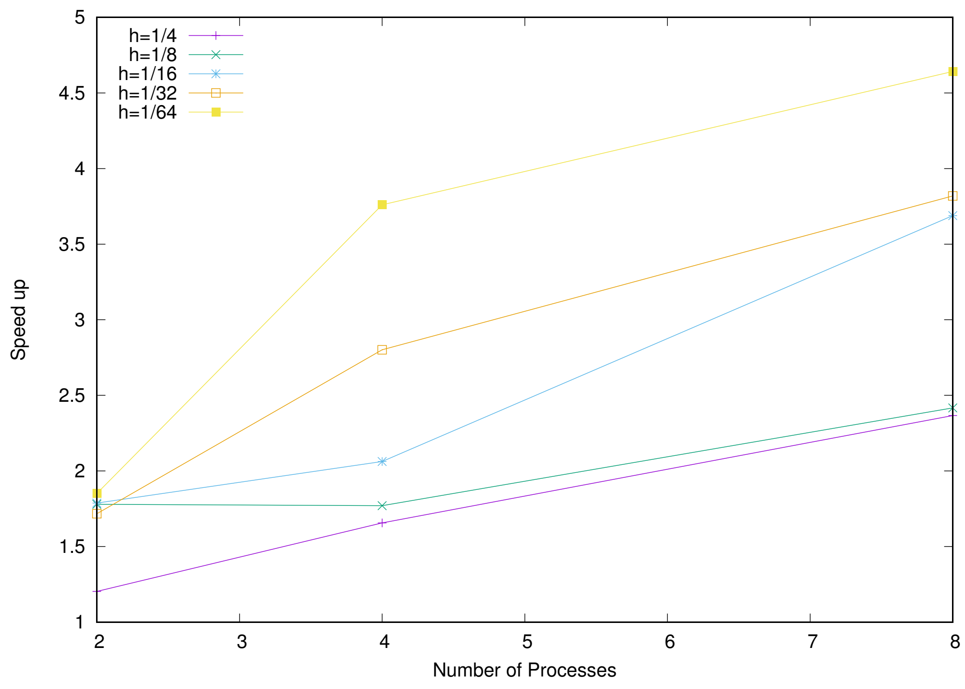


ADLP basis functions.

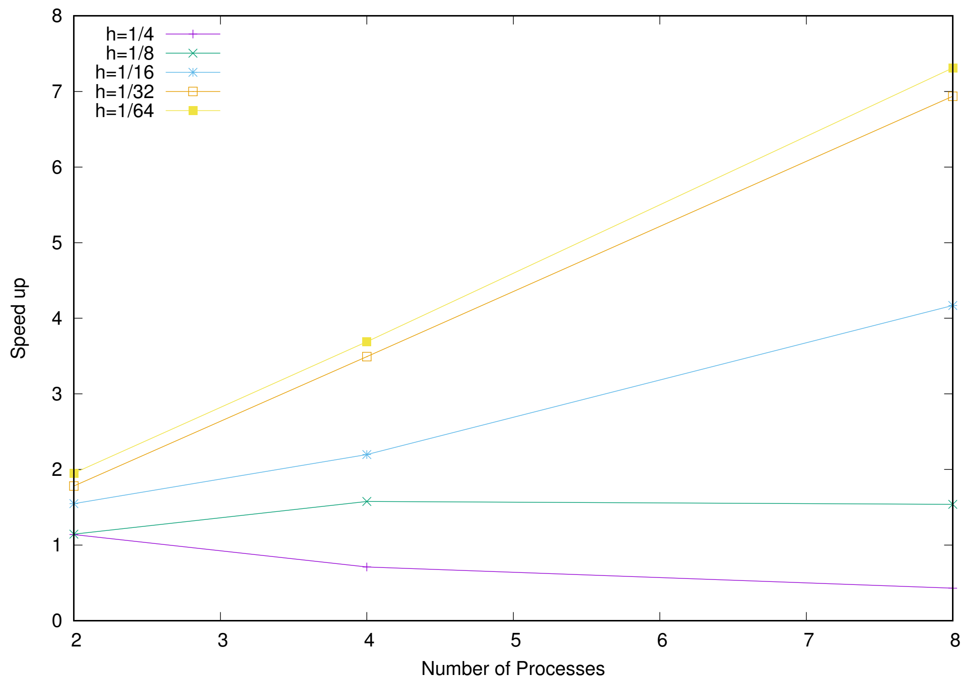


C1 basis functions.

Figure 4.36: CG solver **SC** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 3$. **MPI** execution.

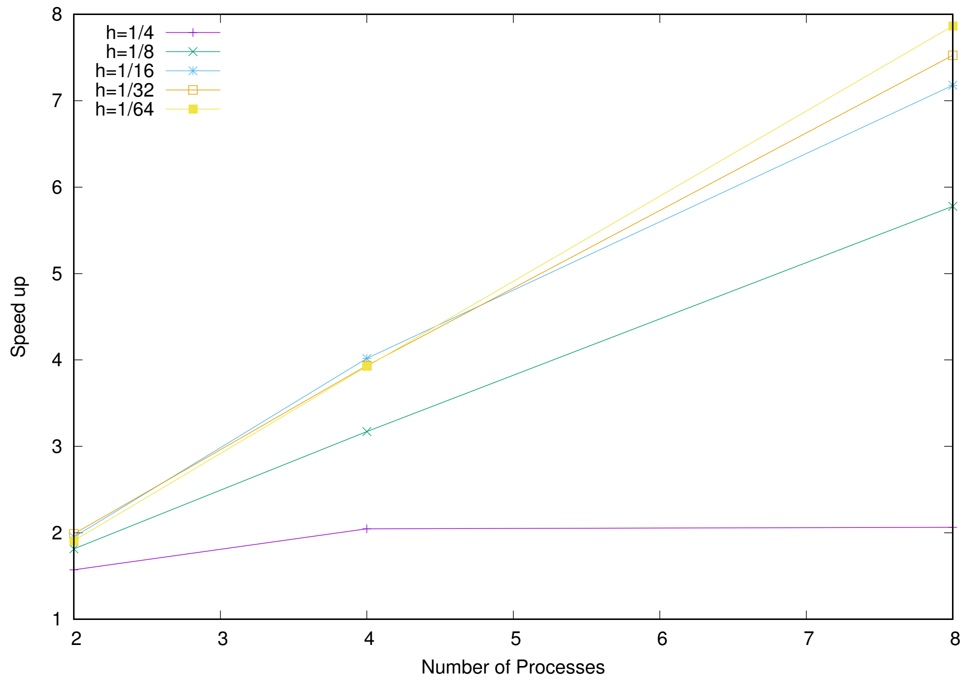


ADLP basis functions.

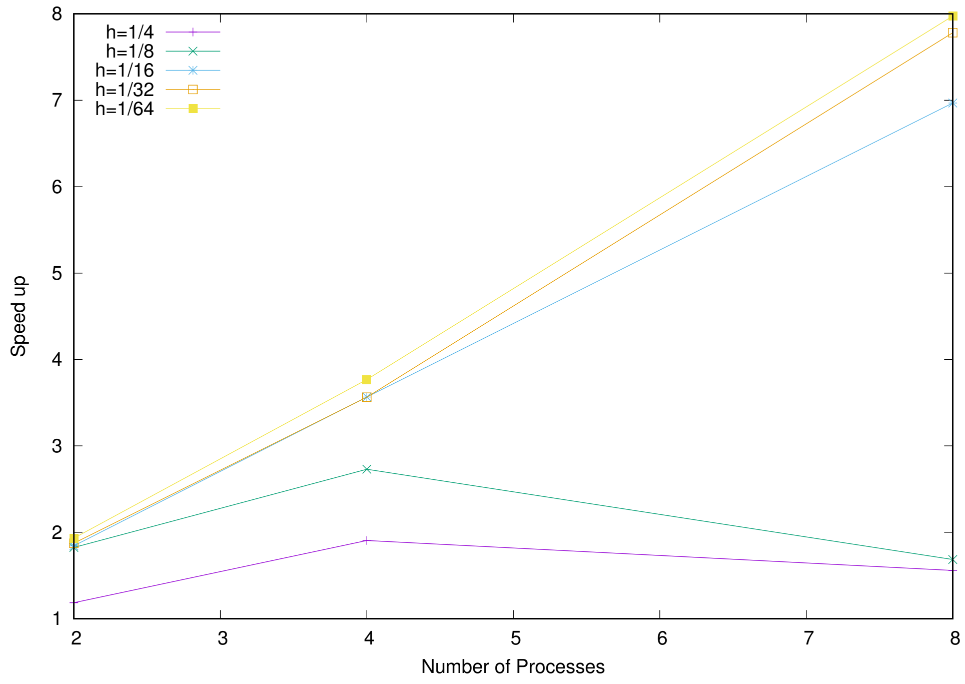


C1 basis functions.

Figure 4.37: CG solver **SC** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 4$. **MPI** execution.



ADLP basis functions.



C1 basis functions.

Figure 4.38: CG solver **SC** speed up vs. Number of available threads. Fixed polynomial degree $N_{pd} = 10$. **MPI** execution.

Chapter 5

Conclusion

5.1 Analyses result

In this research, we have shown, that it is possible to reduce the complexity of the Discontinuous Galerkin method for higher approximation polynomial order with the proper choice of basis function.

In **Section 2.2** we presented the C1-basis function of two kinds, and showed their properties and their derivative's properties, so that later in **Section 2.3** we were able to demonstrate how those properties impact the matrix computation - it's structure, order of sparsity, and total number of non-zero interacting elements. This in turn allowed us to understand the structure of the system matrices, e.g. the structure of the system matrices in Figures 3.1,3.2 and 3.3, which allowed us to develop proper data-storage format χ for matrix-based and for element-neighbours-based parallel distributions. Consequently, we were then able to first compute and then to optimize the parallel matrix-vector multiplication for the choice of the basis function.

Permissibility of the choice of the C1 polynomials as basis functions is demonstrated in **Section 4.5** where we present the error, convergence and condition number analysis. We have observed, that the error estimates are of the same order, which implies equal convergence rates. The difference in the condition number estimates is of negligible order.

Usage of C1 basis functions shows, that the conventional parallel **CG** method can outperform itself for different choices of basis functions. This was noticed by comparing the estimates of computational complexity of parallel version for a system with the Galerkin matrix obtained by use of Antiderivative of Legendre polynomials (3.25), against the same estimates for the C1-polynomials (3.27). This is also what we have observed from numerical experimentation in **Section 4.6** where we performed a specific comparison between the time required for **(i) ADLP** and **(ii) C1** basis functions based Galerkin systems to be solved by the non-preconditioned parallel and non-parallel **CG** to solve the tasks for a given number of degrees of freedom and parallel processes.

Additional gain in efficiency from the C1-polynomials can be seen for **MPI** implementation, due to the fact, that for the sparser matrices, and proper distribution, there is less data to transfer amongst the processes. This was studied in **subsection 3.1.2** with theoretical estimates presented in Table 3.1. The effect of the C1-basis functions can be seen clearly by comparing speed up for $3d$ problem on figures 4.30,4.31,4.32 between the results for **ADLP** and **C1**, where the latter mostly reached the exact parallelization.

At the same time we also demonstrated the possibility to accelerate the convergence for the Conjugate Gradient method itself by applying a preconditioner. In this work we concentrated on the Additive two-level non-overlapping Schwarz method first presented in [4]. We observed that due to the properties of the chosen **C1** basis functions, it is possible to construct the so-called "embarrassingly parallel" (with parallel fraction larger than 0.9) implementation for the fine level decomposition with exact local solvers. This can be seen on figures 4.17, which demonstrate that fine grid problem reaches the Amdahl's limit with the rise in degree of freedom of the problem.

From the analysis of the **ASM** computational complexity (3.53) we have learned, that the coarse level component will be increasing it's impact on the overall preconditioner time with increase in the degree of freedom. Due to the partial parallelization of the coarse grid solver, we observe not much improvement there. To avoid the impact of the serial component implementation we showed that we can restrict ourselves with a low approximation polynomial degree, namely $q = 1$, without losing the convergence. This choice of approximation polynomial degree allows us to deal with much smaller problems on the coarse level in terms of degrees of freedom.

Although the method greatly reduces the time of the computation, we observe that for lower approximation polynomial degrees $N_{pd} = 3, 4$, the speed up does not scale properly with increase in the number of processes. We explain it with cache effect - when the problem blocks are small enough their size can coincide with the CPU's cache memory size, which allows the problem to be solved faster, than for the parallel execution which is heavily affected by data transfer, and required internal initialization of the memory shared protocol on network level, which has lower priority. However, this effect is impacting the system with **C1** basis functions less, due to the fact that data transfer requires less time for a sparser system.

Another method we have analysed, in order to reduce impact of the serial component of the **ASM** method given by exact local solvers on the coarse grid, is an Additive two-level non-overlapping Schwarz method with Multigrid method as a coarse level local solver.

The method presented in **section 3.3** is derived from the Multigrid method for hp-Discontinuous Galerkin [3], and modified to our need. We have shown in **Section 3.3** that the Multigrid method can be used as a suitable solver. Choosing a low approximation polynomial degree $q = 1$ for the Multigrid method as an approximate local coarse level solver renders coarse grid problem to require less operations to solve. Choosing the number of levels in a way such that the lowest level is solved on the smallest possible mesh, allows to reduce the computational complexity down to the order of the mesh size as shown in **Subsection 3.4.3**. The main outcome of the analyses of the Multigrid method for the hp-discontinuous Galerkin is that it is much cheaper than the exact inverse. This can be seen on figures 4.10 and 4.11

In addition, to make the computation even cheaper, we have used the primal bilinear form on the family of spaces inherited from the finite-dimensional space which is defined for the coarse grid discretization (3.79) to compute the local matrices, increasing the set up time, but allowing the method to be spectrally equivalent to the exact inverse.

We have derived the convergence for the modified Additive two-level non-overlapping Schwarz Method with Multigrid method as a coarse level solver in **Subsection 3.4.3**, and found spectral equivalence with the Additive two-level non-overlapping Schwarz method. We observe this in numerical results in Figures 4.5,4.7, which also allow us to state that both methods have equal quasi-uniform growth rate in approximation polynomial degree. This is also true for the iteration number, as both methods have similar iteration bounds, presented in Theorems 3.4.10 and (3.110), and observed in Figures 4.6 and 4.6.

For the analysed methods, we have observed an expected reduction in number of iterations, and expected rate of growth of the condition number. We showed that some of the solver routines can be fully parallelized with the suitable data structures.

All of the aforementioned allows us to conclude that the *hp*-discontinuous Multigrid method with approximation polynomial degree one, allows us to use it as an efficient coarse level solver of the Additive two-level non-overlapping Schwarz method, especially for problems with high degree of freedom by approximation polynomial degree or bigger mesh. We also conclude that the developed data structure χ is effective for the parallel implementation.

5.2 Further research

We would suggest considering more research for the Multigrid method. As one of the interesting scopes for the research from the position of this work is a development of an adaptive scheme for the developed modification of the Additive two-level non-overlapping Schwarz method with Multigrid method as an approximate local solver on a coarse level. This might allow some additional optimization of the method as a coarse level solver, in terms of the execution time and possible parallelization, which can be seen as another possible research topic.

Parallelization of Multigrid method for *hp*-discontinuous Galerkin can also be performed using similar data structure χ, due to the fact, that the method itself, from an algebraic point of view has a matrix inverse and matrix-vector multiplication as the most expensive operations. Analysis of the basic structure of those matrices might provide a possible solution for the parallelization.

Additionally, interesting topic for the possible analysis is the behaviour of the Multigrid method for different parameters such as V-cycle, pre- and post-smoothers number etc. It might be considered as a pareconditioner for the Generalized Minimal Residual and Biconjugate Gradient iterative methods for non-symmetric Discontinuous Galerkin Finite elements methods, and might improve numerical stability of the aforementioned.

The same type of research for the non-symmetric dG FEMs can be suggested for the Schwarz Method as a preconditioner for the mentioned iterative methods. The research might also include an analysis of the effect, which suggested **C1** basis functions might have onto the performance and stability of the methods.

Last, we should suggest a thorough analysis for the exact coarse level solver which might allow parallel versions for bigger problems.

Bibliography

[1] Amdahl, Gene M. 1967. "Validity of the single processor approach to achieving large scale computing capabilities." *AFIPS Conference Proceedings* 30:483—485.
URL: *https://ieeexplore.ieee.org/document/4785615/*

[2] Antonietti, Paola F. and Blanca Ayuso. 2007. "Schwarz domain decomposition preconditioners for discontinuous Galerkin approximations of elliptic problems: nonoverlapping case." *ESAIM: Mathematical Modelling and Numerical Analysis* 41(1):21––54.

[3] Antonietti, Paola F., Marco Sarti and Marco Verani. 2015. "Multigrid Algorithms for *hp*-Discontinuous Galerkin Discretizations of Elliptic Problems." *Journal of Numerical Analysis* 53(1):598–618.

[4] Antonietti, Paola F and Paul Houston. 2011. "A class of domain decomposition preconditioners for hp-discontinuous Galerkin finite element methods." *Journal of Scientific Computing* 46(1):124–149.

[5] Antonietti, Paola F., Paul Houston and Iain Smears. 2015. "A note on optimal spectral bounds for nonoverlapping domain decomposition preconditioners for hp-version discontinuous Galerkin methods." *International Journal of Numerical Analysis and Modeling* .

[6] Arnold, Douglas N. 1982. "An interior penalty finite element method with discontinuous elements." *SIAM journal on numerical analysis* 19:742–760.

[7] Arnold, Douglas N, Franco Brezzi, Bernardo Cockburn and L Donatella Marini. 2002. "Unified analysis of discontinuous Galerkin methods for elliptic problems." *SIAM Journal on Numerical Analysis* 39(5):1749–1779.

[8] Baggag, Abdalkader, Harold Atkins and David Keyes. 1999. "Parallel implementation of the discontinuous Galerkin method." *Nasa Technical Reports Server* .
URL: *https://ntrs.nasa.gov/search.jsp?R=19990100667*

[9] Bassi, Francesco and Stefano Rebay. 1997. "A high-order accurate discontinuous finite element method for the numerical solution of the compressible Navier–Stokes equations." *Journal of Computational Physics* 131(2):267–279.

[10] Baumann, Carlos Erik and J Tinsley Oden. 1999. "A discontinuous hp-finite element method for convection—diffusion problems." *Computer Methods in Applied Mechanics and Engineering* 175(3):311–341.

[11] Bernacki, Marc, Loula Fezoui, Stéphane Lanteri and Serge Piperno. 2006. "Parallel discontinuous Galerkin unstructured mesh solvers for the calculation of three-dimensional wave propagation problems." *Applied Mathematical Modelling* 30(8):744 – 763. Parallel and distributed computing for computational mechanics.
URL: *http://www.sciencedirect.com/science/article/pii/S0307904X0500212X*

[12] Bokhove, Onno and Jaap J.W. van der Vegt. 2008. Introduction to (dis)continuous Galerkin finite element methods. Technical report Department of Applied Mathematics, University of Twente.

[13] Bramble, James H. 1993. *Multigrid Methods*. Chapman & Hall/CRC Research Notes in Mathematics Series 294 0 ed. Chapman and Hall/CRC.

[14] Bramble, James H., Joseph E. Pasciak and Jinchao Xu. 1991. "The Analysis of Multigrid Algorithms with Nonnested Spaces or Noninherited Quadratic Forms." *Mathematics of Computation* .

[15] Burkardt, John. 23 December 2010. "FEM BASIS FUNCTIONS FOR A TRIANGLE." *Lab papers* .
URL: *http://people.sc.fsu.edu/ jburkardt/presentations/*

[16] Bustinza, Rommel, Gabriel N Gatica and Bernardo Cockburn. 2005. "An a posteriori error estimate for the local discontinuous Galerkin method applied to linear and nonlinear diffusion problems." *Journal of Scientific Computing* 22(1-3):147–185.

[17] Castillo, P, B Cockburn, I Perugia and D Schötzau. 2002. "Local discontinuous Galerkin methods for elliptic problems." *Communications in Numerical Methods in Engineering* 18(1):69–75.

[18] Castillo, Paul, Bernardo Cockburn, Ilaria Perugia and Dominik Schötzau. 2000. "An a priori error analysis of the local discontinuous Galerkin method for elliptic problems." *SIAM Journal on Numerical Analysis* 38(5):1676–1706.

[19] Cheng, Y. and Ch.-W. Shu. 2008. "A discontinuous Galerkin finite element method for time dependent partial differential equations with higher order derivatives." *Mathematics of Computation* 77(262):699–730.

[20] Ciarlet, P.G. 1978. "The finite element method for elliptic problems." *Studies in Mathematics and its Applications* 4.

[21] Cockburn, Bernardo and Chi-Wang Shu. 1998. "The local discontinuous Galerkin method for time-dependent convection-diffusion systems." *SIAM Journal on Numerical Analysis* 35(6):2440–2463.

[22] Cockburn, Bernardo and Chi-Wang Shu. 2001. "Runge–Kutta discontinuous Galerkin methods for convection-dominated problems." *Journal of Scientific Computing* 16(3):173–261.

[23] Cockburn, Bernardo, Guido Kanschat, Ilaria Perugia and Dominik Schötzau. 2001. "Superconvergence of the local discontinuous Galerkin method for elliptic problems on Cartesian grids." *SIAM Journal on Numerical Analysis* 39(1):264–285.

[24] Flaherty, Joseph E. N.d. "Finite Element Analysis, Course Notes." *RPI, page 9.*. Forthcoming.
URL: *www.cs.rpi.edu/ flaherje/pdf/fea2.pdf*

[25] Geveler, Markus, Dirk Ribbrock, Dominik Göddeke, Peter Zajac and Stefan Turek. 2013. "Towards a complete FEM-based simulation toolkit on GPUs: Unstructured grid finite element geometric multigrid solvers with strong smoothers based on sparse approximate inverses." *Computers & Fluids* 80:327–332.

[26] Göddeke, Dominik, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, Sven HM Buijssen, Matthias Grajewski and Stefan Turek. 2007. "Exploring weak scalability for FEM calculations on a GPU-enhanced cluster." *Parallel Computing* 33(10):685–699.

[27] Heuer, Norbert. 2001. "Additive Schwarz method for the p-version of the boundary element method for the single layer potential operator on a plane screen." *Numerische Mathematik* 88(3):485–511.

[28] Heuer, Norbert, Ernst Stephan and Thanh Tran. 1998. "Multilevel additive Schwarz method for the hp version of the Galerkin boundary element method." *Mathematics of Computation of the American Mathematical Society* 67(222):501–518.

[29] Houston, Paul, Christoph Schwab and Endre Süli. 2002. "Discontinuous hp-Finite Element Methods for Advection-Diffusion-Reaction Problems." *SIAM Journal on Numerical Analysis* 39(6):2133–2163.

[30] Hussain, Farzana, MS Karim and Razwan Ahamad. 2012. "Appropriate Gaussian quadrature formulae for triangles." *International Journal of Applied Mathematics and Computation* 4(1):023–038.
URL: *www.darbose.in/ijamc*

[31] Jiang, Guang Shan and Chi-Wang Shu. 1994. "On a cell entropy inequality for discontinuous Galerkin methods." *Mathematics of Computation* 62(206):531–538.

[32] Johnson, Claes. 2012. *Numerical solution of partial differential equations by the finite element method*. Courier Dover Publications.

[33] Klockner, A., T. Warburton, J. Bridge and J.S. Hesthaven. 2009. "Nodal discontinuous Galerkin methods on graphic processors." *Journal of Computational Physics* (228):7863–7882.

[34] Krakiwsky, Sean E, Laurence E Turner and Michal M Okoniewski. 2004. *Acceleration of finite-difference time-domain (FDTD) using graphics processor units (GPU)*. Vol. 2.

[35] Loisel S, Szyld D. 2013. "On the convergence of Optimized Schwarz Methods by way of Matrix Analysis.".

[36] Luo, Hong, Luqing Luo, Amjad Ali, Robert Nourgaliev and Chunpei Cai. 2011. "A parallel, reconstructed discontinuous Galerkin method for the compressible flows on arbitrary grids." *Communications in Computational Physics* 9(2):363–389.

[37] Maischak, Dr. Matthias. 20 November 2012*b*. "A priori error estimate and convergence, MA5501 Lecture notes." *MA5501 Lecture notes* .
URL: *https://goo.gl/wL8zh9*

[38] Maischak, Dr. Matthias. 22 October 2012*a*. "Numerical Quadrature." *MA5501 Lecture Notes* .
URL: *https://goo.gl/wL8zh9*

[39] Maischak, Matthias. 20 August 2013. "Lowest-order Raviart-Thomas elements." *MA5501 Lecture Notes* .
URL: *https://goo.gl/wL8zh9*

[40] Maischak, Matthias. 2016. "Technical Manual of the program system maiprogs.".
URL: *http://people.brunel.ac.uk/ mastmmm/tman.pdf*

[41] Maischak, Matthias. 26 August 2014. *Book of Numerical Experiments*. Institut fur Angewandte Mathematik, Universitat Hannover, SISCM, Brunel University, UK.

[42] Peraire, Jaime and P-O Persson. 2008. "The compact discontinuous Galerkin (CDG) method for elliptic problems." *SIAM Journal on Scientific Computing* 30(4):1806–1824.

[43] Persson, Per-Olof. 2009. Scalable parallel Newton-Krylov solvers for discontinuous Galerkin discretizations. In *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition*. p. 606.

[44] Quarteroni, Alfio and Alberto Valli. 1994. *Numerical approximation of partial differential equations*. Springer Series in Computational Mathematics 23 1 ed. Springer-Verlag Berlin Heidelberg.

[45] Raviart, P.-A and J.-M. Thomas. 1977. "A mixed finite element method for second order elliptic problems." *Mathematical aspects of Finite Elements Methods* .

[46] Saad, Yousef. 2003. *Iterative methods for sparse linear system. Second edition*. Society for Industrial and Applied Mathematics.

[47] Samii, Ali, Craig Michoski and Clint Dawson. 2016. "A parallel and adaptive hybridized discontinuous Galerkin method for anisotropic nonhomogeneous diffusion." *Computer Methods in Applied Mechanics and Engineering* 304:118–139.

[48] Schötzau, Dominik and Liang Zhu. 2009. "A robust a-posteriori error estimator for discontinuous Galerkin methods for convection–diffusion equations." *Applied Numerical Mathematics* 59(9):2236–2255.

[49] Shaidurov, V.V. 1989. *Multigrid finite elements method*. Nauka [Science].

[50] Shewchuk, Jonathan Richard. August 4, 1994. "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain." *Unknown* .

[51] Smith, Barry F., Petter E. Bjørstad and William D. Gropp. 1996. *Domain Decomposition. Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press.

[52] Toselli, A. and O. Widlund. 2005. *Domain Decomposition Methods. Algorithms and Theory*. Springer-Verlag.

[53] Winter, Matthias. 2013. "MA5503 Finite elements method lecture notes.".
URL: *https://goo.gl/JzAVrY*

[54] Zhang, Mengping and Chi-Wang Shu. 2003. "An analysis of three different formulations of the discontinuous Galerkin method for diffusion equations." *Mathematical Models and Methods in Applied Sciences* 13(03):395–413.

[55] Zhu, Jun, Xinghui Zhong, Chi-Wang Shu and Jianxian Qiu. 2013. "Runge–Kutta discontinuous Galerkin method using a new type of WENO limiters on unstructured meshes." *Journal of Computational Physics* 248:200–220.