Parallelization of a Winslow variational mesh generator

by

© **Oleksandr Abramov**

A thesis submitted to the School of Graduate Studies in partial fulfillment of the requirements for the degree of Master of Science.

Scientific Computing

Memorial University

December 2018

St. John's, Newfoundland and Labrador, Canada

Abstract

Adaptive meshes appear to be preferable to the uniform meshes for the numerical approximations of some PDEs. As the solution of large scale numerical problems often involves solving a problem on a distributed computational systems (clusters), it is desirable for a mesh adaptation method to also be deployable on such systems. We study various 1D mesh adaptation methods and theoretically explore connection between some of them. An implementation of a library for a parallel 2D Winslow adaptive mesh method is be presented. The library is written in the C++ language and uses the PETSc library and the Message Passing Interface (MPI) protocol. Also an example of using the library with a Stochastic Domain Decomposition (SDD) approach based on a Monte Carlo method is discussed with some numerical results provided.

Lay summary

Nowadays, more and more tasks in different fields of interest, from chemistry to astronomy, from biology to economics, are being solved by computer simulations. In material science, for example, the simulation is much cheaper than making "in-field" experiments, in weather prediction or astronomy such experiments are not possible at all. As the processes being simulated are often described by partial differential equations (PDEs), this usually leads to solving a numerical problem on some kind of discrete mesh. For some of these problems a researcher wants or needs to concentrate a mesh on a specific part of the domain. In a tsunami simulation, for example, the coastal areas are of particular importance. Moreover, for some problems the domain of interest will move with time, like in the case of a shock-wave simulation. This raises the question of how to adapt the layout of the mesh to those needs.

One of the methods for mesh adaptation introduces a function which specifies a desired density of mesh nodes in the area. This function is called a mesh density function or a monitor function. The procedure then consists of moving the nodes so that each piece of the mesh will have an "even share" of this function.

There are different ways of implementing the procedure that will move the nodes. In my thesis I am focusing on a method that introduces a mesh PDE, the solution of which gives the desired mesh. This is an example of a variational mesh adaptation method.

In the first part of my thesis I discuss the basics of variational mesh adaptation. Next I will introduce an implementation of the Winslow method, that can benefit from using a parallel computing cluster. The next part introduces the idea of using Monte Carlo methods for splitting the domain into subdomains. On each of the subdomains we can solve the mesh problem independently. In some configurations this can improve the parallelization effectiveness and the total performance.

Acknowledgements

First of all I want to express my gratitude to my supervisors Dr. Alex Bihlo and Dr. Ronald D. Haynes. They made this work possible and guided me through all my research.
With Fabrizio Donzelli I spent a lot of hours working on the code. Paul Sherren helped me to setup the experiments on a local cluster Torngat. The same was done by Oliver Stueker for Compute Canada cluster Graham. Jed Brown and Matthew G. Knepley really helped me with using PETSc library in my code. Also Matthew G. Knepley and Scott MacLachlan gave me some advises about linear solvers to use in my experiments.
I thank all these people for the contribution they made to my research. And I thank Memorial University for creating the enviroment were it all was possible to do.

Statement of contribution

Dr. Ronald D. Haynes and Dr. Alex Bihlo suggested the idea of the research and supervised the entire research. Theorems in Chapter 2.1.3 were introduced and proven by myself. The library described in Appendix A was implemented by myself with the basic code example provided by Dr. Jed Brown. The code for stochastic domain decomposition was implemented in collaboration with Dr. Fabrizio Donzelli. The preparation of the thesis manuscript was done by myself under the supervision of Dr. Ronald D. Haynes and Dr. Alex Bihlo.

Table of contents

Title page i

Abstract ii

Lay summary iii

Acknowledgements iv

Statement of contribution v

Table of contents vi

List of tables ix

List of figures xv

List of abbreviations xviii

1 Introduction 1

2 PDE mesh generation 6

2.1 Adaptive mesh generation in 1D 6

2.1.1 Equidistribution principle 6

2.1.2 Equidistribution quality measure 8

2.1.3 Solution methods . . . 9
2.1.4 Results . . . 19
2.2 Mesh generation methods in 2D . . . 20
2.2.1 Algebraic methods . . . 20
2.2.2 Differential equation methods . . . 22
2.2.3 Variational methods . . . 23
2.2.4 Winslow mesh generator . . . 24

3 Domain decomposition **27**
3.1 Overlapping domain decomposition methods . . . 28
3.2 Non-overlapping domain decomposition . . . 30
3.3 Schwarz methods as preconditioners . . . 32
3.4 Stochastic domain decomposition . . . 33

4 Results **36**
4.1 Parallelization of the linear solver . . . 37
4.2 SDD convergence studies . . . 40
4.3 SDD performance . . . 52

5 Conclusions **63**

Bibliography **66**

A Parallel library documentation **69**

B Parallel environments' tests **76**
B.1 HYPRE preconditioner . . . 77
B.1.1 Torngat cluster . . . 77
B.1.2 Graham cluster . . . 80

B.2 GAMG preconditioner 84
B.2.1 Torngat cluster 84
B.2.2 Graham cluster 88
B.3 LU preconditioner 92
B.3.1 Torngat cluster 92
B.3.2 Graham cluster 94

List of tables

2.1 Number of steps until convergence for different 1D mesh adaptation methods. "NC" indicates that the algorithm did not converge. 19

4.1 Time scaling of the PETSc example code with HYPRE preconditioner on Torngat cluster, time is minimized between three repetitions. 37

4.2 Time scaling of the PETSc example code with HYPRE preconditioner on Graham cluster, time is minimized between three repetitions. 38

4.3 Time scaling of our single domain solver for a constant mesh density function with HYPRE preconditioner on Graham cluster, time is minimized between three repetitions. 39

4.4 Time scaling of our single domain solver for a constant mesh density function with GAMG preconditioner on Graham cluster, time is minimized between three repetitions. 40

4.5 Timing results for the stochastic solver with the number of Monte Carlo simulations scaling and different interpolation steps, the uniform placing of the interpolation points is used. 44

4.6 Timing results for the stochastic solver with number of Monte Carlo simulations scaling and different number of additional interpolation points, adaptive placing of the interpolation points is used. 47

4.7 Timing results for the stochastic solver with time step scaling and different interpolation steps, uniform placing of the interpolation points is used. 50

4.8 Timing results for the stochastic solver with time step scaling scaling and different interpolation number of additional interpolation points, adaptive placing of the interpolation points is used. 51

4.9 Timing results for the stochastic solver with number of cores scaling and different interpolation steps, uniform placing of the interpolation points is used. 51

4.10 Timing results for the stochastic solver with number of cores scaling and different number of additional interpolation points, adaptive placing of the interpolation points is used. 51

4.11 Timing results on a single domain. GAMG preconditioner is used. Size parameter indicates the mesh with $size \times size$ nodes. 52

4.12 Timing results for SDD with two subdomains, total time. $N = 10^5$ Monte Carlo simulations, $\Delta t = 10^{-3}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes. . 53

4.13 Timing results for SDD with two domains, stochastic solver's time. $N = 10^5$ Monte Carlo simulations, $\Delta t = 10^{-3}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes. 54

4.14 Timing results for SDD with two domains, single domain solver's time. $N = 10^5$ Monte Carlo simulations, $\Delta t = 10^{-3}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes. 54

4.15 Timing results for SDD with one subdomain per processor, total time. $N = 10^5$ Monte Carlo simulations, $\Delta t = 10^{-3}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes. 55

4.16 Timing results for SDD with one subdomain per processor, stochastic solver's time. $N = 10^5$ Monte Carlo simulations, $\Delta t = 10^{-3}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes. 55

4.17 Timing results for SDD with one subdomain per processor, single domain solver's time. $N = 10^5$ Monte Carlo simulations, $\Delta t = 10^{-3}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes. 56

4.18 Timing results for SDD with two subdomains, total time. $N = 10^6$ Monte Carlo simulations, $\Delta t = 10^{-4}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes. . 58

4.19 Timing results for SDD with two domains, stochastic solver's time. $N = 10^6$ Monte Carlo simulations, $\Delta t = 10^{-4}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes. 58

4.20 Timing results for SDD with two domains, single domain solver's time. $N = 10^6$ Monte Carlo simulations, $\Delta t = 10^{-4}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes. 59

4.21 Timing results for SDD with one subdomain per processor, total time. $N = 10^6$ Monte Carlo simulations, $\Delta t = 10^{-4}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes. 59

4.22 Timing results for SDD with one subdomain per processor, stochastic solver's time. $N = 10^6$ Monte Carlo simulations, $\Delta t = 10^{-4}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes. 60

4.23 Timing results for SDD with one subdomain per processor, single domain solver's time. $N = 10^6$ Monte Carlo simulations, $\Delta t = 10^{-4}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes. 60

B.1 Minimized timing results for the PETSc example code with HYPRE preconditioner on Torngat cluster. 77

B.2 Maximum time over minimum time for the PETSc example code with HYPRE preconditioner on Torngat cluster. 77

B.3 Time scaling over the domain for the PETSc example code with HYPRE preconditioner on Torngat cluster. 78
B.4 Minimized timing results for the single domain solver with a constant mesh density function and HYPRE preconditioner on Torngat cluster. . 78
B.5 Maximum time over minimum time for the single domain solver with a constant mesh density function and HYPRE preconditioner on Torngat cluster. 79
B.6 Time scaling over the domain for the single domain solver with a constant mesh density function and HYPRE preconditioner on Torngat cluster. 79
B.7 Double PETSc example code time over single domain solver time with HYPRE preconditioner on Torngat cluster. 79
B.8 Minimized timing results for the PETSc example code with HYPRE preconditioner on Graham cluster. 80
B.9 Maximum time over minimum time for the PETSc example code with HYPRE preconditioner on Graham cluster. 80
B.10 Time scaling over the domain for the PETSc example code with HYPRE preconditioner on Graham cluster. 81
B.11 Torngat cluster time over Graham cluster time for the PETSc example code with HYPRE preconditioner. 81
B.12 Minimized timing results for the single domain solver with a constant mesh density function and HYPRE preconditioner on Graham cluster. . 82
B.13 Maximum time over minimum time for the single domain solver with a constant mesh density function and HYPRE preconditioner on Graham cluster. 82
B.14 Time scaling over the domain for the single domain solver with a constant mesh density function and HYPRE preconditioner on Graham cluster. 83
B.15 Double PETSc example code time over single domain solver time with HYPRE preconditioner on Graham cluster. 83

B.16 Torngat cluster time over Graham cluster time for the single domain solver with HYPRE preconditioner. 84
B.17 Minimized timing results for the PETSc example code with GAMG preconditioner on Torngat cluster. 84
B.18 Maximum time over minimum time for the PETSc example code with GAMG preconditioner on Torngat cluster. 85
B.19 Time scaling over the domain for the PETSc example code with GAMG preconditioner on Torngat cluster. 85
B.20 Minimized timing results for the single domain solver with a constant mesh density function and GAMG preconditioner on Torngat cluster. . 86
B.21 Maximum time over minimum time for the single domain solver with a constant mesh density function and GAMG preconditioner on Torngat cluster. 86
B.22 Time scaling over the domainfor the single domain solver with a constant mesh density function and GAMG preconditioner on Torngat cluster. 87
B.23 Double PETSc example code time over single domain solver time with GAMG preconditioner on Torngat cluster. 87
B.24 Minimized timing results for the PETSc example code with GAMG preconditioner on Graham cluster. 88
B.25 Maximum time over minimum time for the PETSc example code with GAMG preconditioner on Graham cluster. 88
B.26 Time scaling over the domain for the PETSc example code with GAMG preconditioner on Graham cluster. 89
B.27 Torngat cluster time over Graham cluster time for the PETSc example code with GAMG preconditioner. 89
B.28 Minimized timing results for the single domain solver with a constant mesh density function and GAMG preconditioner on Graham cluster. . 90

B.29 Maximum time over minimum time for the single domain solver with a constant mesh density function and GAMG preconditioner on Graham cluster. 90

B.30 Time scaling over the domain for the single domain solver with a constant mesh density function and GAMG preconditioner on Graham cluster. 91

B.31 Double PETSc example code time over single domain solver time with GAMG preconditioner on Graham cluster. 91

B.32 Torngat cluster time over Graham cluster time for the single domain solver with a constant mesh density function and GAMG preconditioner. 92

B.33 Minimized timing results for the PETSc example code with LU preconditioner on Torngat cluster. 92

B.34 Maximum time over minimum time for the PETSc example code with LU preconditioner on Torngat cluster. 93

B.35 Time scaling over the domain for the PETSc example code with LU preconditioner on Torngat cluster. 93

B.36 Minimized timing results for the PETSc example code with LU preconditioner on Graham cluster. 94

B.37 Maximum time over minimum time for the PETSc example code with LU preconditioner on Graham cluster. 94

B.38 Time scaling over the domain for the PETSc example code with LU preconditioner on Graham cluster. 95

B.39 Torngat cluster time over Graham cluster time for the PETSc example code with LU preconditioner. 95

List of figures

2.1 Illustration of the piecewise linear interpolation in the IBVP method. . 15

2.2 Mapping curved four-sided figure into a unit square. 21

3.1 Classic example of the domain for the Schwarz method. 28

3.2 Illustration of stochastic domain decomposition. 34

4.1 Mapping of the uniform mesh to the computational domain (left) and a new interpolated mesh in the physical domain (right) with the mesh density function (4.1) with 101 nodes in X direction and 201 nodes in Y direction produced by a single domain solver. Every $4th$ line is plotted. 41

4.2 The maximum distance in the computational space between the interface nodes and the corresponding nodes obtained by a single domain solver (left) and the equidistributing quality measure of the newly obtained meshes (right). The uniform placing of the interpolation points is used. 43

4.3 The distance in the computational space between the interface nodes and the corresponding nodes obtained by a single domain solver, the uniform placing of the interpolation points with the interpolation step 4 is used. 44

4.4 The maximum distance in the computational space between the interface nodes and the corresponding nodes obtained by a single domain solver (left) and the equidistributing quality measure of the newly obtained meshes (right). The adaptive placing of the interpolation points is used. 45

4.5 Mapping of the uniform mesh to the computational domain (left) and a new interpolated mesh in the physical domain (right) with the mesh density function (4.1) with 2 domains with 101×101 nodes in both directions, 1 additional interpolation point per interval for the adaptive placing of the interpolation points and $N = 10^7$ Monte Carlo simulations per interpolation point. Every $4th$ line is plotted. '-*-' line indicates the interface. 46

4.6 The maximum distance in the computational space between the interface nodes and the corresponding nodes obtained by a single domain solver (left) and the equidistributing quality measure of the newly obtained meshes (right). The adaptive placing of the interpolation points is used. 46

4.7 The maximum distance in the computational space between the interface nodes and the corresponding nodes obtained by a single domain solver (left) and the equidistributing quality measure of the newly obtained meshes (right). The uniform placing of the interpolation points is used. 48

4.8 Mapping of the uniform mesh to the computational domain (left) and a new interpolated mesh in the physical domain (right) with the mesh density function (4.1) with 2 domains with 101×101 nodes in both directions, $N = 10^6$ Monte Carlo simulations per each interface node and $\Delta t = 1$. Every $4th$ line is plotted. '-*-' line indicates the interface. 49

4.9 Mapping of the uniform mesh to the computational domain (left) and a new interpolated mesh in the physical domain (right) with the mesh density function (4.1) with 2 domains with 101×101 nodes in both directions, $N = 10^6$ Monte Carlo simulations per each interface node and $\Delta t = 10^{-2}$. Every $4th$ line is plotted. '-*-' line indicates the interface. 49

4.10 The maximum distance in the computational space between the interface nodes and the corresponding nodes obtained by a single domain solver (left) and the equidistributing quality measure of the newly obtained meshes (right). The adaptive placing of the interpolation points is used. 50

4.11 Equidistribution quality measure for the meshes produced with different SDD approaches. $N = 10^5$ Monte Carlo simulations per interpolation point, $\Delta t = 10^{-3}$ and 4 additional interpolation points per interval. From left to right, from top to bottom: 257×257 nodes, 513×513 nodes, 1025×1025 nodes, 2049×2049 nodes. 57

4.12 Equidistribution quality measure for the meshes produced with different SDD approaches. $N = 10^6$ Monte Carlo simulations per interpolation point, $\Delta t = 10^{-4}$ and 4 additional interpolation points per interval. From left to right, from top to bottom: 257×257 nodes, 513×513 nodes, 1025×1025 nodes, 2049×2049 nodes. 61

4.13 Mesh mapping to the computational space with folding. Mesh size 4097×4097 nodes, 128 cores, single core per subdomain, $N = 10^6$ Monte Carlo simulations, $\Delta t = 10^{-4}$ and 4 additional interpolation points. Every $64th$ line is plotted. 62

List of abbreviations

BiCGStab	biconjugate gradient stabilized method
BVP	boundary value problem
DD	domain decomposition
GMRES	generalized minimal residual method
IBVP	inverse boundary value problem
MPI	Message Passing Interface
PDE	partial differential equation
PETSc	Portable, Extensible Toolkit for Scientific Computation
RAS	restricted additive Schwarz method
SDD	stochastic domain decomposition
TFI	transfinite interpolation

Chapter 1

Introduction

One of the main goals of studying a physical or economic (for example) system is the ability to predict its behavior in different situations or states. To do this, the system is often described in terms of (usually mathematical) laws.

To predict the system's behavior one needs to solve some kind of mathematical problem. The laws that describe the system often lead to equations or systems of equations, that are difficult to be solved analytically. To deal with this, a number of numerical methods have been developed which allow us to approximate the solution of the system on a computer. If the system is described by partial differential equations (PDEs) one can use, for example, finite difference or finite element methods. They allow one to find an approximate value of the solution at a finite number of points (mesh points).

Finite difference methods approximate the derivatives of the function as a linear combination of the values of the function at nearby points. Depending on the exact linear combination (discretization scheme) the approximation of the solution can have different accuracy. The discretization schemes are usually derived using Taylor series. Finite element methods approximate a solution as a linear combination of a basis functions from some class of functions. The coefficients of the linear combination are unknown. The approximation of the solution should satisfy the weak form of the problem. For boundary value problems this leads to solving a system of linear equations. Also, for some problems other numerical methods suit better, for example finite volume or spectral methods. The discretization methods are described in various books, with focus on different problems. For example, [21] covers the basics of such

different methods.

The set of points where the approximate solution is calculated is called a mesh. The quality of such a numerical solution depends on the placement of the mesh points. For some problems uniform (evenly distributed) meshes work very well. But for other problems concentrating more points around some specific areas of the solution domain may be preferable. There may be different reasons for that, a domain may have a complicated geometry, or the solution may change rapidly in some region, etc. For example, in the case of tsunami simulation the grid resolution near the shoreline needs to be relatively high, but if one tries to use such a high resolution for the whole solution domain (a large piece of the ocean) the resulting problem will require too much time and memory to be solved. In this case, the mesh nodes remain fixed even though the solution changes with time. But in some problems the mesh should be concentrated around the areas where the solution changes rapidly. Then the mesh layout should be changed during the process of solving the problem. Therefore methods to automatically create such meshes are required.

There are different ways to improve the quality of the solution by mesh adaptation. h-refinement methods decrease the distance between the mesh points (possibly locally) thus increasing the total number of points. For p-refinement methods higher order elements are used to approximate the solution between the mesh points. We focus here on r-refinement, where the number of points stays fixed but their positions are changed to reduce the error in the solution. In general using a uniform mesh may be cost prohibitive for some problems. The number of uniformly spaced points needed to obtain the desired accuracy may not be known in advance. The construction of an adaptive mesh does add an overhead to the calculations but may be reduced by using parallel methods. This is one of the primary goals of this thesis.

In 1D an equidistribution principle, described in Section 2.1.1, in combination with a mesh density function, which controls the distribution of the grid points along the interval, defines an unique mesh. To find this mesh a nonlinear system of equations needs to be solved. Different methods use different approaches to do this. The nonlinear equation can be solved in a straightforward way using Newton's method. Linearized boundary value problem (BVP) method, suggested in [20], linearizes the system by applying a time lag to the mesh density function. De Boor's method [11]

substitutes the mesh density function with a piecewise constant or a piecewise linear approximation. An inverse boundary value problem (IBVP) instead of directly finding an equidistributed mesh finds a mapping of the existing mesh to the computational space and produces a new mesh by interpolating the existing mapping. The performance and convergence of these methods is different and is discussed in Section 2.1.

In higher dimensions things are more complicated. Algebraic grid generation methods were developed to create grids for domains with a complex geometry. They usually map the domain from the initial space, called the physical space, to another space, called the computational space, in a way that simplifies the geometry of the domain in the computational space. The multisurface method [14] and the transfinite interpolation (TFI) method [19] are two examples of algebraic mesh generation methods. Algebraic grid generation methods have disadvantages such as mesh folding and the problem becoming more complex after coordinate transformation. It is difficult to control the exact distribution of the mesh points and some other important properties of the mesh that can affect the quality of the approximate solution. To overcome some of these problems differential equation based mesh generation methods were introduced. In these methods the distribution of mesh points is controlled by a partial differential equation (PDE) or a system of PDEs. The meshes produced this way are usually smoother than the ones obtained with the algebraic mesh generators. Details of the corresponding coordinate transformations can be found, for example, in [16]. To further study the properties of the meshes obtained with the differential equation based mesh generators, variational mesh generation was developed. In this approach the mesh point distribution is controlled by optimizing the functionals that affect different properties of the mesh.

Using most of the mesh adaptation methods leads to solving a linear system at the end. Even when using adaptive meshes the resulting system can be too large to be handled by a single machine, especially for higher dimensional problems. So, we need a method to efficiently solve such systems on modern computational hardware, which utilizes a distributed memory model. It can be done by splitting the problem into independent subproblems, which can then be solved independently. A number of methods allowing to do this originate from the Schwarz method [31]. The Schwarz method was created for problems with a complex domain geometry, but which could

be decomposed into simple shapes. With computers being used for solving mathematical problems, some modifications of the Schwarz method were developed, such as the additive Schwarz method [13] and the restricted additive Schwarz method (RAS) [9]. While the aforementioned methods use overlapping domain decomposition, other methods can work with non-overlapping domains. Examples for such methods are FETI and Neumann-Neumann methods [23]. With the development of the Krylov subspace methods, it was noticed that using Schwarz methods as preconditioners for Krylov-type methods shows better performance than using them as standalone methods.

Another way to achieve non-overlapping domain decomposition is to compute the approximate solution along some interface that will split the domain into independent subdomains. This can be done by using the stochastic methods for solving the problem at the particular points of the interface. This method is stochastic domain decomposition (SDD). The downside of the stochastic methods are their slow convergence rates, so they cannot be used to obtain the solution everywhere.

The motivation for this research is to compare how different modern parallel linear solvers perform when used for Winslow's method for mesh generation and to study if the SDD method improves the resulting performance. There are a number of different libraries for scientific computations, like the Intel Math Kernel Library, Eigen, Portable Extensible Toolkit for Scientific Computation (PETSc) and Trilinos. We decided to use PETSc, which utilizes Message Passing Interface (MPI) for the distributed memory model and allows to easily switch between different linear solvers. The Winslow mesh generation routine was implemented in a C++ library, as described in Appendix A. Also the code for SDD using Monte Carlo method was implemented as well. The results are discussed in Chapter 4.

The remainder of the thesis is organized as follows.

Chapter 2 starts with introduction to the equidistributing mesh generation in 1D in detail, with a comparison for the different methods of mesh adaptation on the sample problem and some theoretical results provided. Continuing with introducing different mesh generation approaches that were developed over the past decades, starting from algebraic mesh generation, then covering differential equation based and variational mesh generation. Finally, an overview of 2D adaptive mesh generators provided and the Winslow mesh generator is described.

For a distributed memory model that is used for large clusters, we need to split the problem, at least temporarily, into independent subproblems. In Chapter 3 we discuss different methods based on the Schwarz method with overlapping and non-overlapping subdomains. Then Krylov methods are introduced, for which the previously discussed methods can be used as preconditioners. After that the stochastic domain decomposition (SDD) methods are explained, which constitute a very different approach for domain decomposition.

In Chapter 4 we discuss the results obtained with our implementation of the methods mentioned in previous chapters on different hardware and on different problems. We start by testing the parallel environment and the parallel linear solvers, then go through testing the Monte Carlo implementation and finish with comparing the results of the SDD version with a traditional linear solver parallelization. In some situations we see a benefit of the SDD approach over parallel linear solver.

Chapter 5 contains the conclusions of the work done for this thesis with a short discussion of possible further studies.

Appendix A contains the documentation for the library that implements the parallel Winslow mesh generator using PETSC library.

A detailed study containing numerical results obtained on different clusters using various other linear solvers is presented in Appendix B.

Chapter 2

PDE mesh generation

This chapter covers the fundamental concepts on which the mesh generators are based. A number of different methods of mesh generation are introduced and a comparison is made for a sample problem.

2.1 Adaptive mesh generation in 1D

2.1.1 Equidistribution principle

Given a strictly positive function $\rho(x) > 0$ on interval $[a, b]$, define

$$\sigma = \int_a^b \rho(x)dx.$$

A mesh $\{x_j\}_{j=1}^N : a = x_1 < x_2 < \ldots < x_N = b$ is called equidistributed with respect to the mesh density function $\rho(x)$ if

$$\int_{x_1}^{x_2} \rho(x)dx = \int_{x_2}^{x_3} \rho(x)dx = \cdots = \int_{x_N-1}^{x_N} \rho(x)dx = \frac{\sigma}{N-1}.$$

Adding the first j of the integrals above gives

$$\int_a^{x_j} \rho(x)dx = \frac{j-1}{N-1}\sigma, \quad j = 1, \ldots, N. \tag{2.1}$$

Consider a continuous mesh transformation $x = x(\xi) : [0,1] \to [a,b]$ such that $x_j = x(\xi_j)$, where $\xi_j = (j-1)/(N-1)$. Then (2.1) can be rewritten as

$$\int_a^{x(\xi_j)} \rho(x)dx = \xi_j \sigma.$$

Generalizing, at an arbitrary point ξ, we arrive at the continuous integral form of the equidistribution principle

$$\int_a^{x(\xi)} \rho(x)dx = \xi\sigma, \quad \xi \in [0,1]. \tag{2.2}$$

Differentiating (2.2) with respect to ξ we have

$$\rho(x)\frac{dx}{d\xi} = \sigma, \tag{2.3}$$

and differentiating a second time gives

$$\frac{d}{d\xi}\left(\rho(x)\frac{dx}{d\xi}\right) = 0. \tag{2.4}$$

Requiring mesh nodes at $x = a$ and $x = b$ gives the boundary conditions

$$x(0) = a, \; x(1) = b. \tag{2.5}$$

It is possible to write the equidistribution problem in terms of a variational problem: Given $\rho(x)$ find $x = x(\xi)$ such that (2.5) is satisfied so that $x(\xi)$ minimizes the functional

$$I[x] = \frac{1}{2}\int_0^1 \left(\rho(x)\frac{dx}{d\xi}\right)^2 d\xi. \tag{2.6}$$

The Euler–Lagrange equation of (2.6) will be (2.4) as shown in [20].

The choice of the mesh density function is a separate question. The function ρ controls the distribution of mesh points. It has large values where the desired concentration of the mesh points should be higher, usually due to larger numerical error for the existing mesh.

Common choices for the mesh density function are

$$\rho \sim \sqrt{1 + |u'|^2}, \tag{2.7}$$

the arc-length mesh density function, and

$$\rho \sim \sqrt[4]{1 + |u''|^2}, \tag{2.8}$$

the curvature mesh density function, where u is the approximated solution of a given PDE. Also, some error estimator can be used to estimate the error of the solution and, thus, construct ρ respectively. More details about constructing the mesh density function can be found in [20].

Generally speaking, the best choice of ρ is problem dependent. It depends on what one regards as the "best" mesh and may change with factors such as used interpolation method or error norm.

2.1.2 Equidistribution quality measure

Using equation (2.3) the authors of [20] suggest to use the function

$$Q(x) = \frac{\rho x_\xi}{\sigma}$$

as an equidistribution quality measure. For the discrete case it can be written as

$$Q_j = \frac{\rho(x_j) + \rho(x_{j-1})}{2} \, \frac{x_j - x_{j-1}}{\sigma_h},$$

where

$$\sigma_h = \sum_{j=2}^{N} (x_j - x_{j-1}) \frac{\rho(x_j) + \rho(x_{j-1})}{2}.$$

It is easy to see that $\max\limits_j Q_j \geq 1$ and $\max\limits_j Q_j = 1$ if and only if $x_j,\ j = 1, \ldots, N$, is an equidistributing mesh for $\rho(x)$.

2.1.3 Solution methods

Discretizing (2.4) using central finite differences we have

$$\begin{aligned}\frac{d}{d\xi}\Big(\rho(x)\frac{dx}{d\xi}\Big)\Big|_{x=x_j} &\approx \frac{1}{\xi_{j+\frac{1}{2}}-\xi_{j-\frac{1}{2}}}\Big(\rho(x_{j+\frac{1}{2}})\frac{x_{j+1}-x_j}{\xi_{j+1}-\xi_j}-\rho(x_{j-\frac{1}{2}})\frac{x_j-x_{j-1}}{\xi_j-\xi_{j-1}}\Big)\\ &= \frac{2}{\xi_{j+1}-\xi_{j-1}}\Big(\frac{\rho(x_{j+1})+\rho(x_j)}{2}\frac{x_{j+1}-x_j}{\xi_{j+1}-\xi_j}-\frac{\rho(x_j)+\rho(x_{j-1})}{2}\frac{x_j-x_{j-1}}{\xi_j-\xi_{j-1}}\Big).\end{aligned}$$

Thus, approximating (2.4) and (2.5) gives the system of nonlinear algebraic equations

$$\frac{2}{\xi_{j+1}-\xi_{j-1}}\Big(\frac{\rho(x_{j+1})+\rho(x_j)}{2}\,\frac{x_{j+1}-x_j}{\xi_{j+1}-\xi_j}-\frac{\rho(x_j)+\rho(x_{j-1})}{2}\,\frac{x_j-x_{j-1}}{\xi_j-\xi_{j-1}}\Big)=0, \qquad j=2,\dots,N-1, \tag{2.9}$$

(2.10)

with $x_1=a$ and $x_N=b$.

Newton's method approach

One approach to solve (2.9) is to consider it as a system of nonlinear algebraic equations, which we can solve by applying Newton's method. Let $x=(x_1,x_2,\dots,x_N)^T$ be the solution and $x^{(n)}$ be the n^{th} Newton approximation. Newton's method gives the sequence of approximations

$$x^{(n+1)}=x^{(n)}-(J_f^{(n)})^{-1}f^{(n)}.$$

Letting $f^{(n)}=f(x^{(n)})$, we have

$$\begin{aligned}f_j^{(n)}=\frac{2}{\xi_{j+1}-\xi_{j-1}}\Big(&\frac{\rho(x_{j+1}^{(n)})+\rho(x_j^{(n)})}{2}\,\frac{x_{j+1}^{(n)}-x_j^{(n)}}{\xi_{j+1}-\xi_j}\\ &-\frac{\rho(x_j^{(n)})+\rho(x_{j-1}^{(n)})}{2}\,\frac{x_j^{(n)}-x_{j-1}^{(n)}}{\xi_j-\xi_{j-1}}\Big),\quad j=2,\dots,N-1.\end{aligned}$$

The Jacobian is a tridiagonal matrix with entries in the j-th row given by

$$J_f^{(n)}(j,j-1) = \frac{1}{\xi_{j+1}-\xi_{j-1}}\,\frac{1}{\xi_j-\xi_{j-1}}\Big((\rho(x_j^{(n)})+\rho(x_{j-1}^{(n)})) - \rho'(x_{j-1}^{(n)})\,(x_j^{(n)}-x_{j-1}^{(n)})\Big),$$

$$J_f^{(n)}(j,j) = \frac{1}{\xi_{j+1}-\xi_{j-1}}\Big(\rho'(x_j^{(n)})\,\Big(\frac{x_{j+1}^{(n)}-x_j^{(n)}}{\xi_{j+1}-\xi_j} - \frac{x_j^{(n)}-x_{j-1}^{(n)}}{\xi_j-\xi_{j-1}}\Big) - \frac{\rho(x_{j+1}^{(n)})+\rho(x_j^{(n)})}{\xi_{j+1}-\xi_j} - \frac{\rho(x_j^{(n)})+\rho(x_{j-1}^{(n)})}{\xi_j-\xi_{j-1}}\Big),$$

$$J_f^{(n)}(j,j+1) = \frac{1}{\xi_{j+1}-\xi_{j-1}}\,\frac{1}{\xi_{j+1}-\xi_j}\Big((\rho(x_{j+1}^{(n)})+\rho(x_j^{(n)})) + \rho'(x_{j+1}^{(n)})\,(x_{j+1}^{(n)}-x_j^{(n)})\Big),$$

$$j = 2,\dots,N-1,$$

with $x_1 = a$ and $x_N = b$ remain fixed.

Sequential nodes can swap their positions after an iteration of Newton's method. After that the method can start to diverge. Numerical experiments in Section 2.1.4 show that reordering nodes in an increasing order to prevent nodes from swapping significantly improves convergence and convergence rate of the method. This still needs a theoretical proof or justification.

Damped Newton's method

Another way to avoid nodes swapping during Newton's method is to apply a damping at each iteration. Setting $w^{(n)} = (J_f^{(n)})^{-1} f^{(n)}$, the damped Newton's method updates the solution as

$$x^{(n+1)} = x^{(n)} - k^{(n)} w^{(n)},$$

where $k^{(n)}$ is a damping scalar with $0 < k^{(n)} \le 1$. If we put constraints on $x^{(n)}$ to avoid swapping:

$$x_{i+1}^{(n)} - x_i^{(n)} \ge \epsilon, \tag{2.11}$$

where $\epsilon > 0$ is some small positive constant, and require the same for $x^{(n+1)}$, we will get the system of inequalities (taking into account that $x_1^{(n+1)} = x_1^{(n)} = a$, $x_N^{(n+1)} = x_N^{(n)} = b$)

$$\begin{cases} x_2^{(n)} - k^{(n)} w_2^{(n)} - a & \ge \epsilon, \\ (x_{i+1}^{(n)} - k^{(n)} w_{i+1}^{(n)}) - (x_i^{(n)} - k^{(n)} w_i^{(n)}) & \ge \epsilon, \quad i = 2,\dots,N-2, \\ b - (x_{N-1}^{(n)} - k^{(n)} w_{N-1}^{(n)}) & \ge \epsilon. \end{cases}$$

This system can be rewritten as

$$\begin{cases} k^{(n)}(w_2^{(n)} - 0) & \leq & x_2^{(n)} - a - \epsilon, \\ k^{(n)}(w_{i+1}^{(n)} - w_i^{(n)}) & \leq & x_{i+1}^{(n)} - x_i^{(n)} - \epsilon, \quad i = 2, \ldots, N-2, \\ k^{(n)}(0 - w_{N-1}^{(n)}) & \leq & b - x_{N-1}^{(n)} - \epsilon. \end{cases}$$

As $x_1^{(n+1)} = x_1^{(n)} = a$, $x_N^{(n+1)} = x_N^{(n)} = b$, we have $w_1^{(n)} = w_N^{(n)} = 0$. If we define

$$\Delta w^{(n)} : \ \Delta w_i^{(n)} = w_{i+1}^{(n)} - w_i^{(n)}, \quad i = 1, \ldots, N-1,$$

$$\Delta x^{(n)} : \ \Delta x_i^{(n)} = x_{i+1}^{(n)} - x_i^{(n)}, \quad i = 1, \ldots, N-1,$$

then we have

$$k^{(n)} \Delta w_i^{(n)} \leq \Delta x_i^{(n)} - \epsilon, \quad i = 1, \ldots, N-1. \tag{2.12}$$

Assume now the constraint (2.11) is satisfied for $x^{(n)}$. Then the right hand side of (2.12) is non-negative, also $k^{(n)}$ is non-negative. So, (2.12) is satisfied for any $k^{(n)}$ when $\Delta w_i^{(n)} < 0$. Take $\Delta \tilde{w}^{(n)}$ as the subvector of $\Delta w^{(n)}$ with all positive elements. Use the same subset of indices to form $\Delta \tilde{x}^{(n)}$ from $\Delta x^{(n)}$. Then the nodes will not tangle if

$$k^{(n)} \leq \min \left(\min_i \frac{\Delta \tilde{x}_i^{(n)} - \epsilon}{\Delta \tilde{w}_i^{(n)}}, 1 \right). \tag{2.13}$$

Hence we arrive at the following theorem.

Theorem 2.1.1. *If the mesh $\{x^{(n)_i}\}_{i=1}^N$ satisfies (2.11) and $w^{(n)}$ is the step for a Newton's method iteration then the biggest step that prevents mesh nodes from swapping is $k^{(n)} w^{(n)}$, where $k^{(n)}$ is a maximum possible value that satisfies (2.13).*

Linearized boundary value problem method

Another way to work with the scheme (2.9) is the boundary value problem (BVP) method suggested in [20]. The idea of this method is to use an iterative procedure where values of $\rho(x)$ are obtained from the previous step, and new values of x are calculated from the discretization. Therefore, the scheme is: Given $x_j^{(0)}$, $j = 1, ..., N$,

for $n = 1, 2, 3, \cdots$ solve

$$\frac{2}{\xi_{j+1} - \xi_{j-1}} \Big(\frac{\rho(x_{j+1}^{(n)}) + \rho(x_j^{(n)})}{2} \, \frac{x_{j+1}^{(n+1)} - x_j^{(n+1)}}{\xi_{j+1} - \xi_j} - \frac{\rho(x_j^{(n)}) + \rho(x_{j-1}^{(n)})}{2} \, \frac{x_j^{(n+1)} - x_{j-1}^{(n+1)}}{\xi_j - \xi_{j-1}} \Big) = 0, \quad j = 2, \ldots, N-1, \tag{2.14}$$

$x_1 = a, \; x_N = b,$

until a stopping criteria is reached. Note that a linear system of equations solved at each iteration. As a stopping criteria we can choose a maximum number of iterations or the norm of the difference between consecutive meshes.

De Boor's method

The idea of De Boor's method [11] is to approximate the monitor function $\rho(x)$ with a piecewise constant (or piecewise linear) function $p(x)$ using the current mesh $\{x_j\}_{j=1}^N : a = x_1 < x_2 < \ldots < x_N = b$.

The piecewise constant choice is

$$p(x) = \frac{\rho(x_j) + \rho(x_{j+1})}{2}, \tag{2.15}$$

if $x \in (x_j, \, x_{j+1}]$, with $p(a) = (\rho(x_1) + \rho(x_2))/2$.

The piecewise linear choice is

$$p(x) = \rho(x_j) + (\rho(x_{j+1}) - \rho(x_j)) \, \frac{x - x_j}{x_{j+1} - x_j}, \tag{2.16}$$

if $x \in [x_j, \, x_{j+1}]$. Notice that for this case $p(x_j) = \rho(x_j), \quad j = 1, \ldots, N$.

As $\rho(x) > 0$ for all $x \in [a, \, b]$, it is straightforward to see that $p(x) > 0$ for all $x \in [a, \, b]$ for both (2.15) and (2.16).

Now it is easy to find an equidistributing mesh $\{y_k\}_{k=1}^N$ for the piecewise function $p(x)$. Setting

$$P(x) = \int_a^x p(x) dx, \tag{2.17}$$

we require $\{y_k\}_{k=2}^{N-1}$ to satisfy

$$P(y_k) = \xi_k P(b),$$

when $y_1 = a$ and $y_N = b$.

One notes that

$$P(x_j) = \sum_{i=1}^{j-1} (x_{i+1} - x_i) \frac{\rho(x_i) + \rho(x_{i+1})}{2}, \quad j = 1, \ldots, N, \tag{2.18}$$

for both piecewise constant and piecewise linear approximations as we sum the areas of either rectangles or trapezoids respectively. Since $p(x) > 0$ for all $x \in [a, b]$, and $P(x)$ is its integral, then $P(x)$ is monotonically increasing.

In the case of the piecewise constant approximation, for $x \in (x_j,\ x_{j+1})$ we have

$$P(x) = (x - x_j) p(x_{j+1}) + P(x_j).$$

If $P(x_j) < \xi_k P(b) \leq P(x_{j+1})$, then $y_k \in (x_j,\ x_{j+1}]$, so

$$\xi_k P(b) = P(y_k) = (y_k - x_j) p(x_{j+1}) + P(x_j),$$

or

$$y_k = x_j + \frac{\xi_k P(b) - P(x_j)}{p(x_{j+1})}.$$

In the case of piecewise linear approximation, for $x \in (x_j,\ x_{j+1})$ we have

$$P(x) = (x - x_j) \frac{p(x_j) + p(x)}{2} + P(x_j).$$

Again, if $P(x_j) < \xi_k P(b) \leq P(x_{j+1})$, then $y_k \in (x_j,\ x_{j+1}]$, so

$$\begin{aligned} \xi_k P(b) = P(y_k) &= (y_k - x_j) \frac{p(y_k) + p(x_j)}{2} + P(x_j) = \\ &= (y_k - x_j) \Big(\rho(x_j) + \frac{\rho(x_{j+1}) - \rho(x_j)}{2} \, \frac{y_k - x_j}{x_{j+1} - x_j} \Big) + P(x_j), \end{aligned}$$

or

$$\frac{1}{2} \, \frac{\rho(x_{j+1}) - \rho(x_j)}{x_{j+1} - x_j} (y_k - x_j)^2 + \rho(x_j)(y_k - x_j) + P(x_j) - \xi_k P(b) = 0.$$

Thus we get a quadratic equation for $(y_k - x_j)$. As $y_k \in (x_j,\ x_{j+1}]$ we are interested only in the solutions from the interval $(0, x_{j+1} - x_j]$. It is possible to show that this equation has one and only one such solution if $\rho(x_{j+1}) > \rho(x_j)$ or if $\rho(x_{j+1}) < \rho(x_j)$. It is also obvious from the geometrical point of view. $P(x)$ is a monotonic function that increases from 0 to $P(b)$ on $[a,\ b]$, so there exists a unique point $y_k \in [a,\ b]$ where $P(y_k) = P(b)\ (k-1)/(N-1)$, $k = 1, \ldots, N$.

After obtaining a new mesh $\{y_k\}_{k=1}^N$ we can use it again to find a new approximation of the monitor function and the iteration continues.

In [29], Pryce shows that this iterative procedure converges to the equidistributing mesh if $p(x)$ is constructed as a piecewise linear interpolation for smooth enough $\rho(x)$ and a sufficient number of mesh points. In [35], Xu et al. shows the same for $p(x)$ constructed as a piecewise constant interpolation.

Inverse boundary value problem method (IBVP)

An alternative way to find an equidistributing mesh is to look for the inverse coordinate transformation $\xi(x) : [a, b] \to [0, 1]$. Then (2.2) can be rewritten as

$$\xi(x) = \frac{1}{\sigma} \int_a^x \rho(t)dt,\ \xi \in [0, 1]. \tag{2.19}$$

Since $\rho(x) > 0$ for all $x \in [a,\ b]$, then from (2.3) and (2.19) we can see that both $x(\xi)$ and $\xi(x)$ are monotonically increasing. Differentiating (2.3) with respect to x twice we derive the differential equation

$$\frac{d}{dx}\Big(\frac{1}{\rho(x)}\frac{d\xi}{dx}\Big) = 0 \tag{2.20}$$

with boundary conditions

$$\xi(a) = 0,\ \xi(b) = 1. \tag{2.21}$$

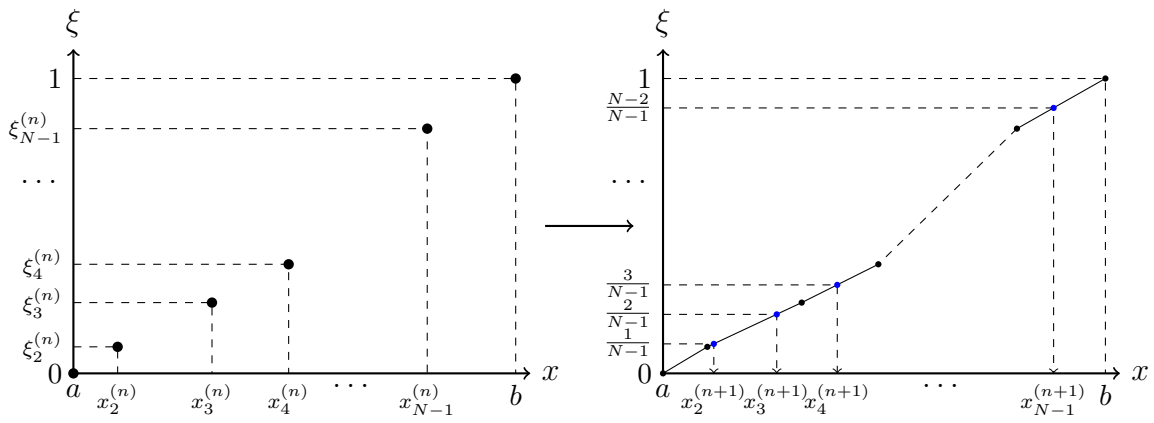

Figure 2.1: Illustration of the piecewise linear interpolation in the IBVP method.

Discretizing it the same way as in the BVP method above we obtain the system

$$\frac{2}{x_{j+1}-x_{j-1}}\left(\frac{2}{\rho(x_{j+1})+\rho(x_j)}\,\frac{\xi_{j+1}-\xi_j}{x_{j+1}-x_j}-\frac{2}{\rho(x_j)+\rho(x_{j-1})}\,\frac{\xi_j-\xi_{j-1}}{x_j-x_{j-1}}\right)=0,\qquad j=2,\ldots,N-1,\tag{2.22}$$

$\xi_1 = 0,\ \xi_N = 1.$

For a given mesh $\{x_j\}$ this is a linear system of equations for $\xi_j \approx \xi(x_j)$, $j = 1, \ldots, N$. As shown later in Section 2.1.3, $\xi_j = P(x_j)/P(b)$ is the solution of this system, where $P(x_j)$ is taken from the equation (2.18). $P(x)$ is a monotonic function for both piecewise constant and piecewise linear interpolation cases, so $\{P(x_j)/P(b)\}_{j=1}^N$ is also monotonic. Since $x(\xi)$ is monotonic, we can write the points as images of either $x(\xi)$ or $\xi(x)$. So, the solution of the system (2.22) can be interpolated to obtain a new mesh $\{\hat{x}_i\}_{i=1}^N$ from $\{\hat{\xi}_i = (i-1)/N\}_{i=1}^N$. Note that the interpolation method should also preserve the monotonicity of the function $x(\xi)$, $\xi \in [0, 1]$. An example using a piecewise linear interpolation can be found in Fig. 2.1.

This completes one iteration of the inverse value boundary problem (IBVP) method. Problem (2.20) can then be solved using the newly obtained physical mesh $\{\hat{x}_j\}$, and so on, resulting in an iterative approach.

Connections between IBVP and De Boor's methods

We numerically observe that the results obtained for the IBVP method using piecewise linear interpolation are to machine-precision identical to the ones obtained from De Boor's algorithm with piecewise constant interpolation.

We now show why this is the case.

Theorem 2.1.2. *De Boor's method with $p(x)$ a piecewise constant approximation to $\rho(x)$ (2.15) is equivalent to the IBVP method (2.22) followed by a piecewise linear interpolation at each iteration.*

Proof. First we will show that $\xi(x_j) = P(x_j)/P(b)$ satisfies (2.22) for $P(x)$ described in (2.15) and (2.17).

On each interval the area under the piecewise constant function $p(x)$ is the area of a rectangle. Hence we have

$$P(x_j) = \sum_{i=1}^{j-1} (x_{i+1} - x_i) p(x_{i+1}).$$

Therefore,

$$P(x_{j+1}) - P(x_j) = (x_{j+1} - x_j) p(x_{j+1}) = (x_{j+1} - x_j) \frac{\rho(x_{j+1}) + \rho(x_j)}{2}.$$

Substituting $\xi_j = cP(x_j)$ into the system (2.22), we see that it is satisfied for all internal points, for all $c \in \mathbb{R}$. To satisfy the boundary condition $\xi(b) = 1$, we need $c = 1/P(b)$.

Note that this is reasonable, as ξ was originally designed to satisfy (2.19), where

$$\sigma = \int_a^b \rho(t) dt,$$

and

$$P(x) \approx \int_a^x \rho(t) dt.$$

In the IBVP method, after obtaining $\{\xi_j\}_{j=1}^N$ we use a piecewise linear interpolation of $\xi(x)$ to find a new $\{x_j\}_{j=1}^N$, so that $\xi(x_j) = (j-1)/(N-1)$. But in De Boor's

method we also look for $\{y_j\}_{j=1}^N$, such that $P(y_j)/P(b) = (j-1)/(N-1)$, where $P(x)$ is a piecewise linear function since it is the integral of a piecewise constant function. ■

A similar statement can be proved for De Boor's method with a piecewise linear approximation to $\rho(x)$. But the integral of a piecewise linear function will be a piecewise quadratic function. We need three points to specify a quadratic function. As an additional point we will use the value at the midpoint of the interval. The value at this point will be set to

$$\xi\Big(\frac{x_{i+1}+x_i}{2}\Big) = \xi(x_i) + \frac{x_{i+1}-x_i}{2}\,\frac{\rho(x_{i+1})+3\rho(x_i)}{4}\,\frac{1}{\hat{\sigma}}, \tag{2.23}$$

where

$$\hat{\sigma} = \sum_{j=1}^{N-1}(x_{j+1}-x_j)\frac{\rho(x_{j+1})+\rho(x_j)}{2}. \tag{2.24}$$

The reason for this value will be shown in the next theorem.

Theorem 2.1.3. *De Boor's method with $p(x)$ a piecewise linear approximation to $\rho(x)$ (2.16) is equivalent to the IBVP method (2.22) followed by a piecewise quadratic approximation at each iteration with the additional interpolation point in the middle of each interval given by (2.23) and (2.24).*

Proof. Again, as in the previous case, first we will show that $\xi(x_j) = P(x_j)/P(b)$ satisfies (2.22) for $P(x)$ described in (2.16) and (2.17).

On each interval, the area under the piecewise linear function $p(x)$ is the area of a trapezoid. Hence we have

$$P(x_j) = \sum_{i=1}^{j-1}(x_{i+1}-x_i)\frac{p(x_{i+1})+p(x_i)}{2},$$

where $p(x_i) = \rho(x_i)$.

So,

$$P(x_{j+1}) - P(x_j) = (x_{j+1}-x_j)\frac{\rho(x_{j+1})+\rho(x_j)}{2}.$$

And again, substituting $\xi_j = cP(x_j)$ into the system (2.22), we see that it is satisfied for all internal points for all $c \in \mathbb{R}$. To satisfy the boundary condition $\xi(b) = 1$, we

again need $c = 1/P(b)$.

Now for $x \in (x_j,\, x_{j+1})$,

$$p(x) = \rho(x_j) + (\rho(x_{j+1}) - \rho(x_j))\frac{x - x_j}{x_{j+1} - x_j},$$

and

$$P(x) = P(x_j) + \frac{p(x_j) + p(x)}{2}(x - x_j).$$

And so $P(x)$ is a piecewise quadratic function.

If $x = (x_{j+1} + x_j)/2$ then

$$p(x) = \frac{\rho(x_{j+1}) + \rho(x_j)}{2}$$

and

$$P(x) = P(x_j) + \frac{\rho(x_{i+1}) + 3\rho(x_i)}{4}\,\frac{x_{i+1} - x_i}{2}.$$

Also we notice that $\hat{\sigma} = P(b)$. Hence, $\xi(x_j) = P(x_j)/P(b)$ satisfies (2.22), the boundary conditions $\xi(a) = 0$, $\xi(b) = 1$, and the midpoint value (2.23) for the quadratic interpolation satisfies

$$\xi\Big(\frac{x_{i+1} + x_i}{2}\Big) = P\Big(\frac{x_{i+1} + x_i}{2}\Big)/P(b).$$

So, we have two piecewise quadratic functions on the same subintervals, that have the same values at three points of each subinterval. As a quadratic function is uniquely defined by its values at three points, we have the same interpolation functions for both methods. Another observation is that $P(x)$ is monotonic and $P(x)/P(b)$ is the function that we use for the quadratic interpolation (2.23), hence the interpolation function is monotonic and legitimate to use. And in both methods we look for the points $\{y_j\}_{j=1}^{N}$, such that $P(y_j)/P(b) = (j-1)/(N-1)$. So, the new meshes we obtain are also the same. ■

Note. *While the results should be the same for both methods in exact arithmetic, in the IBVP method we are solving a linear system to get the values for* $\{\xi_j\}_{j=2}^{N-1}$*, so, the accuracy of this numerical method will be generally lower.*

2.1.4 Results

To illustrate the methods above, we consider as a model problem the mesh density function

$$\rho(x) = 1 + R(1 - \tanh^2(R(x - 0.5))).$$

As stopping criteria we choose that equidistribution quality measure is less then $1 + 10^{-10}$, or that the number of iterations exceeds 1000.

R	BVP method			De Boor's method			Pure Newton method								
							0 pre-steps			1 pre-step			5 pre-steps		
	Number of nodes			Number of nodes			Number of nodes			Number of nodes			Number of nodes		
	21	41	81	21	41	81	21	41	81	21	41	81	21	41	81
10	37	39	40	14	11	10	10	13	18	7	7	7	4	4	4
20	NC	58	61	20	14	12	29	106	257	24	89	77	5	6	6
50	NC	NC	144	19	18	16	36	513	401	13	307	NC	7	289	319

R	Newton method with reordering									Damped Newton		
	0 pre-steps			1 pre-step			5 pre-steps					
	Number of nodes			Number of nodes			Number of nodes			Number of nodes		
	21	41	81	21	41	81	21	41	81	21	41	81
10	7	7	6	7	7	7	4	4	4	7	7	7
20	10	9	10	8	8	8	5	6	6	9	10	9
50	12	13	16	10	12	14	7	9	8	12	12	15

Table 2.1: Number of steps until convergence for different 1D mesh adaptation methods. "NC" indicates that the algorithm did not converge.

As we can see from Table 2.1 the linearized BVP method converges relatively slowly and needs enough nodes to converge. Newton's method converges faster but also needs enough nodes to see convergence. Also, Newton's method can be improved by using a better initial mesh. Taking a couple of steps using the linearized BVP method (pre-steps for Newton's method) before using Newton's method gives much better results but may still lead to situations where it does not converge.

As mentioned earlier Newton's method can be improved by sorting the approximate mesh or solution at each step. This significantly improves stability and greatly speeds up convergence in some cases. We can notice, that using pre-steps does not improve the method with reordering as much as the original Newton's method. The complexity of sorting is $O(Nlog(N))$ while a tridiagonal system can be solved in $O(N)$ operations, so, in the cases of big N, reordering can slow down each iteration.

De Boor's method is more reliable and converges faster in the situations where Newton's method has troubles, but it works slower in cases when Newton's method works well. Results for the IBVP method were not provided because they are the

same as for De Boor's method.

2.2 Mesh generation methods in 2D

2.2.1 Algebraic methods

For numerical methods, such as finite differences, the most natural mesh to solve a partial differential equation is a rectangular mesh. But in the case of a complex geometry finite differences may not be the best choice. Thus, a coordinate transformation can be applied to transform the problem's domain into a simpler shape, usually rectangular. The original space is called the physical space, while a newly obtained one is called the computational space. Hereafter we will denote (x, y) as the coordinates for the physical space and (ξ, η) as the coordinates for the computational space.

In some cases, like an elliptical domain, a transformation to another common coordinate system, like ellipsoidal coordinates, can be used. In more complicated geometries other techniques should be used.

Consider two points r^0 and r^1 on the opposite boundaries in the computational space with a known physical coordinates. Unidirectional interpolation connects these point by a line

$$r(\xi) = r^0 + \xi(r^1 - r^0), \quad 0 \leq \xi \leq 1. \tag{2.25}$$

Grid points can be uniformly distributed across this line or have a different layout according to some other reasoning. After placing the points in the computational space we can use a linear interpolation the get their physical coordinates. Note that the unidirectional interpolation transforms this straight line in a computational space into a straight line in the physical space. Unidirectional interpolation can be used if we have two curves representing the opposite boundaries of the domain, and we want to create a mesh matching the points on these curves. Another method that can be used here is the multisurface method introdused by Eiseman [14]. This method allows control of the grid distribution and angles for the generated mesh.

As an example suppose we have a figure $ABCD$ formed by four segments of different curves (see Figure 2.2) and a coordinate transformation $r(\xi, \eta)$ such that the

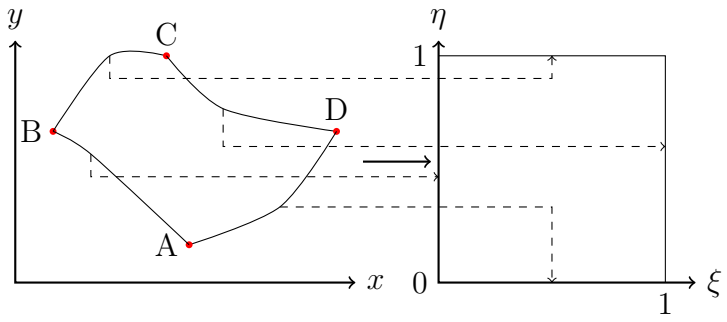

Figure 2.2: Mapping curved four-sided figure into a unit square.

segments $[(0,0),(0,1)]$, $[(1,0),(1,1)]$, $[(1,1),(1,0)]$ and $[(1,0),(0,0)]$ in the computational space are mapped to the curved segments $[A,B]$, $[B,C]$, $[C,D]$ and $[D,A]$ respectively in the physical space. If we try to use unidirectional interpolation for $[(0,0),(0,1)]$ and $[(1,1),(1,0)]$ segments, then we will get straight line segments for $[B,C]$ and $[D,A]$ instead of curves. If we try to use unidirectional interpolation for $[(1,0),(1,1)]$ and $[(1,0),(0,0)]$ segments, then the same will happen to $[A,B]$ and $[C,D]$ segments. For such cases the transfinite interpolation (TFI) method was developed [19].

If we denote $P_\xi(\xi,\eta)$ as a unidirectional interpolation across the $\xi=0$ and $\xi=1$ borders, $P_\eta(\xi,\eta)$ as a unidirectional interpolation across $\eta=0$ and $\eta=1$ borders and $P_{\xi\eta}(\xi,\eta)$ as their tensor product, then TFI can be written as

$$(P_\xi \oplus P_\eta)(\xi,\eta) = P_\xi(\xi,\eta) + P_\eta(\xi,\eta) - P_{\xi\eta}(\xi,\eta). \tag{2.26}$$

The TFI method is fast and allows the construction of meshes for many domain geometries, though not for all.

The initial grid point distribution on the boundaries, that is subsequently used by the unidirectional interpolation and the TFI method, can be accompanied by various interpolation methods or the hybrid curve point distribution algorithm [32].

There are drawbacks of algebraic grid generation methods such as the unidirectional interpolation and the TFI method. First of all, it usually increases the complexity of the PDE due to the coordinate transformation. Second, it does not produce the smooth meshes, and the discontinuities of the gradient of the solution on the boundary will propagate into the interior of the domain [16]. Third, the resulting mesh is

not guaranteed to be orthogonal and can fold.

2.2.2 Differential equation methods

The quality of the numerical solution depends on properties of the mesh. So, it is important to generate the mesh that will provide a better solution. To overcome the problems occurring with the algebraic grids, the differential grid generation methods were introduced.

If we return to the problem illustrated by the Figure 2.2, then the direct problem will be: find smooth enough functions $\xi(x, y)$ and $\eta(x, y)$ that provide the required mapping.

In general, there will be an infinite number of solutions for this problem. To specify one of them we can require $\xi(x, y)$ and $\eta(x, y)$ to satisfy some PDE. If we choose a second-order PDE, as we wish to satisfy Dirichlet boundary conditions, we could choose an elliptic equation. Usually Laplace's equation

$$\begin{aligned} \nabla^2 \xi &= 0, \\ \nabla^2 \eta &= 0, \end{aligned} \tag{2.27}$$

is used for this purpose.

In practice, instead of the direct problem (2.27), the inverse problem is solved: find functions $x(\xi, \eta)$ and $y(\xi, \eta)$ that satisfy the required mapping and with the values set on the boundaries of the unit square. Then (2.27) is transformed to

$$\begin{aligned} g_{22}\frac{\partial^2 x}{\partial \xi^2} - 2g_{12}\frac{\partial^2 x}{\partial \xi \partial \eta} + g_{11}\frac{\partial^2 x}{\partial \eta^2} &= 0, \\ g_{22}\frac{\partial^2 y}{\partial \xi^2} - 2g_{12}\frac{\partial^2 y}{\partial \xi \partial \eta} + g_{11}\frac{\partial^2 y}{\partial \eta^2} &= 0, \end{aligned} \tag{2.28}$$

where $g_{11} = x_\xi^2 + y_\xi^2$, $g_{22} = x_\eta^2 + y_\eta^2$ and $g_{12} = x_\xi x_\eta + y_\xi y_\eta$, see [16] for details. Unlike (2.27), (2.28) is not linear and decoupled any more, but it is elliptic. Hence, the grid generated by it is smooth.

For some problems one may want to alter the distribution of the grid points in the domain. With (2.28) we can control only the distribution of the grid points on the boundaries. Thus, to control the interior distribution, we can modify (2.27) by

solving for ξ and η which satisfy

$$\begin{aligned} \nabla^2 \xi &= P(\xi, \eta), \\ \nabla^2 \eta &= Q(\xi, \eta), \end{aligned} \tag{2.29}$$

where $P(\xi, \eta)$ and $Q(\xi, \eta)$ are so-called control functions [33]. The corresponding changes in the inverse problem, though, can lead to loosing some degrees of smoothness of the resulting mesh [16].

2.2.3 Variational methods

The quality of the grid obtained by the differential equation methods can still vary depending on differential equation used. To study this problem, variational approaches of the grid generation have been developed.

Different functionals can control such properties as the node distribution, the cell area and the grid's orthogonality. Their weighted combinations can control these properties simultaneously.

The Euler–Lagrange equation for the L(length)-functional

$$I_L = \frac{1}{2} \iint_{\Omega_c} \left((x_\xi)^2 + (x_\eta)^2 + (y_\xi)^2 + (y_\eta)^2 \right) d\xi d\eta, \tag{2.30}$$

where Ω_c is the computational domain, are

$$\begin{aligned} \frac{\partial^2 x}{\partial \xi^2} + \frac{\partial^2 x}{\partial \eta^2} &= 0, \\ \frac{\partial^2 y}{\partial \xi^2} + \frac{\partial^2 y}{\partial \eta^2} &= 0, \end{aligned} \tag{2.31}$$

If we use the opposite functional

$$I = \frac{1}{2} \iint_{\Omega_p} \left[\left(\frac{\partial \xi}{\partial x} \right)^2 + \left(\frac{\partial \xi}{\partial y} \right)^2 + \left(\frac{\partial \eta}{\partial x} \right)^2 + \left(\frac{\partial \eta}{\partial y} \right)^2 \right] dx dy, \tag{2.32}$$

where Ω_p is the physical domain, the Euler–Lagrange equations for it are (2.27), the Winslow model.

If we incorporate the weight function $M(x, y)$ with the functional (2.32) in the

way

$$I_W = \frac{1}{2}\iint_{\Omega_p} \frac{1}{M(x,y)}\left[\left(\frac{\partial\xi}{\partial x}\right)^2 + \left(\frac{\partial\xi}{\partial y}\right)^2 + \left(\frac{\partial\eta}{\partial x}\right)^2 + \left(\frac{\partial\eta}{\partial y}\right)^2\right] dxdy, \tag{2.33}$$

the Euler–Lagrange equations for this functional are (2.38) and (2.39) discussed in Section 2.2.4.

The weighted A(area)-functional has the form

$$I_A = \frac{1}{2}\iint_{\Omega_c} \left(\frac{(x_\xi y_\eta - x_\eta y_\xi)^2}{\phi(\xi,\eta)}\right) d\xi d\eta, \tag{2.34}$$

where $\phi(\xi,\eta)$ is a weight function. The Euler–Lagrange equations for this functional can be written as

$$\begin{aligned} \left(\frac{x_\xi y_\eta - x_\eta y_\xi}{\phi}\right)_\xi x_\eta + \left(\frac{x_\xi y_\eta - x_\eta y_\xi}{\phi}\right)_\eta x_\xi &= 0, \\ \left(\frac{x_\xi y_\eta - x_\eta y_\xi}{\phi}\right)_\xi y_\eta + \left(\frac{x_\xi y_\eta - x_\eta y_\xi}{\phi}\right)_\eta x_\xi &= 0. \end{aligned} \tag{2.35}$$

This functional tries to preserve the weighted area of grid cells.

An O(orthogonality)-functional is

$$I_O = \frac{1}{2}\iint_{\Omega_c} (x_\xi x_\eta + y_\xi y_\eta)^2 d\xi d\eta. \tag{2.36}$$

We can express the Euler–Lagrange equations for this functional as

$$\begin{aligned} \big((x_\xi x_\eta + y_\xi y_\eta)x_\eta\big)_\xi + \big((x_\xi x_\eta + y_\xi y_\eta)x_\xi\big)_\eta &= 0, \\ \big((x_\xi x_\eta + y_\xi y_\eta)y_\eta\big)_\xi + \big((x_\xi x_\eta + y_\xi y_\eta)y_\xi\big)_\eta &= 0. \end{aligned} \tag{2.37}$$

This is not a full list of possible functionals, see [16] for additional examples.

2.2.4 Winslow mesh generator

Now we introduce a two-dimensional mesh generation method developed by Winslow [34].

Let $\rho(x,y)$ be the 2D mesh density function. In this method we solve the following

system of two decoupled equations:

$$\frac{\partial}{\partial x}\Big(\frac{1}{\rho(x,y)}\frac{\partial \xi}{\partial x}\Big)+\frac{\partial}{\partial y}\Big(\frac{1}{\rho(x,y)}\frac{\partial \xi}{\partial y}\Big)=0, \tag{2.38}$$

$$\frac{\partial}{\partial x}\Big(\frac{1}{\rho(x,y)}\frac{\partial \eta}{\partial x}\Big)+\frac{\partial}{\partial y}\Big(\frac{1}{\rho(x,y)}\frac{\partial \eta}{\partial y}\Big)=0, \tag{2.39}$$

where (x, y) are the coordinates in the physical space and (ξ, η) are the coordinates in the computational space, with Dirichlet boundary conditions defining the mesh points' distribution on the boundary:

$$\xi(x_i, y_1)=\xi_{i,1},\ \xi(x_i, y_N)=\xi_{i,N},\ \xi(x_1, y_j)=0,\ \xi(x_M, y_j)=1, \tag{2.40}$$

$$\eta(x_1, y_j)=\eta_{1,j},\ \eta(x_M, y_j)=\eta_{M,j},\ \eta(x_i, y_1)=0,\ \eta(x_i, y_N)=1, \tag{2.41}$$

for a N by M mesh.

Equations (2.38) and (2.39) are the Euler–Lagrange equations for the functional (2.33) discussed in Section 2.2.3.

In the following, we will write $\xi_{i,j}=\xi(x_i, y_j)$ and $\rho_{i,j}=\rho(x_i, y_j)$. If an initial mesh in the physical space is a regular rectangular mesh, we can generalize the discretization given in (2.22). At an internal point (x_i, y_j) we approximate (2.38) by

$$\begin{aligned}&\frac{2}{x_{i+1}-x_{i-1}}\Big(\frac{2}{\rho_{i+1,j}+\rho_{i,j}}\frac{\xi_{i+1,j}-\xi_{i,j}}{x_{i+1}-x_i}-\frac{2}{\rho_{i,j}+\rho_{i-1,j}}\frac{\xi_{i,j}-\xi_{i-1,j}}{x_i-x_{i-1}}\Big)+\\ &+\frac{2}{y_{j+1}-y_{j-1}}\Big(\frac{2}{\rho_{i,j+1}+\rho_{i,j}}\frac{\xi_{i,j+1}-\xi_{i,j}}{y_{j+1}-y_j}-\frac{2}{\rho_{i,j}+\rho_{i,j-1}}\frac{\xi_{i,j}-\xi_{i,j-1}}{y_j-y_{j-1}}\Big)=0\end{aligned} \tag{2.42}$$

and approximate (2.39) by

$$\begin{aligned}&\frac{2}{x_{i+1}-x_{i-1}}\Big(\frac{2}{\rho_{i+1,j}+\rho_{i,j}}\frac{\eta_{i+1,j}-\eta_{i,j}}{x_{i+1}-x_i}-\frac{2}{\rho_{i,j}+\rho_{i-1,j}}\frac{\eta_{i,j}-\eta_{i-1,j}}{x_i-x_{i-1}}\Big)+\\ &+\frac{2}{y_{j+1}-y_{j-1}}\Big(\frac{2}{\rho_{i,j+1}+\rho_{i,j}}\frac{\eta_{i,j+1}-\eta_{i,j}}{y_{j+1}-y_j}-\frac{2}{\rho_{i,j}+\rho_{i,j-1}}\frac{\eta_{i,j}-\eta_{i,j-1}}{y_j-y_{j-1}}\Big)=0.\end{aligned} \tag{2.43}$$

To compute boundary conditions for the edges we can use the 1D methods, described before, if the domain has a rectangular shape.

We should note that if the initial mesh is not rectangular, another discretization should be chosen, such as meshless finite differences [27].

After solving (2.42)-(2.43) we get coordinates of the current physical mesh in the computational space. As in Section 2.1.3 we can consider the mapping between nodes as functions $x(\xi, \eta)$, $y(\xi, \eta)$ instead of $\xi(x, y)$, $\eta(x, y)$. Then, using interpolation, we can find a new mesh in the physical space, that corresponds to the uniform mesh in the computational domain.

Note that the mesh, obtained from solving (2.42)-(2.43) usually will not be rectangular, so, one may want to use some specific interpolation methods, like bilinear interpolation, radial basis functions, etc. to find the physical mesh location.

Also, if we would use a numerical scheme that does not require the initial mesh to be rectangular, we could make the method iterative by considering the newly obtained mesh as an initial mesh for the next step.

Chapter 3

Domain decomposition

In this chapter we introduce the idea of a domain decomposition method and its possible implementation with Monte Carlo techniques.

Using Winslow mesh generator requires to solving a linear system. For the 2D case, if the mesh size is $m \times n$ than the system will be $k \times k$, where $k = (m-1) \cdot (n-1)$. It will be a system with the bandwidth equal to $\min(m-1, n-1)$. When the size of the mesh grows, it becomes preferable to use a parallel implementation due to the local memory size and computational cost to solve the linear system. There are already existing methods that can split the matrix of the linear system between different processors and solve it in parallel, for example the parallel Gaussian elimination [26] or the MUMPS method [2, 3]. Unfortunately, most of these methods require a lot of communications between processors or it is difficult to balance the computational load.

Traditional domain decomposition methods split the domain into almost independent subdomains. Then the problem is iteratively solved on each subdomain with exchanging the boundary information between neighboring subdomains. Another approach to tackle the problem of domain decomposition is the stochastic domain decomposition approach. The idea is that we split the domain into independent subdomains and stochastically obtain the solution on the interfaces. Then we can solve a linear system for each subdomain independently. This will reduce both the communication cost and the cost of solving the linear system.

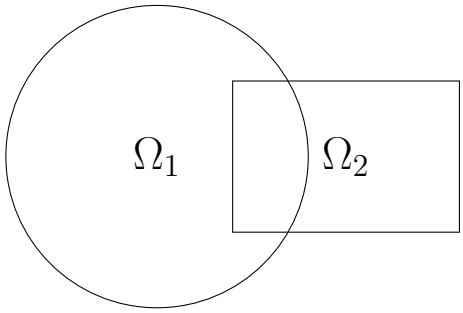

Figure 3.1: Classic example of the domain for the Schwarz method.

3.1 Overlapping domain decomposition methods

The original Schwarz method [31] was developed by Hermann Schwarz in the 19th century to solve a Poisson problem on domains with a complex geometry (that can be represented as a union of simple geometries).

To describe Schwarz method consider a problem of finding $u : \Omega \to \mathbb{R}$, where Ω is as in Fig. 3.1, such that

$$\begin{aligned} -\Delta u &= f, \quad \text{in } \Omega, \\ u &= 0, \quad \text{on } \partial\Omega. \end{aligned} \tag{3.1}$$

Suppose u^n is an approximation to the solution of (3.1) with u_1^n being u^n on Ω_1, u_2^n being u^n on Ω_2. Then a new approximation u^{n+1} can be obtained by solving

$$\begin{aligned} -\Delta(u_1^{n+1}) &= f, \quad \text{in } \Omega_1, \\ u_1^{n+1} &= 0, \quad \text{on } \partial\Omega_1 \cap \partial\Omega, \\ u_1^{n+1} &= u_2^n, \quad \text{on } \partial\Omega_1 \cap \Omega_2, \end{aligned} \tag{3.2}$$

first and then

$$\begin{aligned} -\Delta(u_2^{n+1}) &= f, \quad \text{in } \Omega_2, \\ u_2^{n+1} &= 0, \quad \text{on } \partial\Omega_2 \cap \partial\Omega, \\ u_2^{n+1} &= u_1^{n+1}, \quad \text{on } \partial\Omega_2 \cap \Omega_1. \end{aligned} \tag{3.3}$$

This method can be naturally extended to more than two subdomains, requiring only the newest available information for boundaries while solving the problem for each subdomain.

Used in this way, the method is still sequential, as we need updated information

from the previous subdomains to solve the problem on the next subdomain. The way to make it parallel consists of modifying (3.2)–(3.3) as

$$\begin{aligned} -\Delta(u_i^{n+1}) &= f, && \text{in } \Omega_i, \\ u_i^{n+1} &= 0, && \text{on } \partial\Omega_i \cap \partial\Omega, \\ u_i^{n+1} &= u_{3-i}^{n}, && \text{on } \partial\Omega_i \cap \Omega_{3-i}, \end{aligned} \tag{3.4}$$

for $i = 1, 2$. This way we can solve the problem on each subdomain independently and then only exchange the information about the solutions on the boundaries after an approximate solution has been found on all subdomains to perform the next iteration.

This approach can be extended to more than two subdomains, but then we need to decide which boundary information will be used.

While for the parallel approach variational arguments will not work to prove convergence, Lions in [25] shows that the convergence follows from the maximum principle technique introduced by Schwarz in [31].

Further development of the Schwarz method requires two more definitions, for details see [12].

Definition 3.1.1 (Extension operator)**.** An extension operator E_j from Ω_j to Ω is defined such that for a function $f_j : \Omega_j \to \mathbb{R}$, $E_j(f_j) : \Omega \to \mathbb{R}$ is the extension of f_j by 0 outside Ω_j for any $f \colon \Omega \to \mathbb{R}$.

Definition 3.1.2 (Partition of unity)**.** For $\Omega = \bigcup_{i=1}^{k} \Omega_i$, a set of functions $\{\chi_i\}_{i=1}^{k}$, $\chi_i : \Omega_i \to \mathbb{R}$ that satisfies:

$$\begin{aligned} &\chi_i \geq 0, \\ &\chi_i(x) = 0, && \text{for } x \in \partial\Omega_i \setminus \partial\Omega, \\ &f = \sum_{i=1}^{k} E_i(\chi_i \, f|_{\Omega_i}), && \text{for any } f : \Omega \to \mathbb{R}, \end{aligned} \tag{3.5}$$

is called partition of unity for Ω.

Now, if we solve (3.4) for each subdomain and then define

$$u^{n+1} := \sum_{i=1}^{2} E_i(u_i^{n+1}), \tag{3.6}$$

we obtain the additive Schwarz method [13].

Doing the same, but setting

$$u^{n+1} := \sum_{i=1}^{2} E_i(\chi_i u_i^{n+1}) \tag{3.7}$$

leads to the restricted additive Schwarz method (RAS) [9].

For more details on different Schwarz methods see [18, 12].

3.2 Non-overlapping domain decomposition

Again consider (3.1) as an example. Suppose we have two non-overlapping subdomains Ω_1 and Ω_2 with the boundary Γ between them, and suppose n_1 and n_2 are the outward normals to the boundaries of Ω_1 and Ω_2. To satisfy (3.1) we require

$$\begin{aligned} -\Delta(u_i) &= f \quad && \text{in } \Omega_i,\ i = 1, 2, \\ u_1 &= u_2 && \text{on } \Gamma, \\ \frac{\partial u_1}{\partial n_1} + \frac{\partial u_2}{\partial n_2} &= 0 && \text{on } \Gamma, \\ u_i &= 0 && \text{on } \partial\Omega,\ i = 1, 2. \end{aligned} \tag{3.8}$$

An iterative Neumann-Neumann algorithm solves (3.8) in the following way. It starts from the initial guess u_i^0. On the k-th step first it solves the problem with a Dirichlet boundary conditions

$$\begin{aligned} -\Delta\left(u_i^{k-\frac{1}{2}}\right) &= f && \text{in } \Omega_i, \\ u_i^{k-\frac{1}{2}} &= \frac{1}{2}\left(u_1^{k-1} + u_2^{k-1}\right) && \text{on } \Gamma, \\ u_i^{k-\frac{1}{2}} &= 0 && \text{on } \partial\Omega, \end{aligned} \tag{3.9}$$

for $i = 1, 2$.

Then it finds the correction for the fluxes on the interface

$$\begin{aligned} -\Delta(e_i^k) &= f && \text{in } \Omega_i, \\ \frac{\partial e_i^k}{\partial n_i} &= -\frac{1}{2}\Big(\frac{\partial u_1^{k-\frac{1}{2}}}{\partial n_1} + \frac{\partial u_2^{k-\frac{1}{2}}}{\partial n_2}\Big) && \text{on } \Gamma, \\ e_i^k &= 0 && \text{on } \partial\Omega, \end{aligned} \tag{3.10}$$

for $i = 1, 2$.

And the next iteration is

$$u_i^k = u_i^{k-\frac{1}{2}} + e_i^k, \ i = 1, 2. \tag{3.11}$$

The iterative FETI algorithm works in the opposite order. It also starts from the initial guess u_i^0. On the k-th step first it solves the problem to correct the fluxes on the interface

$$\begin{aligned} -\Delta\Big(u_i^{k-\frac{1}{2}}\Big) &= f && \text{in } \Omega_i, \\ \frac{\partial u_i^{k-\frac{1}{2}}}{\partial n_i} &= \frac{\partial u_i^{k-1}}{\partial n_i} - \frac{1}{2}\Big(\frac{\partial u_1^{k-1}}{\partial n_1} + \frac{\partial u_2^{k-1}}{\partial n_2}\Big) && \text{on } \Gamma, \\ u_i^{k-\frac{1}{2}} &= 0 && \text{on } \partial\Omega, \end{aligned} \tag{3.12}$$

for $i = 1, 2$.

Then it makes corrections to maintain the continuity of the solution on the interface

$$\begin{aligned} -\Delta(e_i^k) &= f && \text{in } \Omega_i, \\ e_i^k &= \frac{1}{2}\Big(\frac{\partial u_{3-i}^{k-\frac{1}{2}}}{\partial n_{3-i}} - \frac{\partial u_i^{k-\frac{1}{2}}}{\partial n_i}\Big) && \text{on } \Gamma, \\ e_i^k &= 0 && \text{on } \partial\Omega, \end{aligned} \tag{3.13}$$

for $i = 1, 2$.

The next iteration is formed according to (3.11), the same way as for Neumann-Neumann method.

We will discuss a different approach for a non-overlapping domain decomposition in Section 3.4.

3.3 Schwarz methods as preconditioners

When we use numerical discretization methods to solve the problem we usually end up with solving the system of linear equations

$$Au = f, \tag{3.14}$$

where A is a square matrix, u is the solution we want to find, f is a right hand side vector formed according to our problem and boundary conditions.

Since solving the linear system directly can be computationally expensive as the size of the system increases, iterative methods to obtain an approximate solution were introduced. One of them is a fixed point iteration method. The basic idea of this method is the following.

We take an initial guess u^0 and then iterate by way of

$$u^{k+1} = u^k + Q^{-1}r^k, \tag{3.15}$$

where $r^k = f - Au^k$ and Q is a matrix of the same size as A that depends on the particular method. If we set $Q = A$ we get the exact solution immediately. Usually Q has a structure that is easy to invert.

If we set $P = Q - A$ and $e^n = u^n - u$, where u is the exact solution of the system (3.14), it is easy to show that

$$e^{n+1} = Q^{-1}Pe^n,$$

or

$$e^{n+1} = (Q^{-1}P)^{n+1}e^0. \tag{3.16}$$

So, the convergence of the fixed point iteration method depends on the properties of $Q^{-1}P$, which is called the iteration matrix.

Back to (3.15), we can notice that

$$u^{k+1} = u^k + Q^{-1}r^k = u^{k-1} + Q^{-1}r^{k-1} + Q^{-1}r^k = \ldots = u^0 + \sum_{i=0}^{k} Q^{-1}r^i.$$

It can be shown that $r^k = (PQ^{-1})^k r^0$, and, since $Q^{-1}(PQ^{-1})^k = (Q^{-1}P)^k Q^{-1}$, we

have

$$u^{k+1} = u^0 + \sum_{i=0}^{k} (Q^{-1}P)^i Q^{-1} r^0. \tag{3.17}$$

This leads us to the definition of Krylov subspaces.

Definition 3.3.1 (Krylov subspace)**.** Given a matrix B and a vector s, the Krylov subspace of dimension m will be

$$\mathcal{K}^m(B, s) = span(s,\ Bs, \ldots,\ B^{m-1}s). \tag{3.18}$$

Obviously, for any finite-dimensional pair $(B,\ s)$ at some point we will get $\mathcal{K}^{m+1}(B, s) = \mathcal{K}^m(B, s)$.

Looking at (3.17) we can see that $u^k - u^0 \in \mathcal{K}^k(Q^{-1}P, Q^{-1}r^0)$. The idea of Krylov type methods is, instead of taking a fixed element from Krylov subspace on each iteration, to find an optimal one in some sense in each subspace. The way it is done depends on the particular Krylov method and its implementation.

The Schwarz methods, discussed previously, can be considered a fixed point methods for a linear system. But, since Krylov type methods show better performance, the Schwarz methods are often used as preconditioners for them. As an example, see [12] for Additive Schwarz as a preconditioner for conjugate gradient method and RAS as a preconditioner for the generalized minimal residual (GMRES) and biconjugate gradient stabilized method (BiCGStab) methods.

3.4 Stochastic domain decomposition

Another possible approach, a stochastic domain decomposition, is to compute an approximate solution on some interface, which can then be used to split the initial domain into independent subdomains (see Figure 3.2). We generally introduce interfaces to obtain equal sized subdomains, which keeps the load on the subdomain problems balanced. One of the ways to obtain the solution of (2.38) and (2.39) at any particular point $(x, y) \in \Omega_p$ is by using Monte Carlo methods [1]. The approximation of the solution on the interface can then be used as a boundary conditions to solve the problems on the subdomains independently.

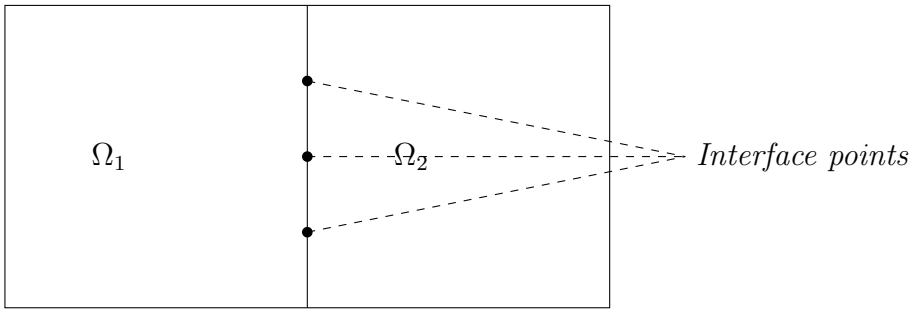

Figure 3.2: Illustration of stochastic domain decomposition.

We can rewrite both (2.38) and (2.39) in the general form

$$\mathcal{L}u = 0,\ u|_{\partial\Omega} = g, \tag{3.19}$$

where $\Omega \subset \mathbb{R}^2$ and

$$\mathcal{L} = \frac{1}{\rho(x,y)}\frac{\partial^2}{\partial x^2} + \frac{1}{\rho(x,y)}\frac{\partial^2}{\partial y^2} + \frac{\partial}{\partial x}\frac{1}{\rho(x,y)}\frac{\partial}{\partial x} + \frac{\partial}{\partial y}\frac{1}{\rho(x,y)}\frac{\partial}{\partial y}$$

is a linear elliptic operator and g is sufficiently smooth function. The probabilistic solution of (3.19) is given by

$$u(x,y) = E^{\mathcal{L}}_{(x,y)}\left[g(\beta(\tau_{\partial\Omega}))\right], \tag{3.20}$$

where $\beta(\cdot)$ denotes the stochastic process associated to the operator $\mathcal{L}$ and $\tau_{\partial\Omega}$ is the first time the stochastic process hits the boundary [22]. If we define

$$a(x,y) = \begin{bmatrix} \frac{1}{\rho(x,y)} & 0 \\ 0 & \frac{1}{\rho(x,y)} \end{bmatrix},\ b(x,y) = \begin{bmatrix} \frac{\partial}{\partial x}\frac{1}{\rho(x,y)} \\ \frac{\partial}{\partial y}\frac{1}{\rho(x,y)} \end{bmatrix},$$

then $\beta(\cdot)$ will be a solution of the stochastic differential equation

$$d\beta = b(x,y)dt + \sigma(x,y)dW(t), \tag{3.21}$$

where $\sigma(x,y)\sigma(x,y)^T = 2\,a(x,y)$ and $W(t)$ is a 2-dimensional standard Brownian motion.

To get the approximate numerical solution for $\beta(\cdot)$ we can use, for example, the Euler–Maruyama method [28].

The error of this approach consists of three parts. The first one, the statistical error, can be written as

$$\epsilon_{stat} = E^{\mathcal{L}}_{(x,y)}\left[g(\beta(\tau_{\partial\Omega}))\right] - \frac{1}{N}\sum_{i=1}^{N} g(\beta_i(\tau^i_{\partial\Omega})), \tag{3.22}$$

where β_i is the i-th realization of the stochastic process β, $\tau^i_{\partial\Omega}$ is the first time the process hits the boundary and N is the number of Monte Carlo simulations. For Monte Carlo methods this error is known to be $O(N^{-1/2})$, see e.g. the paper by Acebron et al. [1]. Other methods may have better convergence rate. Using the quasi-random sequences instead of pseudo-random numbers leads to the quasi-Monte Carlo method [8] have the convergence rate close to $O(N^{-1}_{MC})$. But it has problems with parallelization [7].

The second error we face is due to using an approximation of the stochastic process. we can write it as

$$\epsilon_{approx} = \frac{1}{N}\sum_{i=1}^{N} g(\beta_i(\tau^i_{\partial\Omega})) - \frac{1}{N}\sum_{i=1}^{N} g(\hat{\beta}_i(\tau^i_{\partial\Omega})), \tag{3.23}$$

where $\hat{\beta}_i$ is the numerical approximation of β_i. For the the Euler–Maruyama method this error is proportional to $O(\Delta t^{1/2})$.

The last error appears as we can miss the first time the process exits the domain, it may leave and returns back between two consecutive time steps. This error can be written as

$$\epsilon_{exit} = \frac{1}{N}\sum_{i=1}^{N} g(\hat{\beta}_i(\tau^i_{\partial\Omega})) - \frac{1}{N}\sum_{i=1}^{N} g(\hat{\beta}_i(t^i)), \tag{3.24}$$

where t^i is the approximation of the exit time.

The convergence rate for the Monte Carlo method is proportional to $N^{-1/2}$, so applying it to all of the interface points is still time-consuming. One possible way to circumvent this problem is to evaluate the stochastic solution only at certain points along the interfaces and compute the remaining interface points using interpolation.

Chapter 4

Results

In this chapter we will discuss the numerical results obtained using our parallel implementation of the Winslow mesh generator.

The mesh generator was implemented as a C++ library using the Message Passing Interface (MPI) [17, 10] and PETSc library [4, 5, 6]. PETSc provides a number of linear and nonlinear solvers, some of which can be used in parallel, and allows the user to easily switch between them without altering the source code. The implementation of our library can be found at *https://github.com/OAbr/WMG*. An example for the basic use of the library is provided in *ex1.cpp*. The implementation of the SDD method using Monte Carlo techniques is provided in *MC.cpp*. For more details see Appendix A.

At the moment the library can be used only for rectangular meshes because we restrict ourselves to the discretization schemes (2.42) and (2.43). It requires user-provided mapping of the boundaries in the physical space to the boundaries in the computational space as boundary conditions.

The tests were executed on the local cluster Torngat with nodes with the following specification: XL250a servers with 256 Gb RAM and E5-2650 processors with 24 cores per node in total; and on the Compute Canada cluster Graham which has the following specification of each node: 128 Gb RAM, two Intel E5-2683 v4 "Broadwell" CPUs with 32 cores per node in total.

For a complete list of results see the tables in Appendix B.

4.1 Parallelization of the linear solver

First, to test the parallel computing environment and have some baseline timings, we complete a strong scaling test with the PETSc built-in example 12 for KSP. This example solves Laplace's equation in parallel, which is very similar to our problem.

As a linear solver GMRES was used as implemented in the PETSc with BoomerAMG preconditioner provided by the HYPRE package [15]. BoomerAMG implements a parallel algebraic multigrid method with different available coarsening strategies.

The domain size was changed from 126×126 to 8001×8001 and the number of processors from 1 to 128. With perfect scaling, the CPU time should decrease proportionally to how the number of processors increases.

Three repetitions of each experiment were completed and the shortest time was recorded to negate the influence of random factors, such as activities of the background routines, communication delays, etc., on the timings.

size \ cores	1	2	4	8	16	32	64	128
126	0.030	0.031	0.047	0.073	0.129	0.260	0.650	1.093
251	0.110	0.083	0.092	0.113	0.185	0.430	0.779	1.207
501	0.460	0.322	0.287	0.243	0.318	0.404	0.796	1.316
1001	2.063	1.471	1.071	0.764	0.864	0.886	0.916	1.425
2001	8.703	7.241	5.331	3.607	3.197	2.562	2.076	2.180
4001	36.130	37.692	27.427	16.152	12.502	8.741	6.103	4.857
8001	147.094	200.187	156.235	89.780	58.800	41.278	24.264	15.385

Table 4.1: Time scaling of the PETSc example code with HYPRE preconditioner on Torngat cluster, time is minimized between three repetitions.

size \ cores	1	2	4	8	16	32	64	128
126	0.055	0.051	0.047	0.052	0.056	0.092	0.111	0.146
251	0.158	0.122	0.096	0.075	0.079	0.119	0.165	0.303
501	0.533	0.373	0.246	0.199	0.252	0.172	0.167	0.219
1001	1.888	1.303	0.865	0.503	0.485	0.519	0.578	0.466
2001	7.702	5.973	3.839	2.103	1.374	1.268	1.086	0.998
4001	31.718	29.082	20.146	10.474	6.154	4.266	2.782	2.002
8001	127.560	154.422	118.998	60.813	33.491	20.894	11.002	6.399

Table 4.2: Time scaling of the PETSc example code with HYPRE preconditioner on Graham cluster, time is minimized between three repetitions.

First of all we can notice that for small domains the CPU time increases as the number of processors increases. This happens because the communication between processors costs more than we gain from the parallelization. So, to get best time we do not want to use too many processors for a relatively small domain.

The second point to notice from Tables 4.1 and 4.2 is that Graham cluster is not just faster than Torngat cluster, but the problem is scaling better on Graham cluster as the number of processors increases (see Table B.11). This persisted throughout all our tests; we assume due to the faster network connection on Graham. It also shows that the network speed quickly becomes the key factor for the speed-up.

The third thing we can pay attention to is that for large domains the time increases with switching from 1 to 2 processors. Our analysis of the logs suggests that HYPRE spends much more time to set up the preconditioner when we switch from sequential to parallel mode. We see the same behavior on both clusters and with both the PETSc example code and our code. The reason for this requires further study.

As Graham's hardware setup allows better scaling, we will analyze later tests on this cluster.

Now that we have these results for the PETSc example code we can compare them to our code. Setting $\rho(x, y) = 1$ makes both problems almost identical (up to boundary conditions). Also we need to mention that our code essentially solves the problem twice, for $\xi(x, y)$ and $\eta(x, y)$. Running the tests for the same domains and numbers of processors produces the results shown in Table 4.3.

size \ cores	1	2	4	8	16	32	64	128
126	0.072	0.087	0.079	0.083	0.102	0.184	0.229	0.290
251	0.207	0.201	0.206	0.214	0.138	0.215	0.261	0.328
501	0.709	0.675	0.569	0.484	0.588	0.661	0.697	0.826
1001	2.740	2.255	1.631	1.090	1.015	1.144	0.979	1.098
2001	11.183	9.582	6.587	3.684	2.583	2.406	2.182	1.768
4001	46.830	43.812	31.046	16.369	10.374	7.686	5.612	5.004
8001	203.860	216.123	164.492	88.875	50.511	34.810	20.380	13.847

Table 4.3: Time scaling of our single domain solver for a constant mesh density function with HYPRE preconditioner on Graham cluster, time is minimized between three repetitions.

While the timing results are comparable, as we can see from Table 4.3, PETSc code performs better when the number of processors increases (Table B.15 shows this). We assume that the reason for this is that in our code the boundary conditions are filled only by one processor per subdomain. This was done because of balancing the CPU load for Monte Carlo simulations, but with a fixed number of subdomains it is a non-scalable element.

One of the reasons for using the PETSc library was that it provides a very flexible way of switching between different linear solvers. We tried to compare different preconditioners for our problem and the most interesting results were obtained with a GAMG multigrid preconditioner provided by the PETSc library.

size \ cores	1	2	4	8	16	32	64	128
126	0.164	0.152	0.126	0.099	0.111	0.189	0.239	0.312
251	0.533	0.419	0.319	0.276	0.192	0.207	0.255	0.359
501	1.986	1.377	0.873	0.544	0.545	0.490	0.463	0.540
1001	8.674	5.568	3.205	1.850	1.330	1.222	0.880	0.802
2001	35.273	23.471	13.165	7.295	4.480	3.794	2.594	1.685
4001	155.573	100.836	55.139	29.154	18.129	14.130	8.012	5.947
8001	685.254	430.775	233.815	123.058	75.546	NaN	31.672	20.537

Table 4.4: Time scaling of our single domain solver for a constant mesh density function with GAMG preconditioner on Graham cluster, time is minimized between three repetitions.

The comparison of Table 4.3 and Table 4.4 shows that HYPRE performs better for most cases, especially for one CPU, but GAMG has much better scaling. The test fails for GAMG with 32 CPUs as the node runs out of memory. This test requires the most memory per node as all 32 processors are located on the same machine. It shows that GAMG is more memory intensive than HYPRE.

4.2 SDD convergence studies

First, to test the convergence of the stochastic part of the SDD method, a domain with 201×101 grid points was considered. The mesh density function was chosen as

$$\rho(x, y) = 1 + R \exp(-50(x - x_c)^2 - 50(y - y_c)^2), \tag{4.1}$$

with $R = 15$, $x_c = 0.75$, $y_c = 0.5$ and the domain is a unit square. In this case uniform boundary conditions seem to be a reasonable choice.

As was mentioned in Section 2.2.4, to produce a new mesh after solving the problem we use interpolation. New meshes further in the text are obtained with a Matlab function *griddata()* which uses triangulation-based cubic interpolation. Different interpolation methods can affect the quality of the produced mesh.

To see how the quality of the meshes changes with the error produced by using

the Monte Carlo method, we can use the $2D$ version of the equidistribution quality measure.

We set

$$Q(K) = \frac{\rho(K)\,|K|}{\sigma_h},$$

where

$$\sigma_h = \frac{1}{N_{mesh}} \sum_K \rho(K)\,|K|\,,$$

K is an element of the mesh, $|K|$ is the area of the element K, $\rho(K)$ is the average value of the mesh density function over the element K and N_{mesh} is the number of mesh elements. Then the mesh quality measure will be $EQM = \max_K Q(K)$. Again, as in $1D$, we will have $EQM = \max_K Q(K)$ if the mesh is equidistributed.

We compare the results obtained by Monte Carlo simulations with the results obtained by solving the problem on a single domain. The first plot in Figure 4.1 shows the mapping from the initial uniform mesh to the computational domain for the mesh density function (4.1) with a single domain solver. The second plot shows the mesh obtained by interpolating the mapping of the uniform mesh in the computational space to the physical space.

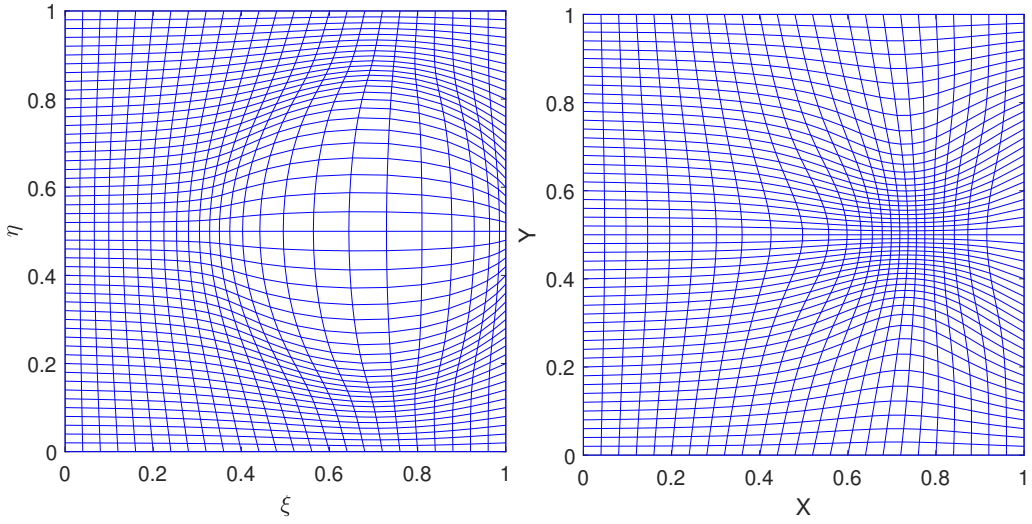

Figure 4.1: Mapping of the uniform mesh to the computational domain (left) and a new interpolated mesh in the physical domain (right) with the mesh density function (4.1) with 101 nodes in X direction and 201 nodes in Y direction produced by a single domain solver. Every $4th$ line is plotted.

Now we split this domain into two subdomains with 101×101 nodes in each. As we expect the stochastic part of the solver to be significantly time consuming, we want to use the stochastic solver only for a certain number of points on the interface, and then interpolate the solution for the rest of the interface points. We use two ways to choose the points where the stochastic solver will be used. The first is to choose every i^{th} point, which we we refer in the following as "uniform placement". The second way is to find the extreme points of the mesh density function and its first derivative along the interface. We then add further interpolation points into each interval between the extreme points. The endpoints of the interface are also treated as extreme points. This approach will be referred to as "adaptive placement". It is similar to the strategy proposed in [7].

For interpolation we use a cubic spline interpolation library implemented by Tino Kluge [24]. In the paper by Acebron et al. [1], the authors used Chebyshev polynomials for the similar purpose. In general, the best choice of the interpolation method, probably, depends on the form of the mesh density function used.

For different series of tests we vary one of the following variables: the number of Monte Carlo simulations, time step for the Euler–Maruyama method or the number of cores.

First, we start from changing the number of Monte Carlo simulations. We fix the time step at $\Delta t = 10^{-4}$. The number of Monte Carlo simulations changes from $N = 10^3$ to $N = 10^7$.

The first plot in Figure 4.2 shows how the error of the stochastic solver changes with different numbers of random walks in the case of a uniform placing of the interpolation points. The second plot shows the equidistributing quality measure of the corresponding interpolated meshes.

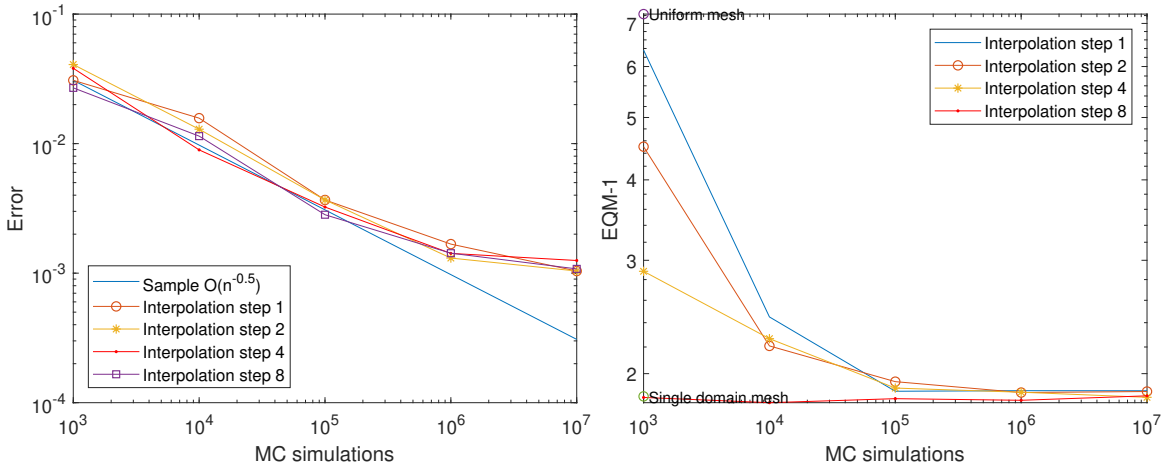

Figure 4.2: The maximum distance in the computational space between the interface nodes and the corresponding nodes obtained by a single domain solver (left) and the equidistributing quality measure of the newly obtained meshes (right). The uniform placing of the interpolation points is used.

As we can see from the first plot, while initially the convergence rate is close to the theoretical $O(N^{-1/2})$, after $N = 10^6$ Monte Carlo simulations it significantly slows down. Analyzing the similar results for $\Delta t = 10^{-3}$, where this happens earlier, and taking into account that the same behavior we see for the case with no interpolation (the interpolation step is 1), we conclude that the reason for this is that the time stepping error starts to dominate the Monte Carlo error at that point. Also we notice that for both plots the results are very close for all interpolation steps after $N = 10^5$ Monte Carlo simulations.

If we look more carefully at the error produced at the interface points, from Figure 4.3 we see that the error decreases more or less uniformly along the interface. The same picture we see for most of the subsequent tests.

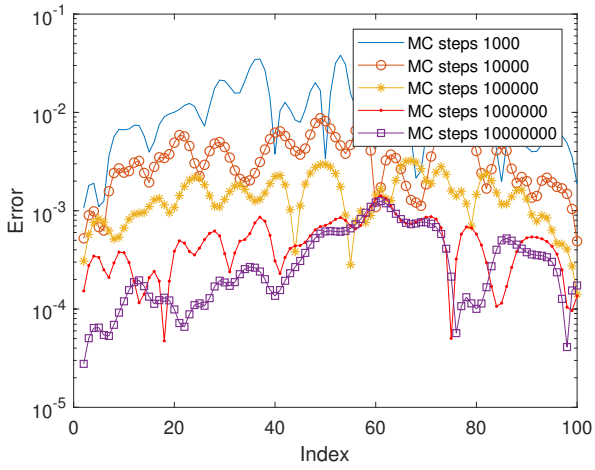

Figure 4.3: The distance in the computational space between the interface nodes and the corresponding nodes obtained by a single domain solver, the uniform placing of the interpolation points with the interpolation step 4 is used.

The timing results for this case are shown in Table 4.5. For all subsequent tests, except the core scaling experiments, 128 cores were used.

interpolation step \ N	1000	10000	100000	1000000	10000000
1	0.344	2.879	28.599	285.140	2854.290
2	0.125	1.423	14.315	142.571	1424.530
4	0.075	0.728	7.131	71.271	712.551
8	0.040	0.367	3.596	35.647	356.139

Table 4.5: Timing results for the stochastic solver with the number of Monte Carlo simulations scaling and different interpolation steps, the uniform placing of the interpolation points is used.

We can see that the time scales linearly with the number of random walks (along the rows) and linearly with the number of the interpolation points (along the columns).

Now we do the same for the adaptive strategy of placing the interpolation points.

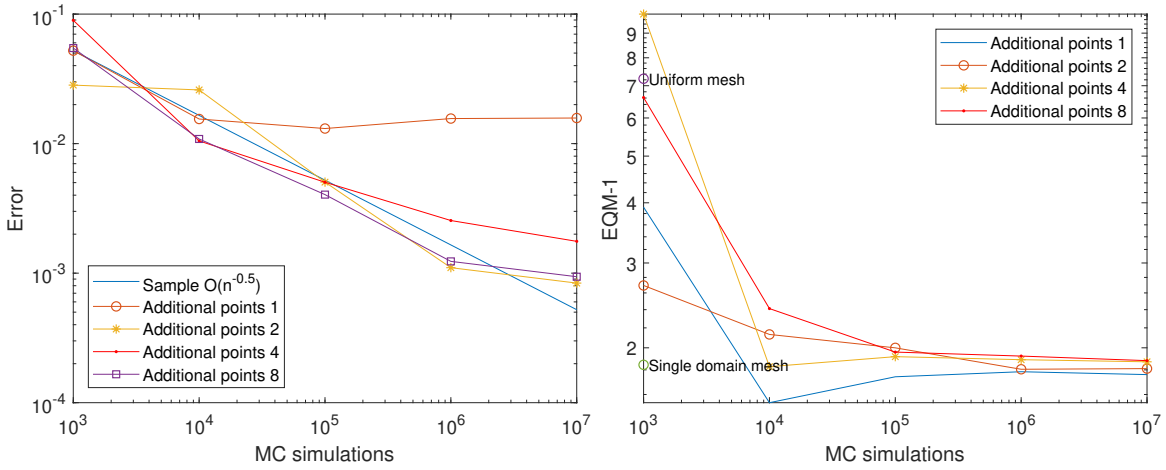

Figure 4.4: The maximum distance in the computational space between the interface nodes and the corresponding nodes obtained by a single domain solver (left) and the equidistributing quality measure of the newly obtained meshes (right). The adaptive placing of the interpolation points is used.

Figure 4.4 shows that all the cases, except for the case of 1 additional interpolation point per interval, the results behave similar to what we saw in the case of uniform placement. We see convergence rate slowdown after $N = 10^6$ Monte Carlo simulations. In the case of 1 additional interpolation point, however, the convergence stops after $N = 10^4$ Monte Carlo simulations. Figure 4.5 demonstrates the mapping and the new mesh for this case after $N = 10^7$ random walks. From there we can see that while the disruption around the interface in the computational space is relatively small, it results in the noticeable kinks in the newly produced mesh. For all other cases the final mesh looks similar to the one produced by the single domain solver.

If we plot the error along the interface for the 1 additional interpolation point case(see Figure 4.6), we see that it stays almost the same after $N = 10^4$ random walks for the area where the disruption appears. We assume the interpolation error is the main source of this error. Adding the additional interpolation point to each interval appears to be enough to overcome this problem.

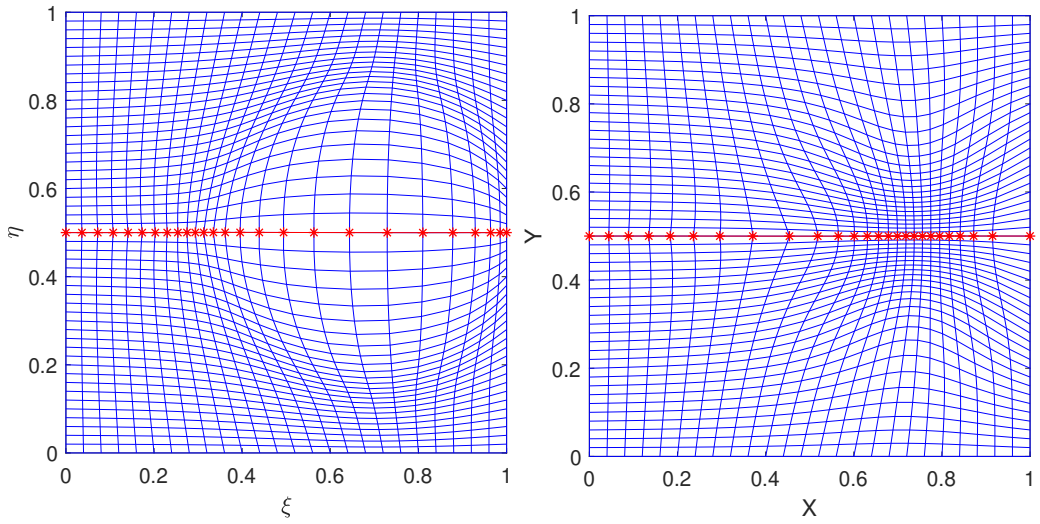

Figure 4.5: Mapping of the uniform mesh to the computational domain (left) and a new interpolated mesh in the physical domain (right) with the mesh density function (4.1) with 2 domains with 101×101 nodes in both directions, 1 additional interpolation point per interval for the adaptive placing of the interpolation points and $N = 10^7$ Monte Carlo simulations per interpolation point. Every $4th$ line is plotted. '-*-' line indicates the interface.

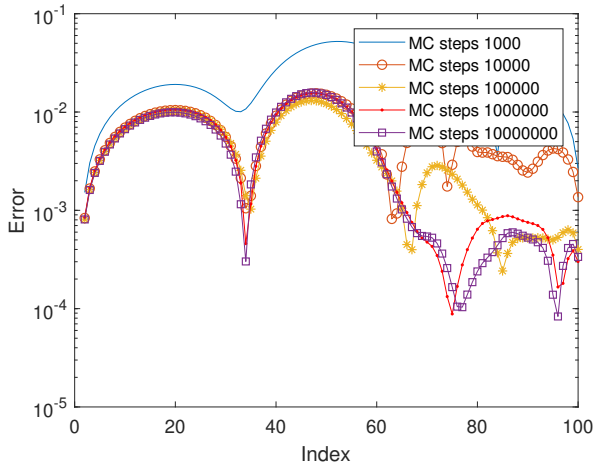

Figure 4.6: The maximum distance in the computational space between the interface nodes and the corresponding nodes obtained by a single domain solver (left) and the equidistributing quality measure of the newly obtained meshes (right). The adaptive placing of the interpolation points is used.

Table 4.6 shows the timing results for the case of adaptive placement. Similar as for the uniform placement strategy, we see the linear scaling along the rows. However,

the scaling along the columns is not so obvious. First of all, we do not directly control the total number of the interpolation points. In the case of our mesh density function we have 3 extreme points and, thus, 4 intervals. But, again, some intervals may have less nodes than the number of interpolation points we want to add. The total number of interpolation points for 1, 2, 4 and 8 additional points should be 7, 11, 19 and 35 respectively. If we look at the table, we see that the time scaling fits well to these numbers.

N / additional points	1000	10000	100000	1000000	10000000
1	0.021	0.186	1.842	18.495	183.757
2	0.026	0.290	2.803	28.113	281.844
4	0.044	0.469	4.778	47.330	475.956
8	0.085	0.888	8.634	85.559	856.931

Table 4.6: Timing results for the stochastic solver with number of Monte Carlo simulations scaling and different number of additional interpolation points, adaptive placing of the interpolation points is used.

After testing how the stochastic solver scales with the number of Monte Carlo simulations we test the scaling while changing the time step for the Euler–Maruyama method. The number of random walks per processor is fixed at $N = 10^6$ for these tests. Time step Δt scales from 10^0 to 10^{-5}. Again, we study two different placement strategies starting with the uniform one.

From Figure 4.7 we can notice that the initial error here is much larger than the one of Monte Carlo simulations scaling.

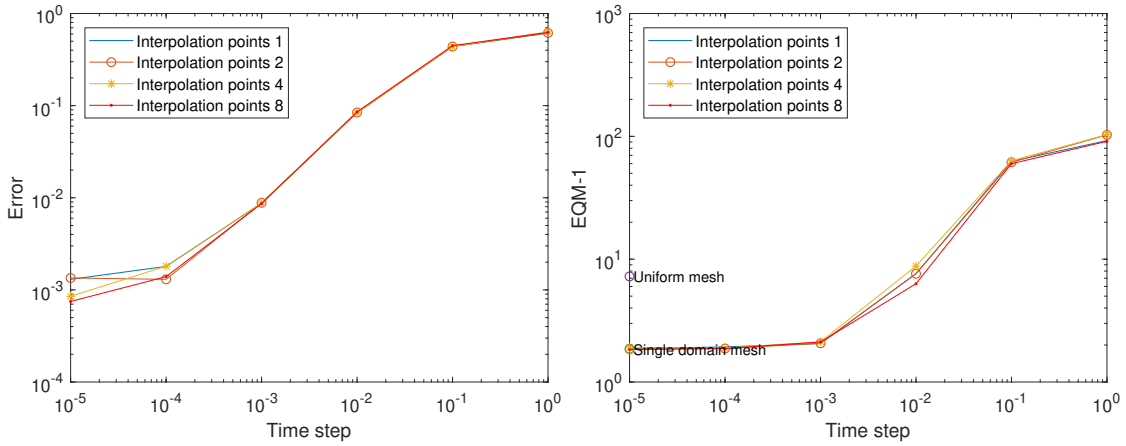

Figure 4.7: The maximum distance in the computational space between the interface nodes and the corresponding nodes obtained by a single domain solver (left) and the equidistributing quality measure of the newly obtained meshes (right). The uniform placing of the interpolation points is used.

If we plot the meshes for this case, from Figures 4.8 and 4.9 we can see that the mesh folds until time step is decreased to $\Delta t = 10^{-3}$. Mesh folding may appear with stochastic solver as long as the error of the method is larger than the distance between mesh nodes.

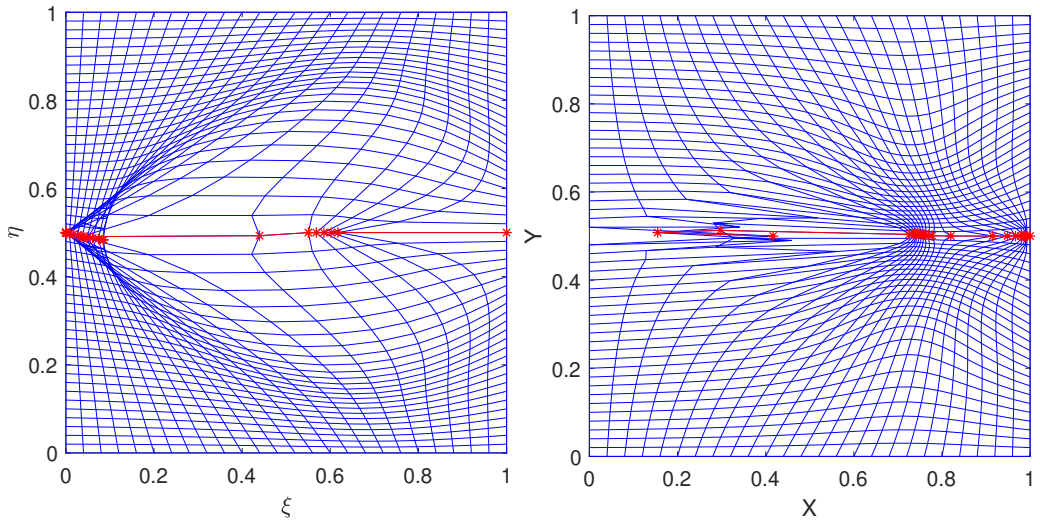

Figure 4.8: Mapping of the uniform mesh to the computational domain (left) and a new interpolated mesh in the physical domain (right) with the mesh density function (4.1) with 2 domains with 101×101 nodes in both directions, $N = 10^6$ Monte Carlo simulations per each interface node and $\Delta t = 1$. Every $4th$ line is plotted. '-*-' line indicates the interface.

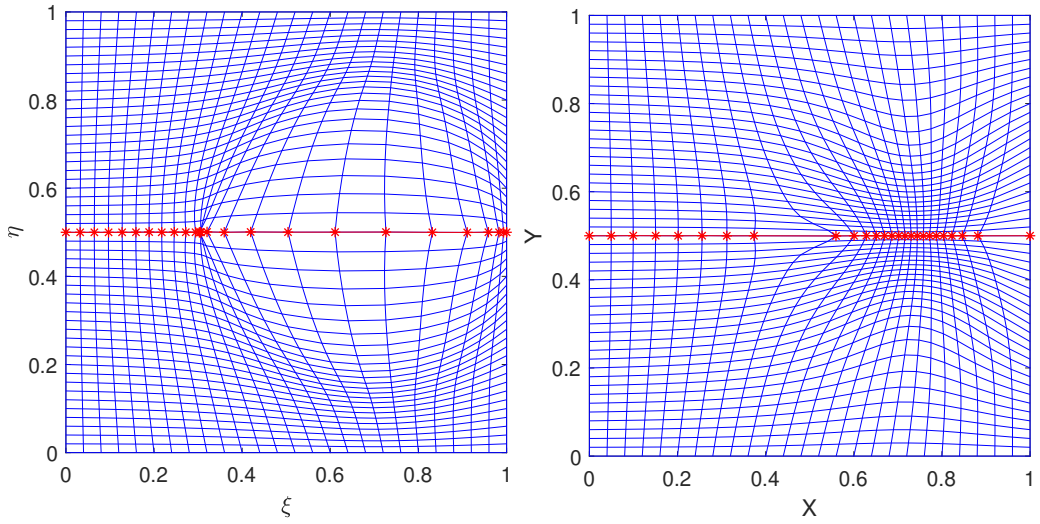

Figure 4.9: Mapping of the uniform mesh to the computational domain (left) and a new interpolated mesh in the physical domain (right) with the mesh density function (4.1) with 2 domains with 101×101 nodes in both directions, $N = 10^6$ Monte Carlo simulations per each interface node and $\Delta t = 10^{-2}$. Every $4th$ line is plotted. '-*-' line indicates the interface.

Table 4.7 shows that the time scales unpredictably until we come to some asymptotic behavior at approximately $\Delta t = 10^{-3}$, after which we have linear scaling.

time step / interpolation step	1	0.1	0.01	0.001	0.0001	0.00001
1	0.165	0.187	0.824	24.149	285.476	2917.860
2	0.081	0.094	0.413	12.064	142.471	1457.940
4	0.041	0.048	0.208	6.028	71.410	727.725
8	0.022	0.025	0.105	3.050	35.662	361.817

Table 4.7: Timing results for the stochastic solver with time step scaling and different interpolation steps, uniform placing of the interpolation points is used.

Figure 4.10 shows the convergence rates of the stochastic method with time step scaling with the adaptive placement of the interpolation points.

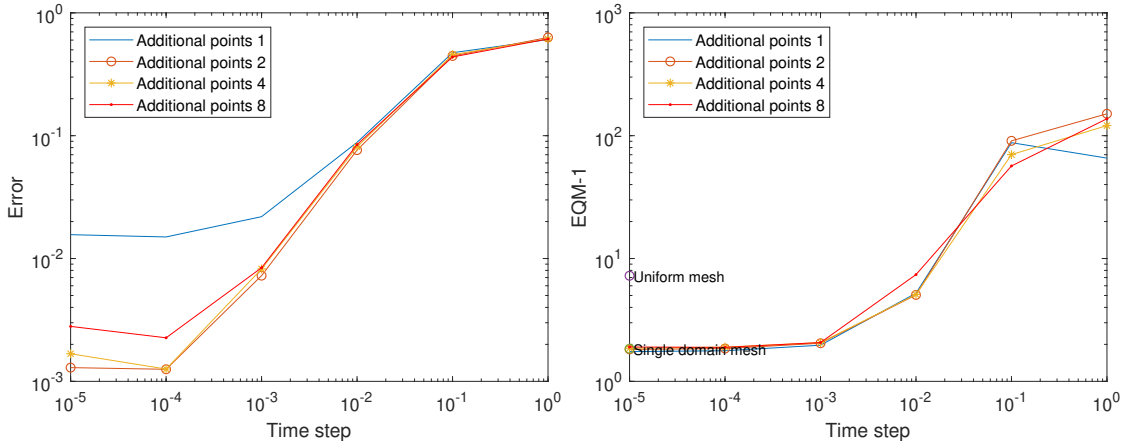

Figure 4.10: The maximum distance in the computational space between the interface nodes and the corresponding nodes obtained by a single domain solver (left) and the equidistributing quality measure of the newly obtained meshes (right). The adaptive placing of the interpolation points is used.

Again, here we have mesh folding for large time steps. The poor scaling in the case of 1 additional interpolation point per interval can be explained by the interpolation error and the meshes obtained in this case are similar to those obtained with scaling the number of Monte Carlo simulations. For all other cases, for both strategies of the interpolation point's placement, the quality of the mesh becomes close after Δt

decreases beyond $\Delta t = 10^{-3}$ and the scaling slows down after $\Delta t = 10^{-4}$ due to not enough number of Monte Carlo simulations.

time step / additional points	1	0.1	0.01	0.001	0.0001	0.00001
1	0.014	0.015	0.057	1.576	18.326	187.321
2	0.020	0.022	0.085	2.429	28.105	288.398
4	0.033	0.037	0.141	4.033	47.346	487.073
8	0.058	0.067	0.254	7.310	85.841	884.224

Table 4.8: Timing results for the stochastic solver with time step scaling scaling and different interpolation number of additional interpolation points, adaptive placing of the interpolation points is used.

Time scaling in Table 4.8 follows the same template as in the uniform case.

Tables 4.9 and 4.10 show the time scaling using different number of computational cores. The number of Monte Carlo simulations per node was fixed at $N = 10^6$ and the time step at $\Delta t = 10^{-3}$.

cores / interpolation	2	4	8	16	32	64	128
1	1542.730	778.697	388.921	192.977	96.659	48.251	24.128
2	768.659	385.581	193.095	96.452	48.749	24.129	12.084
4	385.091	192.949	96.399	48.129	24.184	12.072	6.035
8	192.725	96.290	48.107	24.014	12.052	6.024	3.022

Table 4.9: Timing results for the stochastic solver with number of cores scaling and different interpolation steps, uniform placing of the interpolation points is used.

cores / interpolation	2	4	8	16	32	64	128
1	99.779	49.848	24.952	12.530	6.254	3.138	1.572
2	153.081	76.818	38.393	19.191	9.603	4.797	2.421
4	258.188	128.794	64.593	32.325	16.139	8.097	4.040
8	466.734	233.346	116.636	58.400	29.158	14.595	7.301

Table 4.10: Timing results for the stochastic solver with number of cores scaling and different number of additional interpolation points, adaptive placing of the interpolation points is used.

As we can see, the main benefit of the Monte Carlo method is that it is fully parallelizable. And the tables in this chapter prove the linear scaling with all the parameters we tested when we have a reasonable load per processor.

4.3 SDD performance

Now, to see if the SDD method can improve the computational time for obtaining adapted mesh, we compare it with a single domain solver with different sizes of meshes and with different domain decomposition strategies.

Table 4.11 contains the scaling results for the single domain solver with the same mesh density function (4.1). We can see that the time scales almost linearly with the number of nodes (column wise) and somewhat slower than linearly with the number of processors (row wise).

size \ cores	1	2	4	8	16	32	64	128
257	0.490	0.394	0.277	0.190	0.149	0.149	0.165	0.206
513	1.958	1.361	0.803	0.468	0.365	0.351	0.287	0.449
1025	8.975	5.638	3.158	1.736	1.096	0.915	0.594	0.471
2049	36.545	23.324	13.044	7.178	4.542	3.498	2.037	1.515
4097	162.470	103.830	57.443	30.528	18.621	14.333	7.654	5.015
8193	715.970	449.094	242.996	129.394	81.422	NA	41.405	19.194

Table 4.11: Timing results on a single domain. GAMG preconditioner is used. Size parameter indicates the mesh with $size \times size$ nodes.

One thing to notice here is that for 32 cores we do not get a solution because the computational node runs out of memory.

The first domain decomposition strategy will be to split the initial domain into 2 subdomains and use the available processors evenly to solve the problem on each of them. The second way is to split the initial domain into equal subdomains and assign 1 processor to each subdomain. As the accuracy provided by the Monte Carlo method will be relatively low due to the slow convergence, the interpolation error will not be significant here and therefore we can use reasonably few interpolation points.

Better strategies for placing the interpolation points and different interpolation methods can further reduce the number of points requiring Monte Carlo simulations. For example, in [1] authors used just up to 3 points per interface with Chebyshev interpolation [30]. We start with $N = 10^5$ Monte Carlo simulations per interpolation point and $\Delta t = 10^{-3}$, using the adaptive placing of the interpolation points and 4 additional interpolation points per each interval.

size \ cores	2	4	8	16	32	64	128
257	37.919	19.081	9.585	4.834	2.490	1.337	0.742
513	39.157	19.523	9.943	4.996	2.637	1.433	0.804
1025	41.707	21.493	11.056	5.628	3.188	1.727	0.947
2049	55.827	30.084	16.052	8.642	5.332	2.886	2.677
4097	117.293	67.337	38.023	21.778	16.339	11.243	4.605
8193	382.749	229.035	129.830	77.169	NA	40.513	17.474

Table 4.12: Timing results for SDD with two subdomains, total time. $N = 10^5$ Monte Carlo simulations, $\Delta t = 10^{-3}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes.

Table 4.12 records the total time to obtain the solution of the problem, while Tables 4.13 and 4.14 show the fractions of time taken by the stochastic solver and the single domain solver respectively. If we compare Tables 4.11 and 4.12, we see that for 8193×8193 the SDD method works faster than a single domain solver for most cases.

size \ cores	2	4	8	16	32	64	128
257	37.683	18.899	9.444	4.719	2.367	1.189	0.595
513	38.218	18.895	9.550	4.725	2.367	1.201	0.593
1025	37.714	18.927	9.466	4.737	2.484	1.209	0.603
2049	37.834	18.924	9.495	4.770	2.422	1.237	0.740
4097	37.910	19.111	9.669	4.910	2.593	1.459	0.803
8193	38.570	19.709	10.260	5.506	NA	4.676	1.407

Table 4.13: Timing results for SDD with two domains, stochastic solver's time. $N = 10^5$ Monte Carlo simulations, $\Delta t = 10^{-3}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes.

size \ cores	2	4	8	16	32	64	128
257	0.236	0.182	0.140	0.115	0.123	0.148	0.147
513	0.940	0.628	0.393	0.271	0.270	0.232	0.210
1025	3.993	2.566	1.590	0.891	0.704	0.518	0.343
2049	17.992	11.159	6.556	3.872	2.909	1.649	1.937
4097	79.384	48.225	28.354	16.868	13.746	9.783	3.803
8193	344.179	209.326	119.570	71.663	NA	35.836	16.068

Table 4.14: Timing results for SDD with two domains, single domain solver's time. $N = 10^5$ Monte Carlo simulations, $\Delta t = 10^{-3}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes.

Looking at Table 4.13 we see the same linear scaling with the number of processors, as was shown in the previous section. But also we see that the time does not increase with increasing the number of cores, as the number of interpolation points stays the same. So, when the stochastic solver time becomes small enough in comparison to the single domain solver time, the linear scaling of the stochastic solver allows us to improve the time of the single domain solver, when it scales slower than linearly.

Moving to the multiple domain case, Table 4.15 contains the total time to obtain the solution of problem, while Tables 4.16 and 4.17 show the fractions of time taken

by the stochastic solver and the single domain solver respectively.

size \ cores	4	8	16	32	64	128
257	31.942	29.490	19.726	16.289	10.428	8.088
513	32.380	29.753	19.808	16.347	10.441	8.350
1025	33.802	30.718	20.223	16.546	10.934	8.446
2049	41.021	34.014	22.241	17.685	10.923	9.055
4097	69.934	49.935	31.352	23.168	12.699	9.957
8193	199.965	116.982	67.974	48.701	20.617	15.390

Table 4.15: Timing results for SDD with one subdomain per processor, total time. $N = 10^5$ Monte Carlo simulations, $\Delta t = 10^{-3}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes.

The comparison of Tables 4.11 and 4.15 shows that this approach for most cases will not improve the single domain solver's time.

size \ cores	4	8	16	32	64	128
257	31.813	29.413	19.677	16.255	10.399	8.073
513	31.916	29.517	19.684	16.282	10.403	8.328
1025	31.787	29.722	19.730	16.313	10.810	8.371
2049	31.827	29.538	19.743	16.337	10.460	8.818
4097	32.077	29.975	19.873	16.665	10.678	9.017
8193	32.984	30.307	20.557	17.745	11.711	11.372

Table 4.16: Timing results for SDD with one subdomain per processor, stochastic solver's time. $N = 10^5$ Monte Carlo simulations, $\Delta t = 10^{-3}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes.

From Table 4.16 we see that now the stochastic part scales much worse. The reason is that as we increase the number of the interfaces, we increase the number of points for the stochastic solver. The asymptotic scaling for the stochastic part here is $O(cores^{-1/2})$, while the single domain solver scales almost linearly.

size \ cores	4	8	16	32	64	128
257	0.129	0.078	0.049	0.033	0.030	0.016
513	0.464	0.236	0.124	0.066	0.038	0.023
1025	2.016	0.996	0.493	0.233	0.123	0.074
2049	9.194	4.477	2.498	1.347	0.463	0.237
4097	37.857	19.960	11.479	6.504	2.021	0.940
8193	166.981	86.676	47.417	30.956	8.905	4.018

Table 4.17: Timing results for SDD with one subdomain per processor, single domain solver's time. $N = 10^5$ Monte Carlo simulations, $\Delta t = 10^{-3}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes.

The linear solver part, on the other hand, scales very well. But the real problem with this strategy can be seen if we check the quality of the produced meshes.

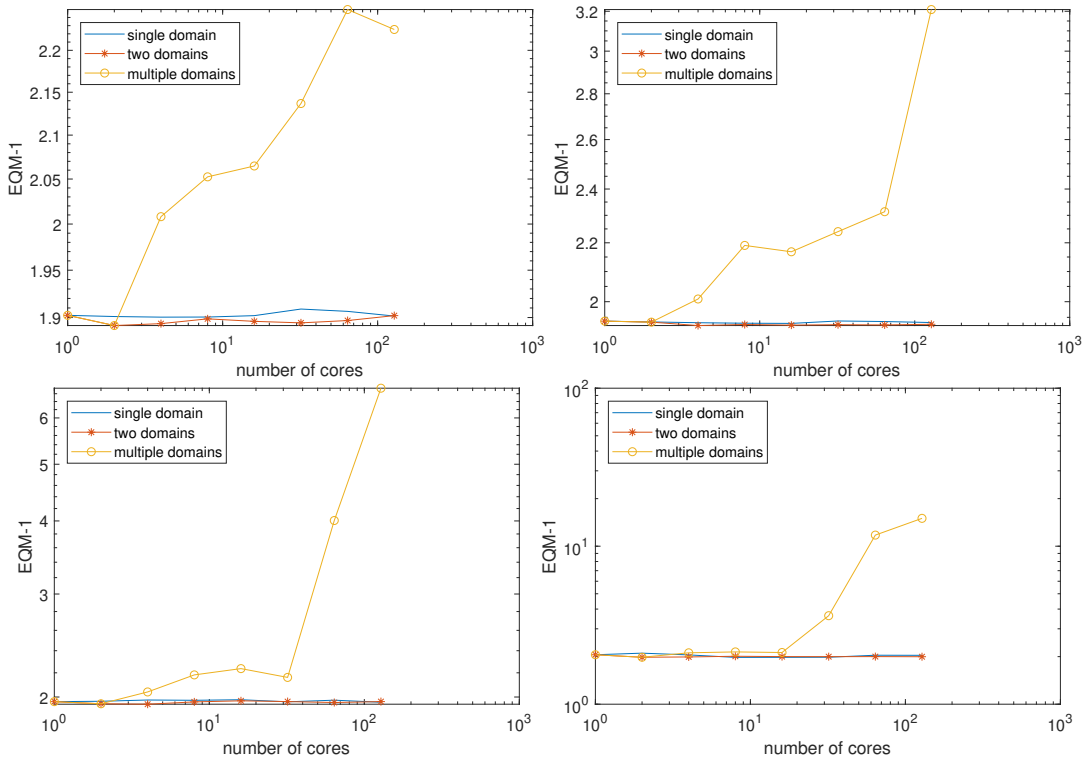

Figure 4.11: Equidistribution quality measure for the meshes produced with different SDD approaches. $N = 10^5$ Monte Carlo simulations per interpolation point, $\Delta t = 10^{-3}$ and 4 additional interpolation points per interval. From left to right, from top to bottom: 257×257 nodes, 513×513 nodes, 1025×1025 nodes, 2049×2049 nodes.

Looking at Figure 4.11 we see that the quality of the meshes produced with the multiple domain approach dramatically decreases. We do not provide the results for the larger meshes because the interpolation of the meshes was done in Matlab on a single desktop machine, which simply runs out of memory for the last two meshes. There are two reasons the quality of the multi domain meshes is decreasing. First is that we add more interfaces and the solution appears to be sensitive to this, which decreases the quality inside a single plot. The second reason is that with increasing the number of mesh points the distance between them decreases, so we need a higher accuracy to maintain the same quality of the produced meshes. This is why we see the decrease of quality between different plots. The two domain case, on the other hand, seems to still have a reasonable mesh quality.

To test what happens with better accuracy we run the same tests with $N = 10^6$ Monte Carlo simulations per interpolation point and $\Delta t = 10^{-4}$. The number of

additional points per interval stays the same, 4.

size \ cores	2	4	8	16	32	64	128
257	4470.886	2238.274	1119.371	560.187	280.367	140.192	70.053
513	4490.083	2242.759	1121.240	561.351	281.179	140.613	70.338
1025	4483.923	2272.542	1123.146	562.655	281.780	140.722	70.384
2049	4498.617	2253.685	1128.353	565.205	286.153	142.366	71.247
4097	4608.248	2292.071	1149.204	577.898	295.543	149.357	74.103
8193	4825.640	2454.146	1240.984	640.164	NA	177.475	86.973

Table 4.18: Timing results for SDD with two subdomains, total time. $N = 10^6$ Monte Carlo simulations, $\Delta t = 10^{-4}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes.

size \ cores	2	4	8	16	32	64	128
257	4470.640	2238.090	1119.230	560.067	280.243	140.058	69.898
513	4489.140	2242.130	1120.840	561.076	280.810	140.388	70.057
1025	4479.880	2269.970	1121.550	561.768	281.072	140.203	70.031
2049	4480.610	2242.490	1121.840	561.353	283.206	140.513	70.162
4097	4528.720	2243.370	1120.690	561.289	282.199	142.067	70.305
8193	4481.740	2245.100	1121.300	568.674	NA	142.618	70.953

Table 4.19: Timing results for SDD with two domains, stochastic solver's time. $N = 10^6$ Monte Carlo simulations, $\Delta t = 10^{-4}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes.

size \ cores	2	4	8	16	32	64	128
257	0.246	0.184	0.141	0.120	0.124	0.134	0.155
513	0.943	0.629	0.400	0.275	0.369	0.225	0.281
1025	4.043	2.572	1.596	0.887	0.708	0.519	0.353
2049	18.006	11.195	6.513	3.852	2.947	1.853	1.085
4097	79.528	48.701	28.514	16.609	13.344	7.290	3.799
8193	343.900	209.046	119.684	71.490	NA	34.857	16.020

Table 4.20: Timing results for SDD with two domains, single domain solver's time. $N = 10^6$ Monte Carlo simulations, $\Delta t = 10^{-4}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes.

From Tables 4.18, 4.19 and 4.20 we see that the stochastic part for the two domain case takes to long now to benefit from domain decomposition.

size \ cores	4	8	16	32	64	128
257	3752.830	3457.238	2296.272	1906.366	1212.369	941.431
513	3757.256	3461.667	2302.854	1905.941	1216.167	976.296
1025	3763.530	3463.246	2304.474	1903.724	1222.494	974.807
2049	3766.089	3499.997	2302.692	1911.260	1225.564	982.812
4097	3794.418	3483.196	2313.182	1908.049	1301.211	1016.075
8193	3923.317	3555.126	2348.966	1933.804	1259.786	1158.424

Table 4.21: Timing results for SDD with one subdomain per processor, total time. $N = 10^6$ Monte Carlo simulations, $\Delta t = 10^{-4}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes.

size \ cores	4	8	16	32	64	128
257	3752.700	3457.160	2296.220	1906.330	1212.320	941.279
513	3756.790	3461.430	2302.730	1905.870	1216.130	976.273
1025	3761.500	3462.250	2303.980	1903.490	1222.370	974.734
2049	3756.730	3495.530	2300.180	1909.960	1225.020	982.551
4097	3756.530	3463.280	2301.670	1901.550	1299.300	1015.130
8193	3756.380	3462.760	2302.020	1905.430	1250.880	1154.430

Table 4.22: Timing results for SDD with one subdomain per processor, stochastic solver's time. $N = 10^6$ Monte Carlo simulations, $\Delta t = 10^{-4}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes.

size \ cores	4	8	16	32	64	128
257	0.130	0.078	0.052	0.036	0.049	0.152
513	0.466	0.237	0.124	0.071	0.037	0.023
1025	2.030	0.996	0.494	0.234	0.124	0.073
2049	9.359	4.467	2.512	1.300	0.544	0.261
4097	37.888	19.916	11.512	6.499	1.911	0.945
8193	166.937	92.366	46.946	28.374	8.906	3.994

Table 4.23: Timing results for SDD with one subdomain per processor, single domain solver's time. $N = 10^6$ Monte Carlo simulations, $\Delta t = 10^{-4}$ and 4 additional interpolation points. The size parameter indicates the mesh with $size \times size$ nodes.

From Tables 4.21, 4.22 and 4.23 we see that for a multi domain approach the situation is even worse, as the scaling of the stochastic part is much slower.

We next study the behavior of the mesh quality.

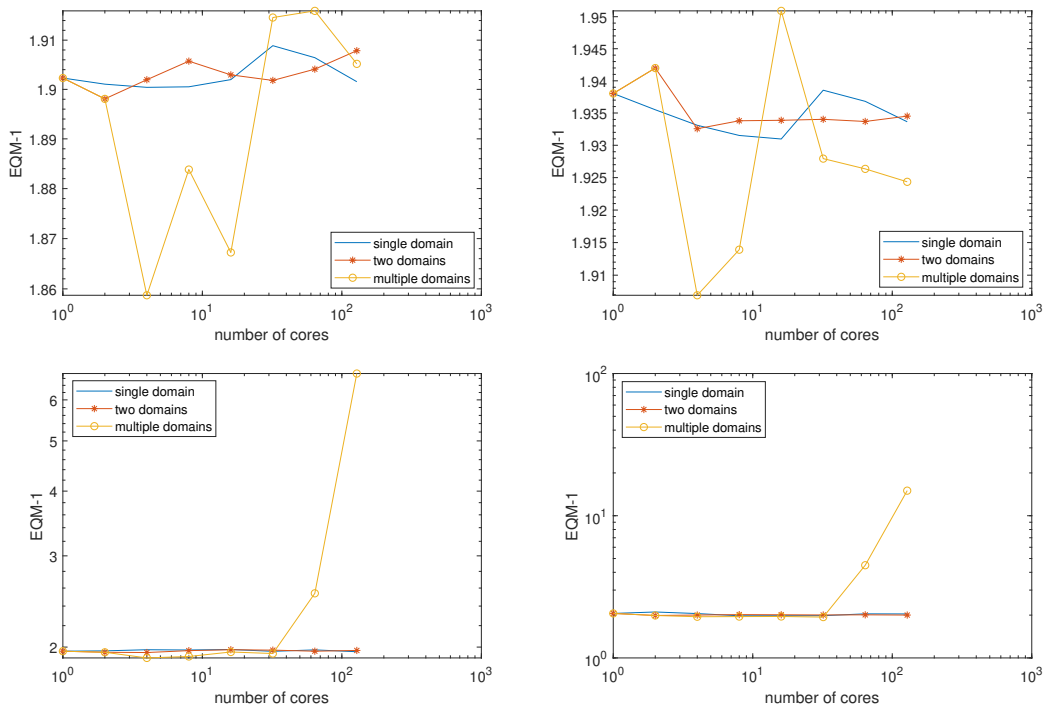

Figure 4.12: Equidistribution quality measure for the meshes produced with different SDD approaches. $N = 10^6$ Monte Carlo simulations per interpolation point, $\Delta t = 10^{-4}$ and 4 additional interpolation points per interval. From left to right, from top to bottom: 257×257 nodes, 513×513 nodes, 1025×1025 nodes, 2049×2049 nodes.

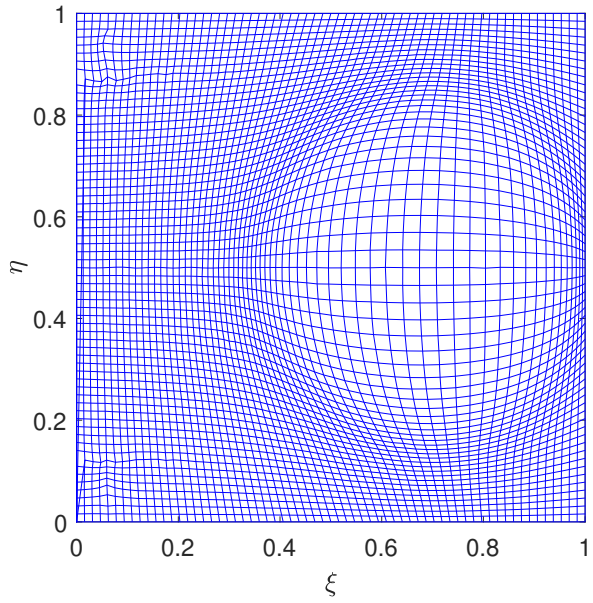

Figure 4.13: Mesh mapping to the computational space with folding. Mesh size 4097×4097 nodes, 128 cores, single core per subdomain, $N = 10^6$ Monte Carlo simulations, $\Delta t = 10^{-4}$ and 4 additional interpolation points. Every $64th$ line is plotted.

From Figure 4.12 we see that while for the smaller meshes the multi domain setup produces meshes of a comparable quality, for the larger meshes there is still not enough accuracy. Figure 4.13 demonstrates this as we can visibly see the mesh folding there even while printing only every $64th$ line.

Chapter 5

Conclusions

This chapter contains the conclusions of the work done for this thesis.

The convergence rate of different moving mesh methods was studied for $1D$. It was shown that the damped Newton method is the best in terms of the number of iterations for this type of problem. A good initial guess, for example obtained by using a few iterations of another method, can improve the convergence of the Newton method. Also, reordering the solution after each iteration of the Newton method to preserve monotonicity significantly improves the convergence of the method. The reason for this is still an open question.

Also, a Theorem was proven which shows that the IBVP method with a particular interpolation choice is equivalent to De Boor's method with piecewise constant or piecewise linear approximation to the mesh density function. This gives a deeper understanding of the nature of De Boor's method from a theoretical point of view.

A C++ library for parallel mesh adaptation was implemented. The library uses the flexibility of PETSc for the parallel linear solve. PETS allows the user to easily change the solvers and preconditioners. Multigrid methods seems to scale very good for this type of problem. This can be seen from comparing the timing tables for HYPRE (Table B.8) and GAMG (Table B.24) with a direct MUMPS solver (Table B.36)

The SDD method was implemented to allow a comparison with a parallel linear solver. It was shown that the interpolation along the interfaces can be used to reduce the solution time without significantly decreasing the quality of the stochastic solution.

Two different strategies for domain decomposition were tested. The strategy to sustain only as much subdomains as needed to keep the single domain solver close to the linear scaling justified itself for some cases even with a simplest stochastic solver discussed in this thesis. Other solvers with better convergence rates can make this approach even more interesting. The approach of assigning a single processor for each subdomain, on the other hand, does not seem be work. It appears to require much better accuracy of the stochastic solver while having significantly lower scaling rate of the stochastic part at the same time. This makes it difficult to compute all interfaces with a stochastic solver in the reasonable amount of time. Especially this is noticeable for the mesh sizes and numbers of processors tested in this work, as the single domain solver scales relatively good there.

While the scaling was tested with some pairs of linear solvers and preconditioners, using other methods may give different results. This is one of the questions still to investigate.

Another factor that may drastically change the timing results is using GPUs instead of CPUs. It may shift the balance between the Monte Carlo simulation time and the linear solver. There are some specifics of the hardware implementation of the GPU computations. First, GPU cores have less memory in comparison to CPU. Second, they can not communicate with each other directly. Third, for the best performance they should execute similar operations at the same time. In this situation different linear solvers can become more appropriate to use.

There is one more question to study. It would be interesting to use another numerical scheme for Winslow equations (2.38) and (2.39), which will allow us to use a non-rectangular initial mesh. This will enable us to make the method iterative and to study its convergence. Our tests of running the current numerical scheme iteratively, while are not expected to produce the correct results, have shown that the produced mesh tends to converge to the mesh obtained by solving 1D problems for ξ along the rows and for η along the columns. Why the convergence is observed at all and why we obtain such the resulting mesh is still not clear. Another thing to mention here, if the correct iterative method would converge to the better mesh, the SDD approach will need to be tested again. The accuracy of the stochastic part of the SDD method may be simply not enough to provide the convergence.

Finally we should mention that the timing results can be highly influenced by

different factors like available hardware, system configuration, network bandwidth and latency, etc.

Bibliography

[1] J. A. Acebrón, M. P. Busico, P. Lanucara, and R. Spigler. Domain decomposition solution of elliptic boundary-value problems via Monte Carlo and quasi-Monte Carlo methods. *SIAM J. Sci. Comput.*, 27(2):440–457, 2005.

[2] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.*, 23(1):15–41, 2001.

[3] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Comput.*, 32(2):136–156, 2006.

[4] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.6, Argonne National Laboratory, 2015.

[5] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang. PETSc Web page. http://www.mcs.anl.gov/petsc, 2015.

[6] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[7] A. Bihlo and R. D. Haynes. Parallel stochastic methods for PDE based grid generation. *Comput. Math. Appl.*, 68(8):804–820, 2014.

[8] R. E. Caflisch. Monte Carlo and quasi-Monte Carlo methods. In *Acta numerica, 1998*, volume 7 of *Acta Numer.*, pages 1–49. Cambridge Univ. Press, Cambridge, 1998.

[9] X.-C. Cai and M. Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM J. Sci. Comput.*, 21(2):792–797, 1999.

[10] L. Clarke, I. Glendinning, and R. Hempel. The MPI message passing interface standard. In *Programming environments for massively parallel distributed systems*, pages 213–218. Springer, 1994.

[11] C. de Boor. Good approximation by splines with variable knots. In *Spline functions and approximation theory (Proc. Sympos., Univ. Alberta, Edmonton, Alta., 1972)*, pages 57–72. Internat. Ser. Numer. Math., Vol. 21. Birkhäuser, Basel, 1973.

[12] V. Dolean, P. Jolivet, and F. Nataf. *An introduction to domain decomposition methods*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2015. Algorithms, theory, and parallel implementation.

[13] M. Dryja and O. B. Widlund. Some domain decomposition algorithms for elliptic problems. In *Iterative methods for large linear systems (Austin, TX, 1988)*, pages 273–291. Academic Press, Boston, MA, 1990.

[14] P. R. Eiseman. A multi-surface method of coordinate generation. *J. Comput. Phys.*, 33(1):118–150, 1979.

[15] R. D. Falgout, J. E. Jones, and U. M. Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In *Numerical solution of partial differential equations on parallel computers*, volume 51 of *Lect. Notes Comput. Sci. Eng.*, pages 267–294. Springer, Berlin, 2006.

[16] M. Farrashkhalvat and J. Miles. *Basic Structured Grid Generation: With an introduction to unstructured grid generation*. Elsevier, 2003.

[17] T. M. Forum. MPI: a Message Passing Interface, 1993.

[18] M. J. Gander. Schwarz methods over the course of time. *Electron. Trans. Numer. Anal.*, 31:228–255, 2008.

[19] W. J. Gordon and C. A. Hall. Construction of curvilinear co-ordinate systems and applications to mesh generation. *Internat. J. Numer. Methods Engrg.*, 7:461–477, 1973.

[20] W. Huang and R. D. Russell. *Adaptive moving mesh methods*, volume 174 of *Applied Mathematical Sciences*. Springer, New York, 2011.

[21] A. Iserles. *A first course in the numerical analysis of differential equations*. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, second edition, 2009.

[22] M. Kac. On distributions of certain Wiener functionals. *Transactions of the American Mathematical Society*, 65(1):1–13, 1949.

[23] A. Klawonn and O. Widlund. FETI and Neumann-Neumann iterative substructuring methods: connections and new results. *Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences*, 54(1):57–90, 2001.

[24] T. Kluge. Cubic spline interpolation library. https://kluge.in-chemnitz.de/opensource/spline/, 2014.

[25] P.-L. Lions. On the Schwarz alternating method. II. Stochastic interpretation and order properties. In *Domain decomposition methods (Los Angeles, CA, 1988)*, pages 47–70. SIAM, Philadelphia, PA, 1989.

[26] S. McGinn and R. E. Shaw. Parallel Gaussian elimination using OpenMP and MPI. In *High Performance Computing Systems and Applications, 2002. Proceedings. 16th Annual International Symposium on*, pages 169–173. IEEE, 2002.

[27] S. Milewski. Selected computational aspects of the meshless finite difference method. *Numerical Algorithms*, 63(1):107–126, 2013.

[28] G. N. Milstein and M. V. Tretyakov. *Stochastic numerics for mathematical physics*. Scientific Computation. Springer-Verlag, Berlin, 2004.

[29] J. D. Pryce. On the convergence of iterated remeshing. *IMA J. Numer. Anal.*, 9(3):315–335, 1989.

[30] T. J. Rivlin. *An introduction to the approximation of functions*. Dover Publications, Inc., New York, 1981. Corrected reprint of the 1969 original, Dover Books on Advanced Mathematics.

[31] K. H. A. Schwarz. Über einen Grenzübergang durch alternierendes Verfahren. *Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich*, 15:272–286, 1870.

[32] J. F. Thompson, B. K. Soni, and N. P. Weatherill, editors. *Handbook of grid generation*. CRC Press, Boca Raton, FL, 1999.

[33] J. F. Thompson, F. C. Thames, and C. W. Mastin. Automatic numerical generation of body-fitted curvilinear coordinate system for field containing any number of arbitrary two-dimensional bodies. *Journal of computational physics*, 15(3):299–319, 1974.

[34] A. M. Winslow. Numerical solution of the quasilinear Poisson equation in a nonuniform triangle mesh. *J. Computational Phys.*, 1:149–172, 1967.

[35] X. Xu, W. Huang, R. D. Russell, and J. F. Williams. Convergence of de Boor's algorithm for the generation of equidistributing meshes. *IMA J. Numer. Anal.*, 31(2):580–596, 2011.

Appendix A

Parallel library documentation

Here the documentation for our library for adaptive mesh generation is given. The library is available at *https://github.com/OAbr/WMG*.

The library is designed to compute the adapted mesh using Winslow's method for 2D problems[1]. It will required PETSc to be initialized to work.

Class WinslowMeshGenerator holds the data required for computations[2]. Constructor takes no parameters.

Public attributes of the class:

// physical domain
Vec globalX - holds X coordinates of the physical mesh, data type - PETSc vector(global), managed by DM da(see below).
Vec localX - holds locally stored portion of the X coordinates of the physical mesh with a stencil required for computations. Data type - PETSc vector(local), managed by DM da(see below).
Vec globalY - holds Y coordinates of the physical mesh, data type - PETSc vector(global), managed by DM da(see below).
Vec localY - holds locally stored portion of the Y coordinates of the physical mesh with a stencil required for computations. Data type - PETSc vector(local), managed by DM da(see below).

[1] Currently works only with rectangular meshes
[2] At the moment class constructor requires PETSc to be initialized, so a variable should be defined after PETSc initialization. I'm planning to change it in the future

// mesh density function[3]

Vec globalRho - holds mesh density values of the corresponding nodes of the physical mesh, data type - PETSc vector(global), managed by DM da(see below).

Vec localRho - holds locally stored portion of the mesh density values of the corresponding nodes of the physical mesh with a stencil required for computations. Data type - PETSc vector(local), managed by DM da(see below).

// Problem coords

PetscScalar left - scalar value that holds the minimum x coordinate of the physical mesh.

PetscScalar right - scalar value that holds the maximum x coordinate of the physical mesh.

PetscScalar bottom - scalar value that holds the minimum y coordinate of the physical mesh.

PetscScalar top - scalar value that holds the maximum y coordinate of the physical mesh.

// Number of nodes along X and Y axes

PetscInt globalSizeX - number of nodes along X axis.

PetscInt globalSizeY - number of nodes along Y axis.

// Computational domain mesh

Vec globalXi - holds ξ coordinates of the computational mesh, data type - PETSc vector(global), managed by DM da(see below).

Vec localXi - holds locally stored portion of the ξ coordinates of the computational mesh with a stencil required for computations. Data type - PETSc vector(local), managed by DM da(see below).

Vec globalEta - holds η coordinates of the computational mesh, data type - PETSc vector(global), managed by DM da(see below).

Vec localEta - holds locally stored portion of the η coordinates of the computational mesh with a stencil required for computations. Data type - PETSc vector(local), managed by DM da(see below).

[3]After mesh density function being set it is computed over all nodes and stored similar way as the mesh coordinates

// Distributed array data manager
DM da - data manager, that holds information about parallel data layout and provides subroutines for inter processes communication[4].

// Optional user data structure
*void *userCtx* - a pointer to additional user data, that can be used, for example, in mesh density function calculation.

Public methods of the class:

// Class constructor
WinslowMeshGenerator() - creates the object of the class.

void init() - initializes MPI information and PETSc linear solver. Automatically executed by constructor if PETSc is initialized when the *WinslowMeshGenerator* object is created. Otherwise needs to be executed after PETSc initialization and before setting the domain information.

// Set uniform physical domain mesh
void setUniformMesh(PetscScalar left, PetscScalar right, PetscInt sizeX, PetscScalar bottom, PetscScalar top, PetscInt sizeY) - function sets a uniform rectangular physical mesh using input parameters.

- *PetscScalar left* - the minimum x coordinate of the physical mesh, real number.
- *PetscScalar right* - the maximum x coordinate of the physical mesh, real number.
- *PetscInt sizeX* - the number of nodes along x axis, integer number.
- *PetscScalar bottom* - the minimum y coordinate of the physical mesh, real number.
- *PetscScalar top* - the maximum y coordinate of the physical mesh, real number.

[4]for more information see PETSc documentation

- *PetscInt sizeY* - the number of nodes along y axis, integer number.

// Set boundary conditions
void setComputationalBoundary(int boundaryID, Vec bottom, Vec top, Vec left, Vec right) - function sets boundary values for the computational domain. Should be executed only after setting the domain.

- *int boundaryID* - identifies for which variable the boundary is being set. Can take values *XI_ID* or *ETA_ID*.
- *Vec bottom* - a PETSc vector that contains boundary data corresponding to the nodes with minimum y coordinate.
- *Vec top* - a PETSc vector that contains boundary data corresponding to the nodes with maximum y coordinate.
- *Vec left* - a PETSc vector that contains boundary data corresponding to the nodes with minimum x coordinate.
- *Vec right* - a PETSc vector that contains boundary data corresponding to the nodes with maximum x coordinate.

*template ⟨class TScalar=double, class TIndex=int⟩ void SetPetscVector(Vec &pVec, const TScalar *cVec, TIndex vecSize, TIndex vecBegin, TIndex vecEnd, TIndex indexStep=1, MPI_Comm comm=PETSC_COMM_WORLD)* - routine provides an easy way for setting PETSc vectors for *setComputationalBoundary()* function from a distributed C++ array.

- *class TScalar* - data type of the input vector.
- *class TIndex* - data type of the index.
- *Vec &pVec* - a parallel PETSc vector that should be filled with boundary values. Will be initialized inside the routine. Needs to be destroyed manually with PETSc *VecDestroy()* function afterward.
- *const TScalar *cVec* - a pointer to the C++ array with data.
- *TIndex vecSize* - the local size of the vector to create.

- *TIndex vecBegin* - the first global index of the local vector.
- *TIndex vecEnd* - the the next after the last global index of the local vector.
- *TIndex indexStep* - allows to set only part of the values of the target vector, used by the library internally.
- *MPI_Comm comm* - the MPI communicator, on which we want to create a new vector.

For an example of usage, see *MC.cpp*.

// set mesh density function
*void setMeshDensityFunction(PetscErrorCode (*RhoFunc)(const WinslowMeshGenerator &))* - function set routine to calculate mesh density matrix. Should be executed only after the domain is set.

- *PetscErrorCode (*RhoFunc)(const WinslowMeshGenerator &)* - user provided routine to calculate mesh density matrix. Function should take *WinslowMeshGenerator* object as an input, modify it's *globalRho* matrix and return PetscErrorCode[5].

*template ⟨class TIn=double, class TOut=double⟩ PetscErrorCode directRhoComputation(const WinslowMeshGenerator &g, TOut (*density)(TIn x, TIn y))* - function allows to easily set the mesh density function for Winslow Mesh Generator (WMG) if density can be computed directly from coordinates.

- *class TIn* - data type of the input coordinates.
- *class TOut* - data type of the output value.
- *const WinslowMeshGenerator &g* - *WinslowMeshGenerator* object which is used as a solver.
- *TOut (*density)(TIn x, TIn y))* - user provided function to compute mesh density at the particular point.

[5]for more details about *PetscErrorCode* see PETSc documentation

The function implementation can be used as an example of how to implement a user-defined mesh density function. See *MC.cpp* for an example of using it.

void updateDensity() - function updates mesh density matrix.

// Solve the inverse problem
void solveComputationalCoords() - function solves the system for ξ and η values. Requires mesh density function and boundary conditions for both variables to be set.

*template ⟨class TOut=double⟩ TOut** getPetscVectorToZero(Vec pVec)* - function creates a C++ 2D array in process with MPI id 0 and copies the globally distributed *pVec* data to it. Should be used only after the domain is set.

- *class TOut* - data type of the output vector.
- *Vec pVec* - expected to be one of *globalX*, *globalY*, *globalRho*, *globalXi*, *globalEta* vectors.

The function implementation can be used as an example of how to work with the solution vector after the problem was solved.

template ⟨class TOut=double⟩ void printGlobal(Vec Out) - function sends the data from PETSc globally distributed vector to the process with MPI id 0 and prints it to the screen.

- *class TOut* - data type of the output vector.
- *Vec Out* - expected to be one of *globalX*, *globalY*, *globalRho*, *globalXi*, *globalEta* vectors.

void clear() - function clears data structures initialized by a solver.

void destroy() - function clears data structures initialized by a solver and destroys a PETSc linear solver if it was initialized. Should be called before PETSc finalized to maintain a correct memory management.

$\sim$*WinslowMeshGenerator()* - WMG solver's destructor. If executed after PETSc was finalized and not all internal data structures were cleared - will rise an error.

To compile a library:

- properly set *PETSC_DIR* and *PETSC_ARCH* variables in your environment;
- modify the *local_install* variable in the makefile to your preferences. Creating *include* and *lib* subfolders in the corresponding folder may be required;
- run "make WMG".

File *ex1.cpp* in the *Examples* folder provides a simple example of using the library. File *MC.cpp* from the same folder implements a stochastic domain decomposition method Winslow mesh generation method using this library as a single domain solver. To compile the example codes:

- compile the library first;
- modify the *local_install* variable in the makefile in the *Examples* folder to point the same folder as was used to compile the library;
- make sure *PETSC_DIR* and *PETSC_ARCH* variables are still properly set in your environment;
- run "make ex1" or "make MC" respectively.

To run the example codes add the folder with the compiled library to the *LD_LIBRARY_PATH* variable for LINUX or a similar environment variable for other operational systems.

Appendix B

Parallel environments' tests

This appendix contains the tables we obtained during testing our library on different clusters with different linear solvers and in different regimes, but which are excluded from the main text for the sake of brevity. **Note:** For each test the first table contains the shortest time over three repetitive runs to negate the influence of random factors, such as activities of the background routines, communication delays, etc., on the timings. The third table represents *test_time* $\times$ *number_of_cores* / *time_for_one_core* for each domain size.

B.1 HYPRE preconditioner

B.1.1 Torngat cluster

PETSc KSP ex12

size \ cores	1	2	4	8	16	32	64	128
126	0.030	0.031	0.047	0.073	0.129	0.260	0.650	1.093
251	0.110	0.083	0.092	0.113	0.185	0.430	0.779	1.207
501	0.460	0.322	0.287	0.243	0.318	0.404	0.796	1.316
1001	2.063	1.471	1.071	0.764	0.864	0.886	0.916	1.425
2001	8.703	7.241	5.331	3.607	3.197	2.562	2.076	2.180
4001	36.130	37.692	27.427	16.152	12.502	8.741	6.103	4.857
8001	147.094	200.187	156.235	89.780	58.800	41.278	24.264	15.385

Table B.1: Minimized timing results for the PETSc example code with HYPRE preconditioner on Torngat cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.020	1.043	3.657	2.135	1.896	1.832	1.143	1.137
251	1.018	1.053	1.289	1.695	1.287	1.274	1.220	1.222
501	1.014	1.030	1.212	1.390	2.151	1.384	1.148	1.141
1001	1.004	1.076	1.023	1.085	1.062	1.100	1.136	1.036
2001	1.010	1.019	1.020	1.117	1.038	1.118	1.152	1.182
4001	1.006	1.003	1.032	1.015	1.141	1.111	1.050	1.023
8001	1.004	1.003	1.010	1.004	1.061	1.228	1.046	1.003

Table B.2: Maximum time over minimum time for the PETSc example code with HYPRE preconditioner on Torngat cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.000	2.088	6.281	19.679	69.161	279.956	1398.687	4706.893
251	1.000	1.500	3.339	8.171	26.784	124.680	451.704	1399.607
501	1.000	1.399	2.497	4.226	11.059	28.109	110.750	366.200
1001	1.000	1.426	2.076	2.965	6.706	13.745	28.428	88.456
2001	1.000	1.664	2.450	3.315	5.878	9.422	15.266	32.065
4001	1.000	2.086	3.036	3.576	5.536	7.742	10.811	17.207
8001	1.000	2.722	4.249	4.883	6.396	8.980	10.557	13.388

Table B.3: Time scaling over the domain for the PETSc example code with HYPRE preconditioner on Torngat cluster.

Single domain solver with $\rho = 1$

size \ cores	1	2	4	8	16	32	64	128
126	0.052	0.053	0.096	0.153	0.300	0.801	1.222	1.924
251	0.197	0.148	0.173	0.219	0.431	0.730	1.496	2.218
501	0.936	0.631	0.631	0.554	0.631	0.946	1.677	3.173
1001	4.436	3.456	2.426	1.693	2.146	1.965	2.373	4.340
2001	17.483	15.333	10.489	7.420	6.600	7.215	6.382	7.166
4001	74.068	69.560	45.456	28.692	26.032	23.298	20.531	16.726
8001	326.766	343.213	242.150	139.358	107.833	93.427	70.625	52.849

Table B.4: Minimized timing results for the single domain solver with a constant mesh density function and HYPRE preconditioner on Torngat cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.013	1.085	1.394	1.552	1.103	1.584	1.239	1.116
251	1.009	1.095	1.424	1.486	1.411	1.409	1.260	1.059
501	1.005	1.125	1.265	1.137	1.214	1.319	1.165	1.088
1001	1.135	1.012	1.136	1.334	1.060	1.109	1.185	1.121
2001	1.003	1.009	1.036	1.066	1.117	1.042	1.124	1.224
4001	1.002	1.005	1.035	1.186	1.169	1.074	1.025	1.118
8001	1.002	1.013	1.011	1.037	1.067	1.106	1.037	1.007

Table B.5: Maximum time over minimum time for the single domain solver with a constant mesh density function and HYPRE preconditioner on Torngat cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.000	2.013	7.283	23.316	91.561	488.252	1489.984	4693.991
251	1.000	1.502	3.520	8.896	35.034	118.507	486.081	1441.053
501	1.000	1.349	2.697	4.740	10.785	32.366	114.678	434.047
1001	1.000	1.558	2.188	3.054	7.742	14.172	34.232	125.236
2001	1.000	1.754	2.400	3.395	6.040	13.206	23.365	52.469
4001	1.000	1.878	2.455	3.099	5.623	10.066	17.740	28.905
8001	1.000	2.101	2.964	3.412	5.280	9.149	13.833	20.702

Table B.6: Time scaling over the domain for the single domain solver with a constant mesh density function and HYPRE preconditioner on Torngat cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.133	1.176	0.978	0.957	0.856	0.650	1.064	1.137
251	1.120	1.119	1.063	1.029	0.857	1.179	1.041	1.088
501	0.983	1.020	0.910	0.877	1.008	0.854	0.950	0.830
1001	0.930	0.851	0.883	0.903	0.806	0.902	0.772	0.657
2001	0.996	0.945	1.017	0.972	0.969	0.710	0.651	0.608
4001	0.976	1.084	1.207	1.126	0.960	0.750	0.595	0.581
8001	0.900	1.167	1.290	1.288	1.091	0.884	0.687	0.582

Table B.7: Double PETSc example code time over single domain solver time with HYPRE preconditioner on Torngat cluster.

B.1.2 Graham cluster

PETSc KSP ex12

size \ cores	1	2	4	8	16	32	64	128
126	0.055	0.051	0.047	0.052	0.056	0.092	0.111	0.146
251	0.158	0.122	0.096	0.075	0.079	0.119	0.165	0.303
501	0.533	0.373	0.246	0.199	0.252	0.172	0.167	0.219
1001	1.888	1.303	0.865	0.503	0.485	0.519	0.578	0.466
2001	7.702	5.973	3.839	2.103	1.374	1.268	1.086	0.998
4001	31.718	29.082	20.146	10.474	6.154	4.266	2.782	2.002
8001	127.560	154.422	118.998	60.813	33.491	20.894	11.002	6.399

Table B.8: Minimized timing results for the PETSc example code with HYPRE preconditioner on Graham cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.015	1.173	1.150	1.098	1.047	1.059	2.437	7.142
251	1.018	1.027	1.098	1.072	1.010	1.082	1.746	2.749
501	1.017	1.020	1.060	1.168	1.059	1.061	1.764	6.672
1001	1.042	1.011	1.014	1.087	1.039	1.062	1.298	1.224
2001	1.004	1.012	1.008	1.046	1.036	1.011	1.109	1.149
4001	1.001	1.006	1.005	1.006	1.002	1.003	1.317	1.097
8001	1.261	1.196	1.128	1.117	1.098	1.057	1.126	1.131

Table B.9: Maximum time over minimum time for the PETSc example code with HYPRE preconditioner on Graham cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.000	1.843	3.454	7.552	16.406	53.578	129.114	341.745
251	1.000	1.547	2.434	3.792	7.963	24.147	66.829	245.564
501	1.000	1.400	1.845	2.985	7.573	10.355	20.060	52.643
1001	1.000	1.381	1.833	2.130	4.111	8.801	19.589	31.567
2001	1.000	1.551	1.994	2.184	2.855	5.266	9.020	16.581
4001	1.000	1.834	2.541	2.642	3.104	4.304	5.614	8.080
8001	1.000	2.421	3.732	3.814	4.201	5.242	5.520	6.421

Table B.10: Time scaling over the domain for the PETSc example code with HYPRE preconditioner on Graham cluster.

size \ cores	1	2	4	8	16	32	64	128
126	0.542	0.615	0.986	1.414	2.287	2.834	5.876	7.471
251	0.700	0.678	0.960	1.507	2.353	3.612	4.729	3.988
501	0.863	0.863	1.169	1.223	1.261	2.344	4.767	6.006
1001	1.093	1.129	1.237	1.521	1.782	1.706	1.586	3.062
2001	1.130	1.212	1.389	1.715	2.327	2.022	1.912	2.185
4001	1.139	1.296	1.361	1.542	2.032	2.049	2.194	2.426
8001	1.153	1.296	1.313	1.476	1.756	1.976	2.205	2.404

Table B.11: Torngat cluster time over Graham cluster time for the PETSc example code with HYPRE preconditioner.

Single domain solver with $\rho = 1$

size \ cores	1	2	4	8	16	32	64	128
126	0.072	0.087	0.079	0.083	0.102	0.184	0.229	0.290
251	0.207	0.201	0.206	0.214	0.138	0.215	0.261	0.328
501	0.709	0.675	0.569	0.484	0.588	0.661	0.697	0.826
1001	2.740	2.255	1.631	1.090	1.015	1.144	0.979	1.098
2001	11.183	9.582	6.587	3.684	2.583	2.406	2.182	1.768
4001	46.830	43.812	31.046	16.369	10.374	7.686	5.612	5.004
8001	203.860	216.123	164.492	88.875	50.511	34.810	20.380	13.847

Table B.12: Minimized timing results for the single domain solver with a constant mesh density function and HYPRE preconditioner on Graham cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.009	1.008	1.059	1.031	1.053	1.041	1.670	1.071
251	1.010	1.094	1.054	1.036	1.062	1.557	1.126	1.127
501	1.018	1.007	1.013	1.071	1.040	1.035	1.034	1.099
1001	1.011	1.036	1.029	1.011	1.075	1.021	1.130	2.225
2001	1.095	1.004	1.015	1.008	1.018	1.051	1.616	1.008
4001	1.008	1.057	1.059	1.039	1.014	1.002	1.279	1.068
8001	1.282	1.101	1.065	1.074	1.072	1.281	1.102	1.022

Table B.13: Maximum time over minimum time for the single domain solver with a constant mesh density function and HYPRE preconditioner on Graham cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.000	2.413	4.400	9.162	22.697	81.703	203.725	515.436
251	1.000	1.939	3.971	8.281	10.688	33.249	80.608	202.828
501	1.000	1.905	3.208	5.458	13.265	29.813	62.881	149.170
1001	1.000	1.646	2.381	3.183	5.928	13.362	22.863	51.302
2001	1.000	1.714	2.356	2.636	3.696	6.886	12.490	20.240
4001	1.000	1.871	2.652	2.796	3.545	5.252	7.670	13.678
8001	1.000	2.120	3.228	3.488	3.964	5.464	6.398	8.694

Table B.14: Time scaling over the domain for the single domain solver with a constant mesh density function and HYPRE preconditioner on Graham cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.522	1.162	1.194	1.254	1.100	0.998	0.964	1.009
251	1.523	1.215	0.933	0.697	1.135	1.106	1.263	1.844
501	1.503	1.105	0.864	0.822	0.858	0.522	0.479	0.530
1001	1.378	1.156	1.061	0.922	0.955	0.907	1.181	0.848
2001	1.378	1.247	1.166	1.142	1.064	1.053	0.995	1.128
4001	1.355	1.328	1.298	1.280	1.186	1.110	0.992	0.800
8001	1.251	1.429	1.447	1.369	1.326	1.200	1.080	0.924

Table B.15: Double PETSc example code time over single domain solver time with HYPRE preconditioner on Graham cluster.

size \ cores	1	2	4	8	16	32	64	128
126	0.728	0.607	1.205	1.853	2.937	4.352	5.326	6.631
251	0.951	0.737	0.843	1.022	3.118	3.390	5.735	6.757
501	1.320	0.935	1.110	1.146	1.073	1.433	2.407	3.840
1001	1.619	1.532	1.487	1.553	2.114	1.717	2.424	3.951
2001	1.563	1.600	1.592	2.014	2.555	2.998	2.925	4.053
4001	1.582	1.588	1.464	1.753	2.509	3.031	3.658	3.342
8001	1.603	1.588	1.472	1.568	2.135	2.684	3.465	3.817

Table B.16: Torngat cluster time over Graham cluster time for the single domain solver with HYPRE preconditioner.

B.2 GAMG preconditioner

B.2.1 Torngat cluster

PETSc KSP ex12

size \ cores	1	2	4	8	16	32	64	128
126	0.083	0.076	0.101	0.154	0.224	0.414	0.912	1.173
251	0.322	0.243	0.227	0.264	0.449	0.746	1.087	1.921
501	1.393	0.951	0.688	0.601	0.717	1.052	1.784	2.187
1001	6.270	4.461	2.826	2.139	1.853	1.951	2.323	3.159
2001	27.499	19.055	10.943	7.529	6.343	4.945	3.636	3.278
4001	124.602	81.030	45.239	26.831	22.662	17.131	10.361	7.165
8001	524.570	354.093	186.030	111.700	84.266	65.065	35.761	22.000

Table B.17: Minimized timing results for the PETSc example code with GAMG preconditioner on Torngat cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.009	1.028	1.740	1.654	2.306	1.858	1.380	1.406
251	1.006	1.012	1.037	2.721	1.086	1.606	1.817	1.200
501	1.004	1.042	1.237	1.442	1.637	1.396	1.586	1.623
1001	1.003	1.008	1.391	1.157	1.229	1.289	1.180	1.078
2001	1.003	1.026	1.116	1.069	1.052	1.162	1.219	1.296
4001	1.005	1.022	1.051	1.217	1.110	1.036	1.096	1.067
8001	1.116	1.006	1.043	1.048	1.011	1.077	1.029	1.024

Table B.18: Maximum time over minimum time for the PETSc example code with GAMG preconditioner on Torngat cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.000	1.821	4.828	14.755	43.009	158.756	700.018	1801.069
251	1.000	1.509	2.824	6.571	22.338	74.207	216.041	763.872
501	1.000	1.365	1.976	3.453	8.231	24.148	81.932	200.891
1001	1.000	1.423	1.803	2.729	4.729	9.959	23.713	64.501
2001	1.000	1.386	1.592	2.190	3.691	5.755	8.462	15.258
4001	1.000	1.301	1.452	1.723	2.910	4.400	5.322	7.360
8001	1.000	1.350	1.419	1.703	2.570	3.969	4.363	5.368

Table B.19: Time scaling over the domain for the PETSc example code with GAMG preconditioner on Torngat cluster.

Single domain solver with $\rho = 1$

size \ cores	1	2	4	8	16	32	64	128
126	0.144	0.168	0.237	0.516	0.771	1.992	3.329	5.064
251	0.578	0.452	0.489	0.610	0.985	1.796	3.308	5.611
501	2.788	1.834	1.471	1.301	1.571	2.807	3.896	6.719
1001	13.792	8.821	5.839	4.352	4.666	3.853	5.732	8.405
2001	64.660	40.502	22.822	16.270	14.377	12.498	10.214	12.965
4001	292.178	189.207	104.127	67.685	53.235	41.978	32.145	26.047
8001	1321.220	796.146	452.596	271.663	224.124	192.688	116.291	82.385

Table B.20: Minimized timing results for the single domain solver with a constant mesh density function and GAMG preconditioner on Torngat cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.027	1.068	1.760	1.355	1.511	1.250	1.162	1.076
251	1.006	1.068	1.653	1.469	1.498	1.495	1.303	1.090
501	1.004	1.035	1.241	1.073	1.371	1.210	1.297	1.094
1001	1.004	1.075	1.128	1.204	1.127	1.213	1.298	1.161
2001	1.131	1.016	1.058	1.082	1.120	1.049	1.142	1.046
4001	1.062	1.022	1.028	1.045	1.041	1.139	1.062	1.030
8001	1.239	1.029	1.018	1.030	1.065	1.062	1.021	1.018

Table B.21: Maximum time over minimum time for the single domain solver with a constant mesh density function and GAMG preconditioner on Torngat cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.000	2.334	6.572	28.616	85.524	442.172	1477.655	4495.508
251	1.000	1.565	3.385	8.445	27.270	99.504	366.483	1243.184
501	1.000	1.316	2.111	3.735	9.018	32.221	89.454	308.518
1001	1.000	1.279	1.693	2.524	5.413	8.939	26.599	78.004
2001	1.000	1.253	1.412	2.013	3.558	6.185	10.109	25.665
4001	1.000	1.295	1.426	1.853	2.915	4.598	7.041	11.411
8001	1.000	1.205	1.370	1.645	2.714	4.667	5.633	7.981

Table B.22: Time scaling over the domainfor the single domain solver with a constant mesh density function and GAMG preconditioner on Torngat cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.157	0.902	0.850	0.596	0.582	0.415	0.548	0.463
251	1.114	1.074	0.930	0.867	0.913	0.831	0.657	0.685
501	1.000	1.037	0.936	0.924	0.913	0.749	0.916	0.651
1001	0.909	1.012	0.968	0.983	0.794	1.013	0.811	0.752
2001	0.851	0.941	0.959	0.925	0.882	0.791	0.712	0.506
4001	0.853	0.857	0.869	0.793	0.851	0.816	0.645	0.550
8001	0.794	0.890	0.822	0.822	0.752	0.675	0.615	0.534

Table B.23: Double PETSc example code time over single domain solver time with GAMG preconditioner on Torngat cluster.

B.2.2 Graham cluster

PETSc KSP ex12

size \ cores	1	2	4	8	16	32	64	128
126	0.130	0.101	0.073	0.060	0.064	0.102	0.124	0.158
251	0.408	0.323	0.189	0.120	0.099	0.130	0.136	0.189
501	1.528	1.040	0.583	0.317	0.250	0.245	0.184	0.194
1001	6.126	4.018	2.171	1.189	0.770	0.613	0.481	0.436
2001	25.639	16.604	8.479	4.563	2.794	2.217	1.269	0.938
4001	110.714	68.192	34.926	18.586	11.392	8.731	4.455	2.514
8001	466.457	285.306	146.127	79.358	46.488	36.708	18.388	9.246

Table B.24: Minimized timing results for the PETSc example code with GAMG preconditioner on Graham cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.012	1.124	1.132	1.092	1.030	1.144	2.570	1.108
251	1.012	1.011	1.063	1.195	1.033	1.078	3.535	1.013
501	1.023	1.001	1.033	1.133	1.028	1.048	1.124	1.751
1001	1.001	1.005	1.002	1.006	1.032	1.067	1.045	1.056
2001	1.004	1.003	1.002	1.022	1.021	1.017	1.103	1.872
4001	1.005	1.000	1.003	1.001	1.007	1.008	1.033	1.010
8001	1.199	1.090	1.096	1.076	1.097	1.119	1.009	1.603

Table B.25: Maximum time over minimum time for the PETSc example code with GAMG preconditioner on Graham cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.000	1.558	2.237	3.672	7.874	25.004	60.767	155.204
251	1.000	1.581	1.854	2.357	3.884	10.218	21.361	59.248
501	1.000	1.361	1.527	1.661	2.618	5.140	7.713	16.270
1001	1.000	1.312	1.418	1.553	2.010	3.202	5.027	9.104
2001	1.000	1.295	1.323	1.424	1.743	2.766	3.167	4.684
4001	1.000	1.232	1.262	1.343	1.646	2.523	2.575	2.907
8001	1.000	1.223	1.253	1.361	1.595	2.518	2.523	2.537

Table B.26: Time scaling over the domain for the PETSc example code with GAMG preconditioner on Graham cluster.

size \ cores	1	2	4	8	16	32	64	128
126	0.640	0.748	1.382	2.573	3.498	4.066	7.377	7.431
251	0.789	0.753	1.202	2.200	4.538	5.730	7.979	10.172
501	0.912	0.914	1.180	1.895	2.866	4.284	9.685	11.258
1001	1.023	1.110	1.302	1.798	2.408	3.183	4.828	7.251
2001	1.073	1.148	1.291	1.650	2.270	2.231	2.865	3.494
4001	1.125	1.188	1.295	1.444	1.989	1.962	2.326	2.850
8001	1.125	1.241	1.273	1.408	1.813	1.773	1.945	2.379

Table B.27: Torngat cluster time over Graham cluster time for the PETSc example code with GAMG preconditioner.

Single domain solver with $\rho = 1$

size \ cores	1	2	4	8	16	32	64	128
126	0.164	0.152	0.126	0.099	0.111	0.189	0.239	0.312
251	0.533	0.419	0.319	0.276	0.192	0.207	0.255	0.359
501	1.986	1.377	0.873	0.544	0.545	0.490	0.463	0.540
1001	8.674	5.568	3.205	1.850	1.330	1.222	0.880	0.802
2001	35.273	23.471	13.165	7.295	4.480	3.794	2.594	1.685
4001	155.573	100.836	55.139	29.154	18.129	14.130	8.012	5.947
8001	685.254	430.775	233.815	123.058	75.546	NaN	31.672	20.537

Table B.28: Minimized timing results for the single domain solver with a constant mesh density function and GAMG preconditioner on Graham cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.089	1.023	1.103	1.256	1.026	1.874	1.079	1.944
251	1.113	1.018	1.009	1.051	1.050	1.807	1.057	3.473
501	1.004	1.013	1.075	1.159	1.032	1.024	1.060	1.024
1001	1.013	1.019	1.021	1.017	1.019	1.103	1.152	1.153
2001	1.150	1.083	1.066	1.033	1.065	1.046	1.020	1.065
4001	1.020	1.012	1.001	1.012	1.012	1.009	1.019	1.372
8001	1.191	1.081	1.070	1.060	1.082	NaN	1.125	1.311

Table B.29: Maximum time over minimum time for the single domain solver with a constant mesh density function and GAMG preconditioner on Graham cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.000	1.852	3.053	4.798	10.767	36.873	93.053	242.771
251	1.000	1.571	2.393	4.136	5.755	12.429	30.632	86.222
501	1.000	1.387	1.759	2.191	4.389	7.904	14.937	34.825
1001	1.000	1.284	1.478	1.706	2.453	4.509	6.492	11.837
2001	1.000	1.331	1.493	1.655	2.032	3.442	4.707	6.113
4001	1.000	1.296	1.418	1.499	1.865	2.906	3.296	4.893
8001	1.000	1.257	1.365	1.437	1.764	NaN	2.958	3.836

Table B.30: Time scaling over the domain for the single domain solver with a constant mesh density function and GAMG preconditioner on Graham cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.584	1.332	1.161	1.212	1.158	1.074	1.034	1.013
251	1.530	1.540	1.185	0.872	1.033	1.258	1.067	1.051
501	1.539	1.511	1.336	1.167	0.918	1.001	0.795	0.719
1001	1.412	1.443	1.355	1.286	1.158	1.003	1.094	1.086
2001	1.454	1.415	1.288	1.251	1.247	1.168	0.978	1.114
4001	1.423	1.353	1.267	1.275	1.257	1.236	1.112	0.846
8001	1.361	1.325	1.250	1.290	1.231	NaN	1.161	0.900

Table B.31: Double PETSc example code time over single domain solver time with GAMG preconditioner on Graham cluster.

size \ cores	1	2	4	8	16	32	64	128
126	0.877	1.105	1.887	5.230	6.965	10.516	13.925	16.238
251	1.083	1.079	1.533	2.212	5.133	8.673	12.960	15.619
501	1.404	1.332	1.685	2.394	2.884	5.723	8.408	12.437
1001	1.590	1.584	1.822	2.353	3.510	3.153	6.515	10.478
2001	1.833	1.726	1.734	2.230	3.209	3.294	3.937	7.696
4001	1.878	1.876	1.888	2.322	2.936	2.971	4.012	4.380
8001	1.928	1.848	1.936	2.208	2.967	NaN	3.672	4.011

Table B.32: Torngat cluster time over Graham cluster time for the single domain solver with a constant mesh density function and GAMG preconditioner.

B.3 LU preconditioner

B.3.1 Torngat cluster

PETSc KSP ex12

size \ cores	1	2	4	8	16	32	64	128
126	0.053	0.180	0.168	0.170	0.198	0.296	0.412	0.748
251	0.308	0.636	0.600	0.611	0.686	0.793	1.060	1.626
501	1.997	2.737	2.377	2.381	2.536	3.150	3.421	5.156
1001	15.427	13.762	10.580	9.637	8.854	11.120	14.790	17.729
2001	129.915	81.771	54.089	44.716	38.814	45.689	59.667	80.055
4001	1024.911	469.099	280.933	233.907	199.934	197.806	248.694	339.636
8001	NaN	4530.985	2525.723	1782.601	1388.885	1143.866	1388.145	1638.056

Table B.33: Minimized timing results for the PETSc example code with LU preconditioner on Torngat cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.060	1.019	1.023	1.163	1.245	1.044	1.014	1.311
251	1.014	1.027	1.142	1.050	1.023	1.053	1.009	1.048
501	1.003	1.006	1.067	1.046	1.039	1.034	1.112	1.026
1001	1.005	1.024	1.016	1.024	1.043	1.055	1.045	1.047
2001	1.002	1.006	1.007	1.012	1.040	1.036	1.022	1.030
4001	1.119	1.010	1.008	1.013	1.006	1.019	1.068	1.010
8001	NaN	1.065	1.037	1.022	1.028	1.049	1.018	1.033

Table B.34: Maximum time over minimum time for the PETSc example code with LU preconditioner on Torngat cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.000	6.829	12.786	25.764	60.323	179.936	500.749	1819.382
251	1.000	4.125	7.789	15.862	35.610	82.333	220.084	675.343
501	1.000	2.741	4.761	9.538	20.321	50.478	109.618	330.499
1001	1.000	1.784	2.743	4.997	9.182	23.065	61.358	147.093
2001	1.000	1.259	1.665	2.754	4.780	11.254	29.394	78.876
4001	1.000	0.915	1.096	1.826	3.121	6.176	15.530	42.417
8001	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Table B.35: Time scaling over the domain for the PETSc example code with LU preconditioner on Torngat cluster.

B.3.2 Graham cluster

PETSc KSP ex12

size \ cores	1	2	4	8	16	32	64	128
126	0.084	0.202	0.178	0.149	0.147	0.170	0.211	0.253
251	0.368	0.741	0.652	0.586	0.662	0.654	0.723	0.859
501	1.930	2.763	2.444	2.219	3.951	4.052	3.699	3.709
1001	12.908	12.124	10.646	9.539	11.121	13.296	16.345	17.447
2001	111.845	54.441	47.710	43.218	41.603	53.000	87.557	99.126
4001	878.439	261.752	226.574	206.416	198.690	199.045	211.430	271.434
8001	NaN	1212.911	1035.853	905.608	859.578	906.572	1103.333	929.694

Table B.36: Minimized timing results for the PETSc example code with LU preconditioner on Graham cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.010	1.014	1.020	1.051	1.008	1.762	1.038	1.552
251	1.028	1.014	1.010	1.021	1.015	1.364	1.242	1.163
501	1.012	1.022	1.008	1.007	1.003	1.009	1.069	1.058
1001	1.008	1.009	1.006	1.009	1.007	1.005	1.034	1.247
2001	1.013	1.002	1.015	1.001	1.001	1.001	1.015	1.067
4001	1.010	1.008	1.009	1.004	1.006	1.035	1.079	1.053
8001	NaN	1.173	1.189	1.164	1.160	1.195	1.265	1.520

Table B.37: Maximum time over minimum time for the PETSc example code with LU preconditioner on Graham cluster.

size \ cores	1	2	4	8	16	32	64	128
126	1.000	4.809	8.505	14.234	28.081	64.786	160.981	386.177
251	1.000	4.031	7.092	12.746	28.818	56.955	125.958	299.049
501	1.000	2.863	5.065	9.198	32.757	67.186	122.668	246.048
1001	1.000	1.879	3.299	5.912	13.786	32.963	81.046	173.021
2001	1.000	0.973	1.706	3.091	5.951	15.164	50.102	113.444
4001	1.000	0.596	1.032	1.880	3.619	7.251	15.404	39.551
8001	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Table B.38: Time scaling over the domain for the PETSc example code with LU preconditioner on Graham cluster.

size \ cores	1	2	4	8	16	32	64	128
126	0.627	0.891	0.943	1.135	1.348	1.742	1.951	2.955
251	0.838	0.858	0.921	1.043	1.036	1.212	1.465	1.893
501	1.035	0.991	0.973	1.073	0.642	0.778	0.925	1.390
1001	1.195	1.135	0.994	1.010	0.796	0.836	0.905	1.016
2001	1.162	1.502	1.134	1.035	0.933	0.862	0.681	0.808
4001	1.167	1.792	1.240	1.133	1.006	0.994	1.176	1.251
8001	NaN	3.736	2.438	1.968	1.616	1.262	1.258	1.762

Table B.39: Torngat cluster time over Graham cluster time for the PETSc example code with LU preconditioner.