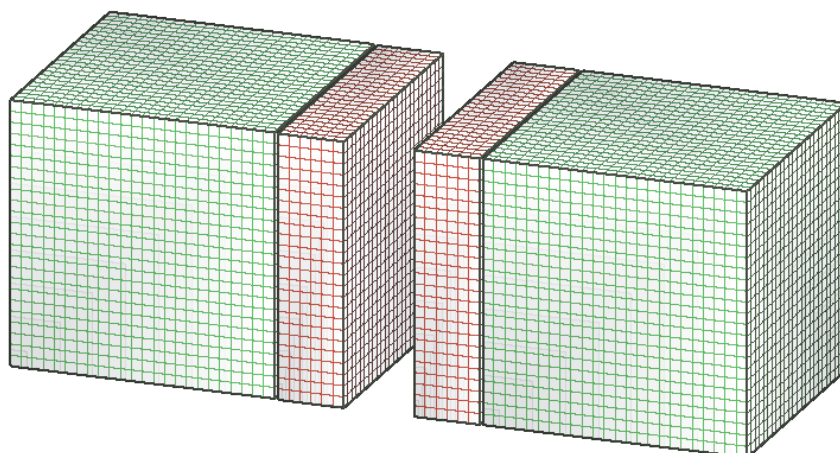
Robert Seidel

Parallelization of a high order 3D compressible Navier-Stokes code

Trondheim, 06 2017

Project work

NTNU
Norwegian University of
Science and Technology
Faculty of Engineering Science
Department of Energy and Process Engineering



NTNU – Trondheim
Norwegian University of
Science and Technology

Norwegian University of Science and Technology

Faculty of Engineering Science and Technology
Department of Energy and Process Engineering

EPT-P-2017-11

NTNU

PROJECT WORK

for

student Robert Seidel

Spring 2017

Parallelization of a high order 3D compressible Navier-Stokes code
Parallellisering av en høyre ordens 3D kompressibel Navier-Stokes kode

Background and objective

High order methods for the 3D compressible Navier-Stokes equations are much more efficient than low order methods, when high accuracy is required like in direct numerical simulation of turbulence. To exploit high performance computing for the fourth order compressible Navier-Stokes code developed in the CFD group of the Department of Energy and Process Engineering at NTNU, parallelization is indispensable.

The objective of the project is to parallelize that high order multi-block code. Since the code uses MPI for the communication between the blocks, domain decomposition can be used to parallelize it. The efficiency of the parallelization is to be tested on a high performance cluster of the Department of Energy and Process Engineering. The project is linked to the larger interdisciplinary research project *Modeling of obstructive sleep apnea by fluid-structure interaction in the upper airways*, which is a collaboration between NTNU, SINTEF and St. Olavs Hospital and funded by the Research Council of Norway.

The following tasks are to be considered:

1. to get a basic understanding of the 3D compressible Navier-Stokes equations and their numerical solution by a high order difference method.
2. to review the literature on efficient parallelization of high order explicit methods on structured grids.
3. to parallelize the high order multi-block code.
4. to check the efficiency of the parallelization on a high performance computer.

-- " --

Address
N-7034 Trondheim
Norway

Location
K. Hejes vei 1b

Tel. +47 73 59 38 60
Fax +47 73 59 35 80
Org. no. NO 974 767 880

The project work comprises 15 ECTS credits.

The work shall be edited as a scientific report, including a table of contents, a summary in Norwegian, conclusion, an index of literature etc. When writing the report, the candidate must emphasise a clearly arranged and well-written text. To facilitate the reading of the report, it is important that references for corresponding text, tables and figures are clearly stated both places.
By the evaluation of the work the following will be greatly emphasised: The results should be thoroughly treated, presented in clearly arranged tables and/or graphics and discussed in detail.

The candidate is responsible for keeping contact with the subject teacher and teaching supervisors.

Risk assessment of the candidate's work shall be carried out according to the department's procedures. The risk assessment must be documented and included as part of the final report. Events related to the candidate's work adversely affecting the health, safety or security, must be documented and included as part of the final report. If the documentation on risk assessment represents a large number of pages, the full version is to be submitted electronically to the supervisor and an excerpt is included in the report.

According to "Utfyllende regler til studieforskriften for teknologistudiet/sivilingeniørstudiet ved NTNU" § 20, the Department of Energy and Process Engineering reserves all rights to use the results and data for lectures, research and future publications.

The report shall be submitted to the department via Its Learning.

Submission deadline: *June 9, 2017.*

☐ Work to be done in lab (Water power lab, Fluids engineering lab, Thermal engineering lab)
☐ Field work

Department for Energy and Process Engineering, *January 15, 2017.*

Bernhard Müller
Supervisor

Abstract

In this project report, we present a parallel numerical solver for the three dimensional compressible Navier-Stokes equations in perturbation form by the finite difference method. We employ fourth order accurate summation by parts (SBP) difference operators and the explicit fourth order accurate classical Runge-Kutta method. The parallelization is achieved using domain decomposition and the Message Passing Interface (MPI). Different implementations of the SBP operators and their cache profile are studied. We find that cache optimizations have only small run time effects. Surprisingly, also non-blocking communication did not outperform the reference implementation. The HDF5 format is used for disk I/O. In our scaling study, we find very favorable scaling properties of this implementation. In particular, we measure almost perfect weak scaling. However, our results are physically not valid, as we could not fix all formula errors by the deadline of this report.

Sammendrag

I denne prosjektrapporten presenterer vi en parallell numerisk løsning av den tredimensjonale kompressible Navier-Stokes-ligningen på perturbert form ved hjelp av differansemetoden. Vi anvender fjerdeordens summation by parts (SBP) differensoperatorerer og en eksplisitt klassisk Runge-Kutta-metode av fjerde orden. Parallelliseringen oppnås ved å bruke domain decomposition og Message Passing Interface (MPI). Vi studerer ulike implementasjoner av SBP-operatoren og deres respektive cache-profiler, og observerer at cache-optimalisering kun har minimal effekt på kjøretid. Overraskende nok er heller ikke non-blocking communication signifikant bedre enn referanseiplementasjonen. HDF5-formatet brukes for disk I/O. Videre observerer vi fordelaktige skaleringseffekter ved implementasjonen. Vi måler tilnærmet perfekt weak scaling. Resultatene er likevel ikke fysisk gyldige, da vi ikke rakk å endre alle formelfeil før denne rapporten skulle leveres.

Contents

1 Introduction **10**

2 The Navier-Stokes equations **12**

2.1 Perturbation formulation 12

2.2 Variable transformation 13

2.3 Discretization 15

2.4 Domain decomposition 15

3 Methods and implementation details **17**

3.1 Decomposed domain indexing 17

3.2 Modelling of walls 17

3.3 Cache-efficient approximation of derivatives 18

3.4 Non-blocking communication 21

3.5 HDF5 parallel I/O 21

4 Results **22**

4.1 Validation 23

4.2 Cache simulation 23

4.3 Cost of communication 24

4.4 Scaling 25

4.4.1 Strong scaling 25

4.4.2 Weak scaling 27

5 Conclusions and outlook **28**

List of Tables

1 Cache profiling of SBP operator: Best results of "new" implementation are highlighted. The abbreviations are explained in section 4.2 23

2 Non-blocking communication: Run time and pagefaults after computing the metric terms with 100^3 points. The abbreviations are explained in section 4.3. 24

3 Strong scaling: Runtime of 500 time steps on 100^3 points with different numbers of nodes and threads. Best and worst performance are highlighted. The abbreviations are explained in section 4.4.1. 25

4 Weak scaling: Runtime of 500 time steps on different grids with different numbers of points per MPI rank. The abbreviations are explained in section 4.4.2. 26

List of Figures

1 Illustration of two neighboring blocks. The overlap of the two blocks is shown with magenta lines. This illustration can be seen as 3D analogy to figure 2 in [6]. 16

2 Strong scaling: Runtime of 500 time steps on 100^3 points with different numbers of nodes and threads. 27

3 Weak scaling: Runtime of 500 time steps on different grids with different numbers of points per MPI rank. 28

Foreword

This project report is the outcome of my participation in "TEP4165 Computational Heat and Fluid Flow" in autumn 2016 and "TEP4540 Engineering Fluid Mechanics, Specialization Project" in spring 2017, two courses I was able to take during my one-year exchange at NTNU in Trondheim, Norway, and it linked to the research project "Modeling of Obstructive Sleep Apnea by Fluid-Structure Interaction in the Upper Airways", which has been funded by the *Research Council of Norway* under project number 231741 since mid 2014. As a mathematician without previous knowledge about fluids, it was adventurous to dive into both an unknown society and a new field of research. My stay in Norway was founded by the *ERASMUS+* program of the European Union and the *German Academic Scholarship Foundation.*

I want to thank my colleagues Petra Tisovská and Mohammadtaghi Khalili who were working with me on the three dimensional formulation and the program. It were Jarmo Rantakokko and Marcus Holm from the University in Uppsala, Sweden who provided me with helpful remarks on drafts of this project report. Anders Hjort helped me with the translation of this report's abstract to Norwegian. Finally, I want to thank my lecturer and supervisor Bernhard Müller for proposing this project to me and for his patient and competent help at all stages of work.

Robert Seidel

Trondheim, Mai 2017

1 Introduction

Numerical simulations of physical problems play an important role in mathematics, engineering and applied sciences. When it comes to the Navier-Stokes equations, scientists encounter a non-linear problem that is particularly demanding in terms of computational power, memory and accuracy. In addition, the three dimensional scenario suffers from a high number of grid elements that significantly slows down direct numerical simulations. The latter problem can be solved by using high performance computers and parallelized implementations of the numerical schemes. This approach has been followed successfully, using different numerical methods such as the finite element method (FEM) [30] or the finite volume method (FVM) [26] for incompressible flow governed by the Navier-Stokes equations. There are several frameworks such as the CFD software *openFOAM* and the toolkit *PETSc* that bring parallelized solvers for differential equations out-of-the-box or as plugins.

Obstructive sleep apnea (OSA) is a medical disorder that is linked to the flow of air through the upper airways of the human body. The OSA syndrome (OSAS) "affects 2-4% of the population and is recognized by heavy snoring, frequent breathing stops, gasping for breath, and awakenings. OSAS is the cause for low quality sleep and reduced oxygen consumption and is considered as a major cause for reduced life quality and increased mortality in the modern society." [17] This project work heavily relies on the work by Khalili et al. [6], where the interaction between a simplified soft palate and compressible viscous flow has been studied in the context of the OSA syndrome. They use a high order finite difference method in order to solve the two dimensional compressible Navier-Stokes equations for the fluid flow and employ Euler-Bernoulli thin beam theory for the structure model. The master's thesis by Tisovská [25] extended their two dimensional code to a three dimensional setup.

Aiming to solve computations in meterology and oceanography, Kreiss and Oliger [9] study the shallow water equations and show the advantages of higher-order difference methods over second- and lower-order methods. Finite difference operators that satisfy the summation by parts (SBP) rule and yield higher accuracy have been developed by Kreiss and Scherer [7, 8] and revisited by Strand [18]. Among others, Svärd and Mishra [20] capture the importance of high-order schemes when dealing with high eigenfrequencies and long time calculations. However, enforcing the boundary conditions in this method directly can lead to growing solutions [4] or other misleading instabilities [20]. By imposing boundary conditions weakly, the Simultaneuous Approximation Term (SAT) technique allowes for stability proofs for more complicated partial differential equations [1]. Experimental results can, again, be found in the work by Svärd and Mishra [20]. Both Gustafsson [4] and Svärd and Nordström [21] study accuracy

conditions of SBP operators and their relaxations at the boundary, and derive convergence rates for the SBP method. Generally, a comprehensive rewiew of SBP schemes has been written by Svärd and Nordström [19], on which this section is mostly based.

Mattsson et al. [10] compute an unsteady airfoil flow governed by the Euler equations and find that, unlike lower-order methods, higher-order methods can accurately reproduce the reference solution. They also outline a domain splitting technique that involves block-to-block communication for parallelization. An almost linear speedup could be observed. Svärd et al. [23] reproduces these finding for airfoil computations governed by the Navier-Stokes equations. Svärd and Nordström [22] further stresses the superior computational efficiency of higher-order methods in terms of wall-clock time, after evaluating the flow around a cylinder governed by the Navier-Stokes equations. This finding is confirmed by Weide et al. [28] for the flow over a smooth bump.

The project at hand delivers a parallelized implementation of three dimensional code developed by Khalili et al. [6]. It abstracts the the block structure presented there towards a more general domain decomposition. The two dimensional code developed in [6] was not rewritten, but instead, we started with a new framework from scratch and then transferred the changes that were simultaneously done for the three dimensional implementation by Tisovská [25] into the new framework. This procedure was functional, but at the same time overly time consuming. We used [2] as the main reference to the FORTRAN language.

The project report is organized as follows. In Section 2, we introduce the governing equations and derive a variable transformation for a generalized physical domain. We then explain the applied discretization and domain decomposition. Section 3 clarifies the methods and improvements that have been implemented for parallelization. In Section 4, we discuss our numerical results and benchmarks. Section 5 concludes.

2 The Navier-Stokes equations

2.1 Perturbation formulation

In this project, we study the 3D compressible Navier-Stokes equations in perturbation form [6, 11]

$$U'_t + F'_x + G'_y + H'_z = 0, \tag{1}$$

where

$$F' = F^{c\prime} - F^{v\prime}, \tag{2}$$

$$G' = G^{c\prime} - G^{v\prime}, \tag{3}$$

$$H' = H^{c\prime} - H^{v\prime}, \tag{4}$$

$$\tag{5}$$

and

$$F^{c\prime} = \begin{pmatrix} (\rho u)' \\ (\rho u)'u' + p' \\ (\rho v)'u' \\ (\rho w)'u' \\ ((\rho H)_0 + (\rho H)')u' \end{pmatrix}, \qquad F^{v\prime} = \begin{pmatrix} 0 \\ \tau'_{xx} \\ \tau'_{xy} \\ \tau'_{xz} \\ u'\tau'_{xx} + v'\tau'_{xy} + w'\tau'_{xz} + \kappa T'_x \end{pmatrix}, \tag{6}$$

$$G^{c\prime} = \begin{pmatrix} (\rho v)' \\ (\rho u)'v' \\ (\rho v)'v' + p' \\ (\rho w)'v' \\ ((\rho H)_0 + (\rho H)')v' \end{pmatrix}, \qquad G^{v\prime} = \begin{pmatrix} 0 \\ \tau'_{yx} \\ \tau'_{yy} \\ \tau'_{yz} \\ u'\tau'_{yx} + v'\tau'_{yy} + w'\tau'_{yz} + \kappa T'_y \end{pmatrix}, \tag{7}$$

$$H^{c\prime} = \begin{pmatrix} (\rho w)' \\ (\rho u)'w' \\ (\rho v)'w' \\ (\rho w)'w' + p' \\ ((\rho H)_0 + (\rho H)')w' \end{pmatrix}, \qquad H^{v\prime} = \begin{pmatrix} 0 \\ \tau'_{zx} \\ \tau'_{zy} \\ \tau'_{zz} \\ u'\tau'_{zx} + v'\tau'_{zy} + w'\tau'_{zz} + \kappa T'_z \end{pmatrix}, \tag{8}$$

$$\tag{9}$$

where t is the physical time, x, y, z are the physical Cartesian coordinates, ρ denotes the density, u, v, w the x-, z- and z- direction velocity components, E the specific total energy, T the temperature and κ

the heat conduction coefficient calculated from a constant Prandtl number $\text{Pr} = \frac{\mu c_p}{\kappa} = 1$. Note that $U = (\rho, \rho u, \rho v, \rho w, \rho E)^T$ is the vector of the conservative variables, while $U' = U - U_0$ is the vector of conservative perturbation variables with $U_0 = (\rho_0, 0, 0, 0, (\rho E)_0)^T$, the stagnation values. Similarly, $(\rho H)_0$ denotes the stagnation enthalpy density. The perturbation variables are given by $\rho' = \rho - \rho_0$, $(\rho \mathbf{u})' = \rho \mathbf{u}$ with $\mathbf{u}' = \frac{(\rho \mathbf{u})'}{\rho_0 + \rho}$, and $(\rho E)' = \rho E - (\rho E)_0$, $(\rho H)' = (\rho E)' + p'$, $\boldsymbol{\tau}' = \mu(\nabla \mathbf{u}' + (\nabla \mathbf{u}')^T) - \frac{2}{3}\mu(\nabla \cdot \mathbf{u}')\mathbf{I}$, and $T' = \frac{p'/R - \rho' T_0}{\rho_0 + \rho'}$, R being the perfect gas constant. $\boldsymbol{\tau}'$ is the viscous stress tensor, $\mathbf{I}$ is the identity matrix, and μ is the viscosity from the Sutherland law

$$\frac{\mu}{\mu_0} = \left(\frac{T}{T_0}\right)^{1.5} \frac{1 + S_c}{T/T_0 + S_c}. \tag{10}$$

where $S_c = \frac{110}{301.75}$. The pressure perturbation can be computed from other perturbation variables for perfect gas by

$$p' = (\gamma - 1)\left[(\rho E)' - \frac{1}{2}((\rho \mathbf{u}') \cdot \mathbf{u}')\right], \tag{11}$$

with the ratio of specific heats $\gamma = \frac{c_p}{c_v} = 1.4$ for air [6]. The perturbation formulation reduces rounding errors in simulations of low Mach number flows [11].

We aim to model a channel with an inlet and an outlet in x-direction, walls with adiabatic no-slip conditions in y-direction and periodic boundary conditions in z-direction. The boundary treatment follows the work of Poinsot and Lele [15].

2.2 Variable transformation

In this section, we examine a variable transform that allows us to transform the Navier-Stokes equations from the physical domain to a computational domain, as for example shown by Pletcher et al. [14]. Note that the transformation is time-independent. Let the physical domain be parametrized by a triplet (x, y, z) and the computational domain be parametrized by a triplet (ξ, η, ζ). Further, we assume that the relations

$$\xi = \xi(x, y, z), \tag{12}$$

$$\eta = \eta(x, y, z), \tag{13}$$

$$\zeta = \zeta(x, y, z) \tag{14}$$

are given. By the chain rule, we obtain the following differential expressions in matrix form:

$$\begin{pmatrix} \mathrm{d}\xi \\ \mathrm{d}\eta \\ \mathrm{d}\zeta \end{pmatrix} = \begin{pmatrix} \xi_x & \xi_y & \xi_z \\ \eta_x & \eta_y & \eta_z \\ \zeta_x & \zeta_y & \zeta_z \end{pmatrix} \begin{pmatrix} \mathrm{d}x \\ \mathrm{d}y \\ \mathrm{d}z \end{pmatrix} \tag{15}$$

and

$$\begin{pmatrix} \mathrm{d}x \\ \mathrm{d}y \\ \mathrm{d}z \end{pmatrix} = \begin{pmatrix} x_\xi & x_\eta & x_\zeta \\ y_\xi & y_\eta & y_\zeta \\ z_\xi & z_\eta & z_\zeta \end{pmatrix} \begin{pmatrix} \mathrm{d}\xi \\ \mathrm{d}\eta \\ \mathrm{d}\zeta \end{pmatrix}. \tag{16}$$

In particular, we find

$$\frac{\partial}{\partial x} = \xi_x \frac{\partial}{\partial \xi} + \eta_x \frac{\partial}{\partial \eta} + \zeta_x \frac{\partial}{\partial \zeta}, \tag{17}$$

$$\frac{\partial}{\partial y} = \xi_y \frac{\partial}{\partial \xi} + \eta_y \frac{\partial}{\partial \eta} + \zeta_y \frac{\partial}{\partial \zeta}, \tag{18}$$

$$\frac{\partial}{\partial z} = \xi_z \frac{\partial}{\partial \xi} + \eta_z \frac{\partial}{\partial \eta} + \zeta_z \frac{\partial}{\partial \zeta}. \tag{19}$$

By combining (15) and (16), we arrive at

$$\begin{pmatrix} \xi_x & \xi_y & \xi_z \\ \eta_x & \eta_y & \eta_z \\ \zeta_x & \zeta_y & \zeta_z \end{pmatrix} = \begin{pmatrix} x_\xi & x_\eta & x_\zeta \\ y_\xi & y_\eta & y_\zeta \\ z_\xi & z_\eta & z_\zeta \end{pmatrix}^{-1} = J \begin{pmatrix} y_\eta z_\zeta - y_\zeta z_\eta & -(x_\eta z_\zeta - x_\zeta z_\eta) & x_\eta y_\zeta - x_\zeta y_\eta \\ -(y_\xi z_\zeta - y_\zeta z_\xi) & x_\xi z_\zeta - x_\zeta z_\xi & -(x_\xi y_\zeta - x_\zeta y_\xi) \\ y_\xi z_\eta - y_\eta z_\xi & -(x_\xi z_\eta - x_\eta z_\xi) & x_\xi y_\eta - x_\eta y_\xi \end{pmatrix}, \tag{20}$$

where

$$J = \frac{\partial(\xi,\eta,\zeta)}{\partial(x,y,z)} = \left(\frac{\partial(x,y,z)}{\partial(\xi,\eta,\zeta)}\right)^{-1} = \begin{vmatrix} x_\xi & x_\eta & x_\zeta \\ y_\xi & y_\eta & y_\zeta \\ z_\xi & z_\eta & z_\zeta \end{vmatrix}^{-1} \tag{21}$$

$$= \frac{1}{x_\xi(y_\eta z_\zeta - y_\zeta z_\eta) - x_\eta(y_\xi z_\zeta - y_\zeta z_\xi) + x_\zeta(y_\xi z_\eta - y_\eta z_\xi)}. \tag{22}$$

These expressions can be used to numerically determine the values of $\frac{\partial(\xi,\eta,\zeta)}{\partial(x,y,z)}$. In this project, we employ their conservative forms as derived by Visbal and Gaitonde [27], in order to avoid uniform flow errors. For example, the corresponding terms for $\frac{\partial}{\partial x}$ are

$$\xi_x = J\big((y_\eta z)_\zeta - (y_\zeta z)_\eta\big), \tag{23}$$

$$\eta_x = J\big((y_\zeta z)_\xi - (y_\xi z)_\zeta\big), \tag{24}$$

$$\zeta_x = J\big((y_\xi z)_\eta - (y_\eta z)_\xi\big). \tag{25}$$

By rearranging terms, we obtain

$$U'_t = -J(\hat{F}'_\xi + \hat{G}'_\eta + \hat{H}'_\zeta) \tag{26}$$

where

$$\hat{F}' = J^{-1}(\xi_x F' + \xi_y G' + \xi_z H'), \tag{27}$$

$$\hat{G}' = J^{-1}(\eta_x F' + \eta_y G' + \eta_z H'), \tag{28}$$

$$\hat{H}' = J^{-1}(\zeta_x F' + \zeta_y G' + \zeta_z H'). \tag{29}$$

The transformed 3D compressible Navier-Stokes equations in perturbation form (26) are the equations that we solve numerically in our program.

2.3 Discretization

We use the finite difference method. The *computational* grid $\mathcal{C}$ is realized as a 3D uniform Cartesian integer grid. For a given number of points $\texttt{nPoints}_\ell \in \mathbb{N}$ along each dimension $\ell \in \{\xi, \eta, \zeta\}$, the computational grid consists of the points

$$\mathcal{C} = \{(\xi, \eta, \zeta) \mid \xi = 1, \ldots, \texttt{nPoints}_\xi,\ \eta = 1, \ldots, \texttt{nPoints}_\eta,\ \zeta = 1, \ldots, \texttt{nPoints}_\zeta\}. \tag{30}$$

In this report, we consider a box of dimensionless size $(\texttt{box_size}_\xi, \texttt{box_size}_\eta, \texttt{box_size}_\zeta) \in \mathbb{R}^3_+$ as physical domain $[0, \texttt{box_size}_\xi] \times [0, \texttt{box_size}_\eta] \times [0, \texttt{box_size}_\zeta]$. The *physical* Cartesian grid $\mathcal{P}$ consists of the points

$$\mathcal{P} = \left\{ (x, y, z) = \left(\frac{(\xi - 1) \cdot \texttt{box_size}_\xi}{\texttt{nPoints}_\xi - 1}, \frac{(\eta - 1) \cdot \texttt{box_size}_\eta}{\texttt{nPoints}_\eta - 1}, \frac{(\zeta - 1) \cdot \texttt{box_size}_\zeta}{\texttt{nPoints}_\zeta - 1} \right) \,\middle|\, (\xi, \eta, \zeta) \in \mathcal{C} \right\}. \tag{31}$$

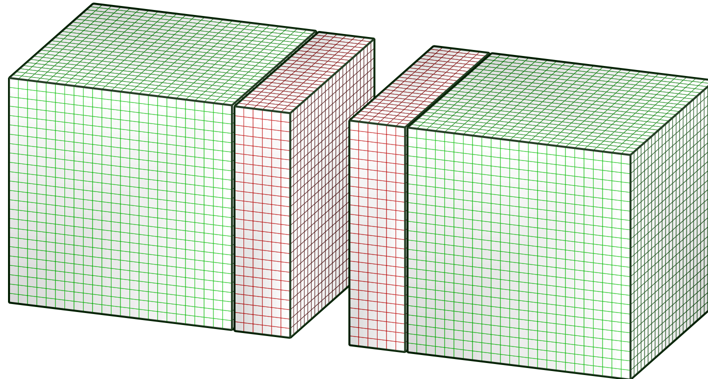
Note that in this formulation, the grid spacing (often denoted as h) has $\tilde{h}_\ell = 1$ on the computational grid and $\hat{h}_\ell = \frac{1}{\texttt{nPoints}_\ell - 1}$ $(\ell = \xi, \eta, \zeta)$ on the physical grid. Further, the distance to the boundaries of the box at 0 and $\texttt{box_size}_\ell$ from the outermost grid point equals h_ℓ on the physical grid $(\ell = \xi, \eta, \zeta)$.

The approximations of $\left(\frac{\partial}{\partial \xi}, \frac{\partial}{\partial \eta}, \frac{\partial}{\partial \zeta}\right)$ are based on globally fourth order SBP operators [4, 18]. In the interior, these operators correspond to the classical sixth order central difference operator [6]. For time integration, we employ the classical fourth order explicit Runge-Kutta method.

2.4 Domain decomposition

The computational domain is split into cuboidal subdomains, or "blocks", and the problem is solved on each subdomain individually. The central finite difference approximation's seven point stencil requires a three point overlap between two neighboring blocks. In consequence, each boundary point on a subdomain boundary has information about three ghost points in the outer normal direction, i.e. from

Figure 1: Illustration of two neighboring blocks. The overlap of the two blocks is shown with magenta lines. This illustration can be seen as 3D analogy to figure 2 in [6].



the adjacent blocks. Nonetheless, the amount of overlapping points shall in the following be treated as parameter `nOverlap` $\in \mathbb{N}$, since bigger overlaps might be desirable. Figure 1 illustrates this approach. A general introduction to domain decomposition can be found in the book by Quarteroni [16].

Unlike Khalili et al. [6], we consider walls as physical features that are independent of the computational subdomains. In particular, we generally avoid the assumption that interior walls and computational boundaries coincide. Whereas in [6], in this situation, block-to-block communication is avoided, in our approach, data of neighboring blocks will always be exchanged. This redundant communication simplifies the implementation and should have little impact on the overall performance as it overlaps with the remaining data exchange. In both implementations, communication on outer boundaries is avoided, except for periodic boundary conditions.

The communication between blocks is realized with the Message Passing Interface (MPI). This approach allows for parallel computation of the residual. The Message Passing Interface (MPI) is a standard that "allows one to code parallel programs exchanging data by sending and receiving messages." [13] Each instance of a parallely executed code is in the following called a *rank*, *process* or *thread*, and we use these expressions as synonyms, even though this at some level of detail not entirely accurate. Each process is executed on a specific CPU, which is physically installed in a *(compute) node*. Every node may consist of multiple CPUs. Therefore, a parallel MPI program (or more precisely: its threads) can be executed on multiple nodes and multiple CPUs. Node-to-node communication is realized by a network link between nodes. However, the four stages of a single Runge-Kutta step have to be computed serially.

3 Methods and implementation details

3.1 Decomposed domain indexing

Since the domain decomposition is handled on the computational grid, we only have to consider integer indices. We first define a set of splits

$$\Sigma^{(\xi)} = \{0 = \Sigma_0^{(\xi)} < ... < \Sigma_{i_{\max}}^{(\xi)} = \texttt{nPoints}_\xi\} \tag{32}$$

$$\Sigma^{(\eta)} = \{0 = \Sigma_0^{(\eta)} < ... < \Sigma_{j_{\max}}^{(\eta)} = \texttt{nPoints}_\eta\} \tag{33}$$

$$\Sigma^{(\zeta)} = \{0 = \Sigma_0^{(\zeta)} < ... < \Sigma_{k_{\max}}^{(\zeta)} = \texttt{nPoints}_\zeta\} \tag{34}$$

that divide $\mathcal{C}$ into a total amount of $i_{\max} \cdot j_{\max} \cdot k_{\max}$ subsets

$$\mathcal{C} = \bigcup_{i,j,k} \{(\xi, \eta, \zeta) \mid \xi = \Sigma_{i-1}^{(1)} + 1, \ldots, \Sigma_i^{(1)},\ \eta = \Sigma_{j-1}^{(2)} + 1, \ldots, \Sigma_j^{(2)},\ \zeta = \Sigma_{k-1}^{(3)} + 1, \ldots, \Sigma_k^{(3)}\}, \tag{35}$$

where $1 \leq i \leq i_{\max}$, $1 \leq j \leq j_{\max}$, $1 \leq k \leq k_{\max}$. The corresponding grid size is then

$$\texttt{grid_size}_{i,j,k} = \begin{pmatrix} \Sigma_i^{(1)} - \Sigma_{i-1}^{(1)} \\ \Sigma_j^{(2)} - \Sigma_{j-1}^{(2)} \\ \Sigma_k^{(3)} - \Sigma_{k-1}^{(3)} \end{pmatrix} \tag{36}$$

and when adding the ghost points, we have

$$\texttt{grid_size_ghost}_{i,j,k} = \begin{pmatrix} \Sigma_i^{(1)} - \Sigma_{i-1}^{(1)} + 2\texttt{nOverlap} \\ \Sigma_j^{(2)} - \Sigma_{j-1}^{(2)} + 2\texttt{nOverlap} \\ \Sigma_k^{(3)} - \Sigma_{k-1}^{(3)} + 2\texttt{nOverlap} \end{pmatrix}. \tag{37}$$

As starting and ending indices on the computational grid, we find

$$\texttt{idx_base}_{i,j,k} = \begin{pmatrix} \Sigma_{i-1}^{(1)} + 1 & \Sigma_i^{(1)} \\ \Sigma_{j-1}^{(2)} + 1 & \Sigma_j^{(2)} \\ \Sigma_{k-1}^{(3)} + 1 & \Sigma_k^{(3)} \end{pmatrix} \tag{38}$$

and with ghost points, we have

$$\texttt{idx_base_ghost} = \begin{pmatrix} \Sigma_{i-1}^{(1)} + 1 - \texttt{nOverlap} & \Sigma_i^{(1)} + \texttt{nOverlap} \\ \Sigma_{j-1}^{(2)} + 1 - \texttt{nOverlap} & \Sigma_j^{(2)} + \texttt{nOverlap} \\ \Sigma_{k-1}^{(3)} + 1 - \texttt{nOverlap} & \Sigma_k^{(3)} + \texttt{nOverlap} \end{pmatrix}. \tag{39}$$

3.2 Modelling of walls

In this section, we briefly describe the modeling of walls in this program. Note that walls are modeled in terms of the computational grid, not in terms of the physical grid.

In a three dimensional scenario, we model a wall as a subset of a two-dimensional hyperplane, that can be fully described by one vector that stands orthogonal on the hyperplane. For sake of simplicity, we only allow for walls parallel to the grid lines and therefore, only multiples (not equal to zero) of the three unit vectors have to be considered as possible normal vectors. We therefore set `normal_dim` $\in \{1, 2, 3\}$ and `offset` $\in \mathbb{N}$. In the case `offset` $= 0$, this approach would not be well-defined. In this case, we simply describe a hyperplane that is orthogonal to the `normal_dim`-th unit vector and goes through the origin. Because the walls are rectangular, it is sufficient to give two dimensional start and end indices $\texttt{span_idx}_{\text{start}}, \texttt{span_idx}_{\text{end}} \in \mathbb{N}^2$ along the wall span dimensions.

This approach is implemented in the FORTRAN derived type `wall_t`, see [2]. The additional type member `onThisProc` is set during grid allocation and denotes, whether this wall lies inside the domain of the current MPI rank. In particular, the parameter `walls(:)` has to be initialized before the grid allocation is started. We implement the following FORTRAN code:

```
1 type :: wall_t
2   integer :: normal_dim
3   integer :: span_idx(nDims-1,2)
4   integer :: offset
5
6   logical :: onThisProc
7 end type wall_t
```

3.3 Cache-efficient approximation of derivatives

The numerical approximation of derivatives with a summation by parts (SBP) operator is an often performed subroutine in our simulation. We therefore study the efficiency of different implementations for these operators and compare both speed and cache efficiency. Inside each CPU, we find a hierarchy of memory that typically consists of three cache levels that are connected with a bus to the main memory. Typically, we find a 32 KB first level cache, a 256 KB second level cache and a 16 MB third level cache. The first and second level caches are often present on every core, while there is one shared third level cache per CPU. While the first level cache speed is around 1 ns, we reach around 10 ns on the third level cache and around 100 ns on the main memory [5]. The *miss rate* is the share of memory accesses that cannot be served by the cache, while the *hit rate* is defined as the share of memory requests than can be served from the cache. It is our goal to minimize the penalty from reading from the main memory by reducing the number of cache misses during the execution of our compiled code. We thereby rely on

the *principle of locality*, which states that programs do not access all data or code uniformly, but close subsets of memory within a short piece of time [5].

For sake of simplicity, we assume in this section that there is only one block involved and that there are $n \in \mathbb{N}$ points in every dimension, i.e. n^3 points in total in the simulation. We further assume that there is a variable $X \in (\mathbb{R}^n)^3$ and we want to approximate $\frac{\partial X}{\partial x}$. Note that X is padded with ghost values of width `nOverlap` on each side, i.e. there are $(n - 2\texttt{nOverlap})^3$ entries of X at interior points. These ghost points are either communicated from neighboring blocks or have no physical correspondence at all. In the latter case, they remain uninitialized and are never used. The derivative approximations at the ghost points are set to zero. This corresponds to the reference implementation.

Reference and alternative implementations In the reference implementation `D1SBP`, the columns `X(:,j,k)` are copied into a buffer. If necessary, the SBP stencil at the boundaries is applied. Afterwards, the classical centered seven point stencil is applied at the interior points and the result is written into another buffer. Then, the values are copied to their final destination and the procedure is repeated for the next column of X.

However, in the scenario where walls and computational boundaries are disentangled, this approach is no longer feasible: We would have to check for every point, if it is in the range of a wall normal to the direction of derivative, which would imply $\mathcal{O}(n^3)$ checks. Instead, we flatten X into a one-dimensional buffer

$$\texttt{buf} = (\texttt{X(1,1,1)}, \dots, \texttt{X(n,1,1)}, \texttt{X(1,2,1)}, \dots, \texttt{X(n,n,1)}, \texttt{X(1,1,2)}, \dots, \texttt{X(n,n,n)})$$

and apply the seven point stencil on the whole domain. We aim to take advantage of the vectorization capability of compiler and hardware. Apparently, in this flattened representation of X, the seven point stencil is also applied to points that are physically not neighbors. As we have chosen a sufficiently large overlap of domains, false values will only be assigned to ghost points. The ghost points obtain their correct values either by communication from adjacent blocks or remain unused due to other boundary or wall treatment.

Because we know the positions of the walls with respect to the computational grid, we have to "correct" for the true values around walls by applying the SBP boundary stencil there. Since walls are hyperplanes, this can be done with only $\mathcal{O}(n^2)$ operations on the buffer. Note that the relevant walls are orthogonal to the direction of the derivative. Thus, neighboring wall points have a big distance in memory, and the flattened representation of X is not cache optimal for the wall correction. If there are

many walls, a second reshape of X along one of the wall's span dimensions could improve the performance.

Built-in functions A naive approach for the reshaping is the use of built-in functions.

```
1 buf = pack(reshape(X, (/n,n,n/), order=(/1, 2, 3/), .TRUE.)
```

Note that `reshape` is actually an identity mapping in this situation, but in case, we are considering the derivative with respect to y or z, reshaping cannot be omitted. In the following, we also consider this approach as the *naive* approach.

Optimizing cache-read hits In this approach, we access the memory of X in memory order and compute for every element in `buf` the corresponding indices of X:

```
1 do k=1,n
2   do j=1,n
3     do i=1,n
4       idx = (i-1) + (j-1) * n + (k-1) * n**2 + 1
5       buf(idx) = X(i,j,k)
6     end do
7   end do
8 end do
```

Optimizing cache-write hits In this approach, we access the memory buf `buf` in memory order, i.e. we invert the index computation from the previous approach and compute for every entry in `buf` the corresponding indices of X:

```
1 do idx=0,n**3-1
2   k = (idx) / n**2
3   j = (idx - n**2 * k) / n
4   i = (idx - n**2 * k  - n * j) / 1
5   buf(idx+1) = X(i+1,j+1,k+1)
6 end do
```

Fused add-multiply It turned out to be useful to have a fused add-multiply option in the implementation of the derivative operator. It allows to compute e.g.

$$Z \leftarrow Z + aD_\xi X$$

for $Z \in (\mathbb{R}^n)^3$ without allocating an extra buffer the the intermediate result $D_\xi X$. Here, D_ξ is the difference operator, $D_\xi X$ is the difference approximation of $\frac{\partial X}{\partial \xi}$, and $\leftarrow$ a variable assignment.

3.4 Non-blocking communication

Communication between MPI ranks takes time and may cause idle CPUs, e.g. due to network latency. In the reference implementation, the blocking communication subroutine `mpi_sendrecv` is used, i.e. the program execution will stop until all data is exchanged. One of this project's contributions is the implementation non-blocking communication.

Generally, every rank has up to six cuboidal subsets where data has to be sent and the same number of cuboidal subsets where data has to be received. The number of the cuboids depends on the number of adjacent blocks. In the allocation phase of the grid, the indices and size information of the cuboids are computed.

Each pair of these sending/receiving cuboids is represented by an element of type `sendrecv_t`. It consists of MPI tags and requests and two buffers.

```
1 type :: sendrecv_t
2   integer :: send_tag, recv_tag
3   integer :: send_request, recv_request
4   real, allocatable, dimension(:) :: send_buf, recv_buf
5 end type
```

The function `grid_sync` initializes the data exchange on the overlap between neighboring nodes by setting the tags, allocating the buffers and calling the subroutines `mpi_isend` and `mpi_irecv`. The corresponding requests are also stored in the type members. Note that the send buffer is initialized with that data that has to be sent. The function returns immediately.

The function `grid_wait` is a barrier that waits until all blocks have exchanged the data. It takes the object of type `sendrecv_t` and calls `mpi_wait` on the two request handles and writes the data from the receive buffer to its final destination. Other parts of the simulation can be done between the calls of `grid_sync` and `grid_wait` while communication happens in the background.

For sake of simplicity, we furthermore implemented a type `comm_t` that stores six elements of type `sendrecv_t` to facilitate the exchange of one variable over all six sides of a block.

3.5 HDF5 parallel I/O

The export of (possibly intermediate) simulation results to disk is essential in this program. We implemented therefore two subroutines that allow the export of data to disk using the HDF5 library [24].

HDF5 is both a hierarchical data format specification and a corresponding library implementation.

It includes "a versatile data model that can represent very complex data objects and a wide variety of metadata, a completely portable file format [...], a software library that runs on a range of computational platforms [...], a rich set of integrated performance features that allow for access time and storage space optimizations, [and] tools and applications for managing, manipulating, viewing, and analyzing the data in the collection." [3] HDF5 is also one of the recommendations in [29, see *File Systems and Data Handling]*.

In our multi-block setting, it is useful to export a specific variable for each MPI rank and its portion of the grid. The subroutine `h5_write_rank` therefore creates a new HDF5 group and writes for each MPI rank a numbered HDF5 dataset that contains the specified variable information stored on that rank, including ghost points. The subroutine `h5_write_grid` assembles the information from all ranks and writes the specified variable on the full grid in one larger dataset, excluding ghost points. Note that these operations are blocking the program until they are executed entirely.

Disk I/O is typically a time consuming process and high performance computers can make use of parallel I/O by parallel file systems. We have enabled HDF5's MPI functions and all datasets are written to disk concurrently, if that is possible. In particular, when exporting the whole grid with `h5_write_grid`, each MPI rank may write into its chunk of the dataset simultaneously.

The output filename can be implemented as parameter or is inferred by reading the environment variables `H5_NAME` and `PBS_JOBID`. As part of the development process, we also implemented a visualization tool with Python, that reads the HDF5 file and plots the projections of each dataset along each of its three dimensions. HDF5 files can also be converted into the CSV or RAW format and then be read by other postprocessing tools such as Paraview.

4 Results

In this section, we outline the results of our simulations. All tests have been run on *Vilje*, a high performance computing system maintained by NTNU, *met.no* and UNINETT Sigma. *Vilje* consists of 1404 nodes where each node has two Intel Xeon E5-2670 eight-core processors with 20 MB L3 cache and 32 GiB memory (DDR3 1600 MHz-SDRAM). Interconnect is realized with Infiniband FDR [29].

In all tests, SGI's MPI library has been used. All programs have been compiled with Intel's Fortran compiler `ifort` and the compiler options `-O3 -xHost -real-size 64`. We have experienced huge differences in run time and cache behavior with different compilers and compiler settings.

Table 1: Cache profiling of SBP operator: Best results of "new" implementation are highlighted. The abbreviations are explained in section 4.2

	Dr	D1mr	DLmr	Dw	D1mw	DLmw	run time [s]
A	25,896,541	912,381	59,280	14,117,222	1,061,609	1,050,556	1.13
B	75,836,698	4,994,782	59,384	37,344,937	5,142,497	1,765,571	1.14
C	**141,227,168**	8,286,006	65,925	**69,294,558**	10,726,828	2,955,503	1.20
D	162,627,430	**6,174,225**	**61,992**	69,637,008	8,426,806	2,717,120	**1.17**
E	155,303,540	7,821,887	62,046	70,254,631	**6,885,624**	**2,717,115**	**1.17**

4.1 Validation

In a first validation stage, we test the correct implementation of the communication between nodes and the new implementation of the SBP operator. We find that the results between the new implementation and the reference implementation are equal. We want to stress that it is crucial to use double precision reals in FORTRAN – the default compiler setting in both Intel's and GNU's compiler is to use single precision reals, which easily leads to differences of the magnitude 10^{-6} after a single application of the SBP operator.

Even though we used a tremendous share of the project time to eliminate all errors from the Navier-Stokes implementation, it was by now not possible to create a stable simulation of the Poiseuille flow as validation scenario with our program.

4.2 Cache simulation

In this benchmark, we want to compare different implementations of the SBP operator. We therefore compute the derivative of $f(\xi, \eta, \zeta) = \frac{1}{2}(\xi^2 + \eta^2 + \zeta^2)$ with respect to the first, second and third argument. The analytical result is expected to be the identity mapping of the corresponding argument.

The profiling tool `valgrind/cachegrind` [12] has been used to simulate the number of cache misses, i.e. the number of data cache reads (Dr), data cache first level missed reads (D1mr), data cache last level missed read (DLmr), data cache writes (Dw), data cache first level missed writes (D1mw) and data cache last level missed writes (DLmw). We additionally measured the run time of the program. Table 1 summarizes our results.

Experiment A shows a dummy derivative operator that immediately returns without any computations. The following experiments show the cache profiling of the "old" derivative operator (experiment B), the naive implementation (experiment C), the cache-read optimized implementation (experiment D) and the cache-write optimized implementation (experiment E), see section 3.3.

Table 2: Non-blocking communication: Run time and pagefaults after computing the metric terms with 100^3 points. The abbreviations are explained in section 4.3.

	A	B	C	D	E	F	G	H
# nodes	64	27	8	1	8	4	2	1
# threads/node	1	1	1	1	1	2	4	8
time non-blocking [s]	2.35	1.80	1.66	2.93	1.67	1.67	1.80	1.62
time blocking (benchmark) [s]	1.94	1.60	1.56	2.89	1.54	1.53	1.56	1.58
# major pagefaults non-blocking	7	7	8	7	7	7	7	7
# major pagefaults blocking	0	0	0	0	0	0	0	0

With respect to all applied performance measures, the "old" implementation has the best cache profile and the shortest runtime. However, this implementation solves a simplified problem where walls can only be modeled on block boundaries. The "new" implementation allows for arbitrary wall positions and is therefore more complex. We consider this as main cause for experiment B being the fastest of the experiments conducted.

Notably, both the cache-read optimized version and the cache-write optimized version led to the smallest number of missed reads and writes, respectively. The naive implementation has the smallest number of cache hits when both reading and writing.

The two optimized versions have the same run time, with the naive implementation being the slowest among the three flavors the of "new" implementation. Generally, the run time differences are very small, especially when the runtime of the dummy program is taken into account.

It is worth to note that the findings about the "new" implementation were the same, when using the GNU `gcc` compiler with weaker optimization settings. However, the benchmark experiment B performed worse in this setting, and experiment D was the most favorable. We therefore want to stress, that compiler choice and settings are important and a further study of the assembled or compiled code can be useful to gain a better understanding of the different SBP implementations.

4.3 Cost of communication

The calculation of metric terms (23) - (25) and similar terms for y- and z derivatives of ξ, η, ζ is a section in our code where communication and computation heavily overlap. We therefore chose as test scenario for the non-blocking communication between the MPI ranks the time it takes to compute the metric terms on a grid with 100^3 points.

As outlined, the subroutine `grid_wait` blocks the execution until all data has been transferred. We

Table 3: Strong scaling: Runtime of 500 time steps on 100^3 points with different numbers of nodes and threads. Best and worst performance are highlighted. The abbreviations are explained in section 4.4.1.

	A	B	C	D	E	F	G	H	I	J	K	L
# nodes		64			27			8			1	
# points		100^3			100^3			100^3			100^3	
# threads/node	1	8	27	1	8	27	1	8	27	1	8	27
# ranks	64	512	1728	27	216	729	8	64	216	1	8	27
# points/rank	15,625	1,953	578	37,037	4,629	1,371	125,000	15,625	4,629	1,000,000	125,000	37,037
time [hh:mm:ss]	3:36	3:25	8:24	5:10	**3:06**	5:01	14:44	5:33	4:43	**1:41:15**	24:29	15:56
gain threads	×1.05	×0.41	–	×1.67	×0.62	–	×2.65	×1.17	–	×4.14	×1.54	–
gain nodes	–	–	–	×1.44	×0.91	×0.60	×2.85	×1.79	×0.94	×6.87	×4.41	×3.38

realize a blocking version of our program by calling `grid_wait` immediately after every call of `grid_sync`.

In a first set of tests A-D, we varied the total number of nodes, while keeping the number of threads per node fixed at one. This forces the different nodes to communicate via the network and prevents intra-node communication. Surprisingly, the blocking benchmark is always faster than our new non-blocking approach, and the difference increases by the number of nodes.

In a second set of tests E-H, we kept the total number of MPI ranks constant at eight and tried different distributions of threads per node. Again, the non-blocking communication performs better than our non-blocking approach. There is no dependence of the performance on the number of threads per node in this test setting. Table 2 summarizes these test results.

A possible explanation for the advantage of the blocking implementation is the higher number of major pagefaults in the non-blocking implementation. Major pagefaults cause typically slow disk I/O operations and can therefore lead to a higher run time. Notably, this difference of pagefaults is caused in a scenario where the *same* code is executed and the order of execution is changed. Generally, this can be interpreted as consequences of spatial cache locality, i.e. the principle that close portions of memory should be referenced within a short period of time rather than distant portions of memory. The blocking communication follows this principle rather than the non-blocking communication as no other memory is referenced before the communication is finished.

4.4 Scaling

4.4.1 Strong scaling

In this section, we want to evaluate the general performance of our parallel implementation on a problem of *fixed size*. We therefore set the number of points to 100^3 and the number of time steps to 500 and

Table 4: Weak scaling: Runtime of 500 time steps on different grids with different numbers of points per MPI rank. The abbreviations are explained in section 4.4.2.

	A	B	C	D	E	F	G	H	I	J	K	L
# points/rank	1,000,000				125,000				37,037			
# threads/node	1				8				27			
# nodes	64	27	8	1	64	27	8	1	64	27	8	1
# ranks	64	27	8	1	512	216	64	8	1728	729	216	27
# points	400^3	300^3	200^3	100^3	400^3	300^3	200^3	100^3	400^3	300^3	200^3	100^3
time [hh:mm:ss]	1:52:10	1:42:42	1:40:33	1:39:38	29:20	26:50	29:06	24:22	24:35	21:12	18:12	15:58
loss	×0.89	×0.97	×0.99	–	×0.83	×0.91	×0.84	–	×0.61	×0.71	×0.83	–

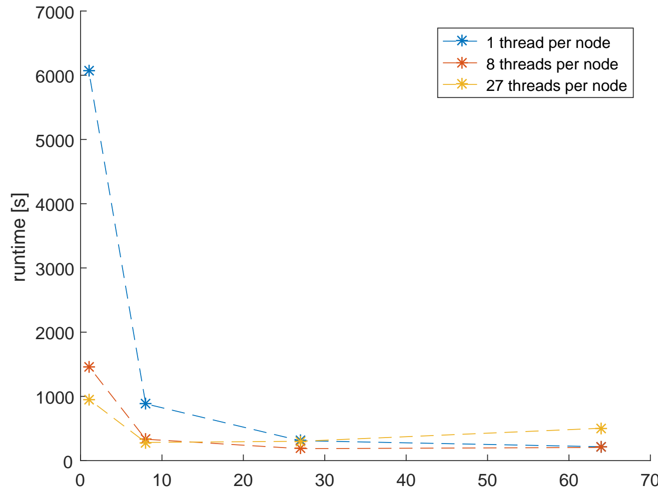
measure the total run time of the program. Table 3 and figure 2 summarize these results. The row *gain threads* denotes the run time gain when running the same problem with a higher number of threads per node and a constant number of nodes. The row *gain nodes* denotes the run time gain when running the same problem with a higher number of node and a constant number of threads per node.

Increasing the number of nodes has generally a positive effect on the runtime of our program. When there is a certain number of nodes reached, this effect saturates, as it can be seen in the smaller node gains. The same holds for increasing the number of threads per node and the smaller gains per thread added. The experiments with the highest total number of ranks, experiments C and F, illustrate this saturation. They both have more ranks than experiments B and E, but perform worse. An explanation for this phenomenon is that the cost of communication cancels out the positive effects of concurrency.

Surprisingly, the allocation of one thread on each nodes (experiment G) has a bigger impact on the performance than allocation eight threads on one node (experiment K), compared with the serial scenario (experiment J). The performance loss from using inter-nodal instead of intra-nodal communication is presumably smaller than the performance loss caused by threading. The mediocre performance of experiment C underlines this finding by suffering from both negative effects at the same time.

Looking at the total number of MPI ranks, we clearly find the outliers in scenarios with less than 10 and more than 1000 ranks. Generally, we found an optimum between 200 and 600 ranks, see experiments B, E and I, with experiment E having the shortest overall runtime. As high performance computers are usually used by multiple users with limited core hours, we want to stress that experiment I with only eight nodes yields a particularly *economic* scenario for the productive use of this program.

Figure 2: Strong scaling: Runtime of 500 time steps on 100^3 points with different numbers of nodes and threads.



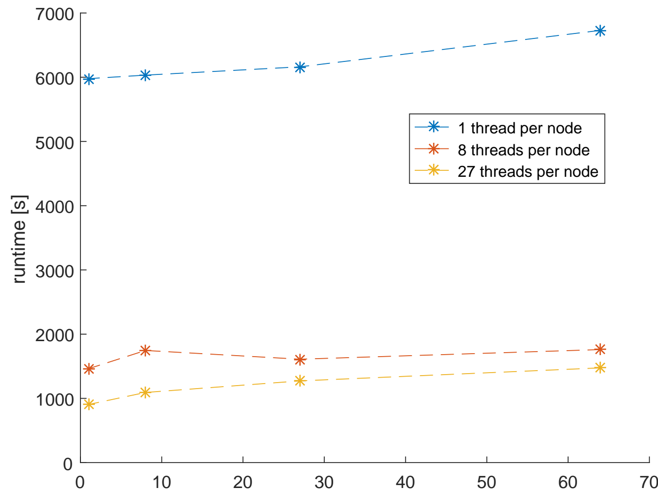
4.4.2 Weak scaling

In this section, we evaluate the performance of our program in a scenario of *fixed problem size per node*. In particular, the number of points increases as the number of nodes increases. The number of points per rank depends on both the number of nodes and the number of threads per node. Again, we measure the time it takes to compute 500 time steps. Table 4 and figure 3 summarize the scenarios and the results. The row *loss* compares run time loss when running the program on the given number of nodes with the lowest number of nodes, keeping the number of points per node constant. A loss of $\times 1.00$ would mean perfect weak scaling.

We observe almost perfect scaling when going from one to eight nodes, see experiment C. In particular, this transition was leading to the biggest gain in the study of string scaling. More generally, those combinations of nodes and threads per node that have proven to be fast in the study of strong scaling also show good weak scaling properties in this experiment, see columns F and K. This supports the robustness of the presented results.

Especially in the scenario with one thread per node, we experience a very favorable scaling behavior: Comparing experiments A and D, we can solve our problem on a grid with $\times 64$ points and a loss of only 11% time, given $\times 64$ the computational resources. Again, the experiments I and J with the highest number of ranks yield the worst scaling behavior. This can, again, be explained by the increasing cost of communication.

Figure 3: Weak scaling: Runtime of 500 time steps on different grids with different numbers of points per MPI rank.



5 Conclusions and outlook

In this project report, we presented a parallel implementation of a high order difference method for the three dimensional compressible Navier-Stokes equations by domain decomposition. In order to improve the runtime of the program not only by concurrent execution, we evaluate three cache optimized flavors of a fourth order accurate SBP operator and the effects of non-blocking communication. Both optimizations yielded little or no effect. As a handy tool for disk I/O, we built a HDF5 interface into our program. The final scaling study showed very favorable scaling properties and almost perfect weak scaling.

We think this project forms a good basis for another student project or master thesis. It can be interesting to fix the existing problems in the Navier-Stokes implementation and try out the presented framework with other partial differential equations. Another perspective could be the implementation of a moving mesh and fluid-structure interaction based on this project.

A more computer science related follow-up project could be the further profiling of caching and communication. For example, it would be interesting to see, how the SBP operator and the communication could be merged. Thread-level parallelism with OpenMP could be evaluated. Finally, as fluid-structure interaction comes into play, it might become important to think about load balancing, as several nodes have to compute both fluid and structural terms, and the number of floating point operations per node is expected to be different for fluid and structure nodes.

References

[1] Mark H. Carpenter, David Gottlieb, and Saul Abarbanel. Time-Stable Boundary Conditions for Finite-Difference Schemes Solving Hyperbolic Systems: Methodology and Application to High-Order Compact Schemes. *Journal of Computational Physics*, 111(2):220 – 236, 1994.

[2] Stephen J. Chapman. *Fortran 95/2003 for scientists and engineers.* McGraw-Hill Higher Education, third edition, 2008.

[3] The HDF Group. What is hdf5?, accessed 2017-05-19. https://support.hdfgroup.org/HDF5/whatishdf5.html.

[4] Bertil Gustafsson. *High Order Difference Methods for Time Dependent PDE*, volume 38 of *Springer Series in Computational Mathematics.* Springer, 2008.

[5] John L. Hennessy, David A.. Patterson, and Krste Asanović. *Computer architecture : a quantitative approach.* Elsevier, fifth edition, 2012.

[6] Mohammadtaghi Khalili, Martin Larsson, and Bernhard Müller. Interaction between a simplified soft palate and compressible viscous flow. *Journal of Fluids and Structures*, 67:85 – 105, 2016.

[7] Hans-Otto Kreiss and Godela Scherer. Finite element and finite difference methods for hyperbolic partial differential equations. In Carl de Boor, editor, *Mathematical Aspects of Finite Elements in Partial Differential Equations*, pages 195–212. Academic Press, Inc., 1974.

[8] Hans-Otto Kreiss and Godela Scherer. On the existence of energy estimates for difference approximations for hyperbolic systems. Technical report, Uppsala University, 1977.

[9] Heinz-Otto Kreiss and Joseph Oliger. Comparison of accurate methods for the integration of hyperbolic equations. *Tellus*, 24(3):199–215, 1972.

[10] Ken Mattsson, Magnus Svärd, Mark Carpenter, and Jan Nordström. High-order accurate computations for unsteady aerodynamics. *Computers & Fluids*, 36(3):636 – 649, 2007.

[11] Bernhard Müller. High order numerical simulation of aeolian tones. *Computers & Fluids*, 37(4): 450 – 462, 2008.

[12] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[13] Frank Nielsen. *Introduction to HPC with MPI for Data Science.* Undergraduate Topics in Computer Science. Springer International Publishing, 2016.

[14] Richard H. Pletcher, John C. Tannehill, and Dale A. Anderson. *Computational Fluid Mechanics and Heat Transfer.* Series in Computational and Physical Processes in Mechanics and Thermal Sciences. CRC Press, third edition, 2013.

[15] T. J. Poinsot and S. K. Lele. Boundary conditions for direct simulations of compressible viscous flows. *Journal of Computational Physics*, 101:104–129, 1992.

[16] Alfio Quarteroni. *Domain decomposition methods for partial differential equations.* Numerical mathematics and scientific computation. Clarendon Press, 1999.

[17] Research project "Modeling of Obstructive Sleep Apnea by Fluid-Structure Interaction in the Upper Airways". Project description, accessed 2017-05-22. http://osas.no/description.

[18] Bo Strand. Summation by Parts for Finite Difference Approximations for d/dx. *Journal of Computational Physics*, 110(1):47 – 67, 1994.

[19] M. Svärd and J. Nordström. Review of summation-by-parts schemes for initial-boundary-value problems. *Journal of Computational Physics*, 268:17–38, 2014.

[20] Magnus Svärd and Siddhartha Mishra. Entropy stable schemes for initial-boundary-value conservation laws. *Zeitschrift für Angewandte Mathematik und Physik*, 63:985–1003, 2012.

[21] Magnus Svärd and Jan Nordström. On the order of accuracy for difference approximations of initial-boundary value problems. *Journal of Computational Physics*, 218(1):333 – 352, 2006.

[22] Magnus Svärd and Jan Nordström. A stable high-order finite difference scheme for the compressible Navier–Stokes equations: No-slip wall boundary conditions. *Journal of Computational Physics*, 227(10):4805 – 4824, 2008.

[23] Magnus Svärd, Johan Lundberg, and Jan Nordström. A computational study of vortex–airfoil interaction using high-order finite difference methods. *Computers & Fluids*, 39(8):1267 – 1274, 2010.

[24] The HDF Group. Hierarchical Data Format, version 5, 1997-2017. http://www.hdfgroup.org/HDF5/.

[25] Petra Tisovská. *High Order Numerical Simulation of Viscous Compressible Flow in a Simplified Geometry of the Human Upper Airways*. Department of Energy and Process Engineering, Norwegian University of Science and Technology, 2017. Unpublished master's thesis.

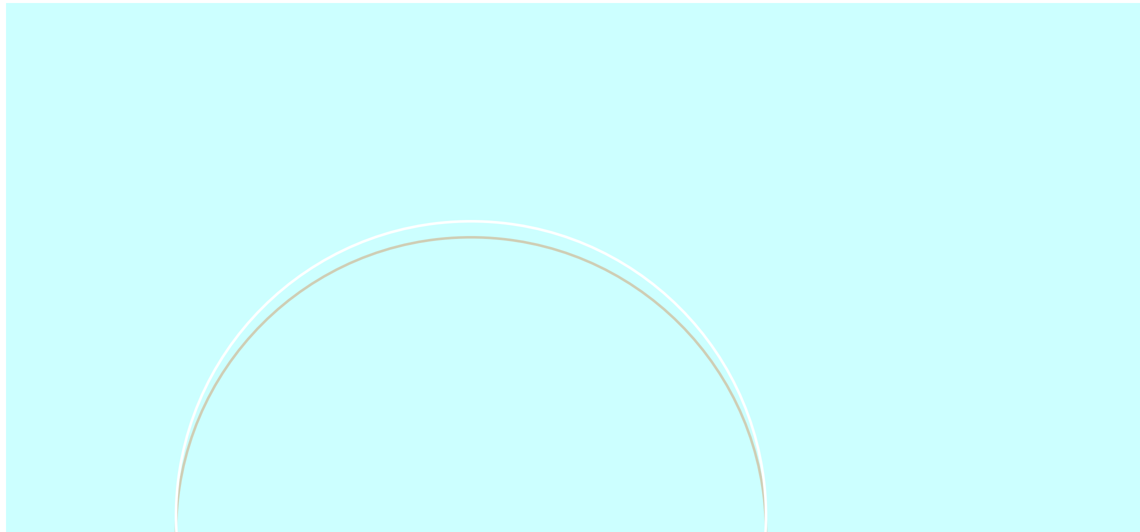
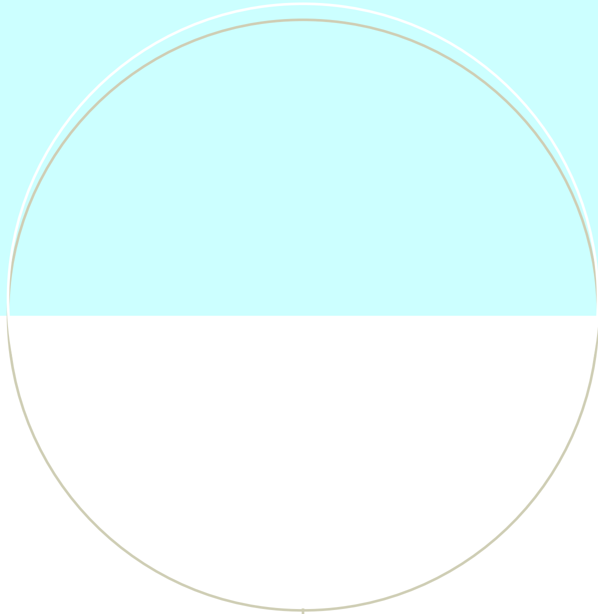
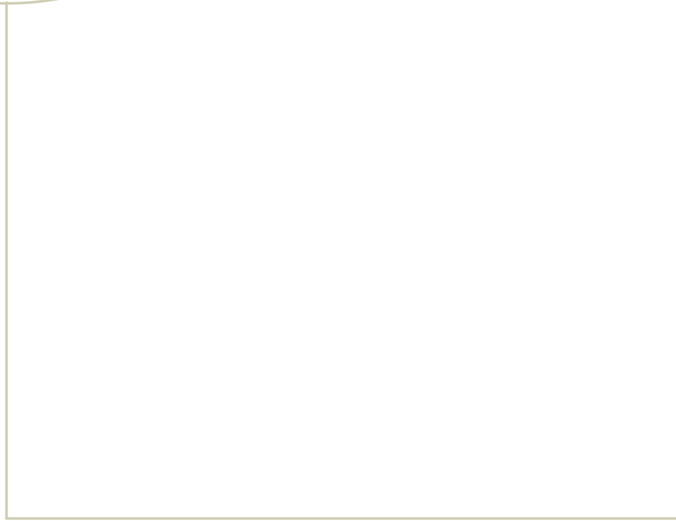
[26] J. M. F. Trindade and J. C. F. Pereira. Parallel-in-time simulation of the unsteady Navier–Stokes equations for incompressible flow. *International Journal for Numerical Methods in Fluids*, 45(10): 1123–1136, 2004.

[27] Miguel R. Visbal and Datta V. Gaitonde. On the Use of Higher-Order Finite-Difference Schemes on Curvilinear and Deforming Meshes. *Journal of Computational Physics*, 181(1):155 – 185, 2002.

[28] Edwin van der Weide, Giorgio Giangaspero, and Magnus Svärd. Efficiency Benchmarking of an Energy Stable High-Order Finite Difference Discretization. *AAIA (American Institute of Aeronautics and Astronautics) Journal*, 53(7):1845–1860, 2015.

[29] HPC Wiki. About Vilje, accessed 2017-05-19. https://www.hpc.ntnu.no/display/hpc/About+Vilje.

[30] Jakub Šístek and Fehmi Cirak. Parallel iterative solution of the incompressible Navier–Stokes equations with application to rotating wings. *Computers & Fluids*, 122:165 – 183, 2015.

NTNU – Trondheim
Norwegian University of
Science and Technology