PARALLEL PRECONDITIONERS FOR AN OCEAN MODEL IN CLIMATE SIMULATIONS

Zur Erlangung des akademischen Grades eines

DOKTORS DER NATURWISSENSCHAFTEN

von der Fakultät für Mathematik des
Karlsruher Instituts für Technologie (KIT)
genehmigte

DISSERTATION

von
Dipl.-Math. Florian Wilhelm
aus
Neustadt a. d. Weinstraße

Tag der mündlichen Prüfung: 25. Januar 2012

Referent: Prof. Dr. Vincent Heuveline
Korreferent: Jun.-Prof. Dr. Jan-Philipp Weiß

dedicated to my parents

meinen Eltern gewidmet

ABSTRACT

In this work the utilization of various parallel preconditioner techniques to improve the performance of linear solvers in the field of high-performance computing (HPC) is examined. The major focus lies on the evaluation of different solvers and preconditioners for the barotropic subsystem of the ocean/sea-ice model MPIOM as well as on the development of a numerical solver library as part of the *Scalable-Earth-System-Models for high productivity climate simulations* project. Furthermore, we extend techniques from the field of *support theory* to derive upper bounds for the number of iterations the conjugate gradient methods needs when a block-Jacobi *Steiner graph* preconditioner is used. Based on this, a model for a hardware-aware preconditioner is described that takes into account the topology of the network interconnection. In the field of reconfigurable computing we analyze the application of a high-level approach for programming preconditioners on *Field Programmable Gate Arrays* (FPGAs) as accelerators for HPC with the help of a C to hardware language converter technology.

ZUSAMMENFASSUNG

In dieser Arbeit werden verschiedene parallele Vorkonditionierungstechniken zur Verbesserung der Leistung von linearen Lösern im Bereich des Hochleistungsrechnens (HPC) untersucht. Der Hauptfokus liegt hierbei auf der Analyse verschiedener Löser und Vorkonditionierer für das barotropische Subsystem des Ocean/Meereismodells MPIOM sowie auf der Entwicklung einer numerischen Bibliothek als Teil des *Scalable-Earth-System-Models for high productivity climate simulations* Projekts. Darüber hinaus werden Techniken aus dem Bereich der *Support Theorie* angewendet um obere Schranken für die Anzahl der Iterationen der Methode der konjugierten Gradienten zu berechnen falls ein *Steiner Graph* Vorkonditionierer verwendet wird. Darauf aufbauend wird ein Modell für einen Vorkonditionierer beschrieben, der sich an die Gegebenheiten der Hardware, d.h. der Netzwerktopologie, anpasst. Im Bereich der rekonfigurierbaren Rechensysteme analysieren wir den Einsatz einer C ähnlichen Hochsprache zur Programmierung von Vorkonditionierern auf *Field Programmable Gate Arrays* (FPGAs) als Beschleuniger für HPC Systeme unter Zuhilfenahme eines C zu VHDL Konverters.

If I have seen further it is only by standing on the shoulders of giants.

— Sir Isaac Newton, 1676

ACKNOWLEDGMENTS

My special thanks are devoted to Prof. Dr. Vincent Heuveline who made this work possible at all and put me in the position to participate in the ScalES project. I am grateful for his enduring support and the many fruitful discussions we had. Also, I want to express my special thanks to Jun.-Prof. Dr. Jan-Philipp Weiß for his kind suggestions and helpful remarks regarding my work. Furthermore, I am indebted to Ioannis Koutis, Assistant Professor in the Computer Science Department at the University of Puerto Rico, for his guidance and hints regarding the domain of support theory.

I also want to extend my thanks to my colleagues in the working group *Numerical Simulation, Optimization and High Performance Computing* as well as my project partners in the ScalES project for their support, many helpful discussions and the wonderful atmosphere.

CONTENTS

1 INTRODUCTION

1.1 CLIMATE PROJECTIONS FOR THE IPCC

Climate change and anthropogenic influence on it are one of the most important issues mankind has to deal with in this century. It is an largely accepted fact that world climate is changing because of higher CO_2 emissions since the beginning of the second industrial revolution in mid 19^{th} century until today [72, 135]. The first consequences of global warming, like more frequent storms, droughts and glacier melt appear worldwide and raise the question how climate will change in the future due to man-made greenhouse gas emissions [73]. The significance of this field of research is obvious, given the fact that the United Nations Environment Programme (UNEP) established the Intergovernmental Panel on Climate Change (IPCC) in November 1988 to coordinate global research efforts in climate change and to provide an IPCC Assessment Report (AR) regularly. The crucial part of this report consists of climate projections provided by simulations based on possible future scenarios of CO_2 emissions which are of paramount importance to assess the consequences of economic decisions now to be taken.

The outcome of two climate projections based on two different IPCC AR future scenarios B1 and A1B for the year 2100 are demonstrated in Figure 1. In A1B, it is assumed that the global economy grows at a high rate and the CO_2 emissions with it. The world population is assumed to increase until 2050, followed by a decrease. In scenario B1 the same assumptions as in A1B are made with the difference that the economy becomes more information and service oriented which leads to less emissions as well as resource consumption.

Climate simulations are usually based on highly complex systems of coupled numerical models originating from different scientific domains [72, chapter 8]. Global climate models are typically structured into sub-models of the different components of the earth system, i.e., atmosphere, ocean, sea-ice, atmospheric chemistry, land, ocean biogeochemistry and others. Commonly, sub-models use different kinds of methods to describe physical processes for example partial differential equations in the ocean and atmosphere for fluid movement or ordinary differential equations for chemistry. Besides, often different time-scales and computational grids are applied in sub-models to adequately resolve continuous physical processes in discretized space and time.

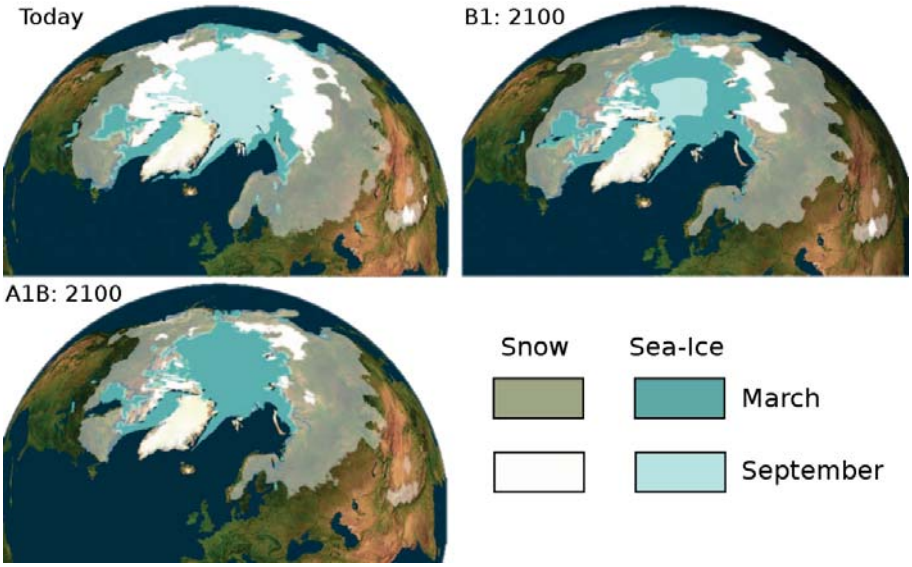


Figure 1: Comparison of the simulated sea-ice coverage from today in March and September with the climate projections of the scenarios B1 and A1B in 2100 [1] simulated with ECHAM5 and MPIOM which are described in the following sections. This image is courtesy of DKRZ and MPI-M.

Aspects of software engineering are adding up to the theoretical complexity of climate models. Today, many climate models consist of several million lines of historically developed code, mostly Fortran, and have a typical life time of at least two decades. Sub-models tend to use custom data partitionings and structures adapted to the hardware. Thus, during model simulations, very large amounts of data constantly need to be transformed between different data representations and to be exchanged between sub-models. Another challenge for software engineers is the fact that advancements in computer architectures need to be continually integrated into existing code structures to be utilized.

Enabled by the increase of compute power, climate scientists employ finer computational grids, smaller time steps, and include additional physical processes to further improve quality and reliability. These improvements are limited by available compute power, but are needed for urgent scientific goals, e.g., prediction of regional changes, adequate simulation of clouds and precipitation to overcome the uncertainty of climate models; the most expensive of which is finer resolution [136]. A cloud resolving global atmosphere model, for example, requires a grid spacing of the order of magnitude of 10 km which means several 1000 times as many grid points as can be realized in today's models. At the same time the number of necessary time steps which need to be computed for a given simulated period increases tenfold at least.

Such models will require several petaflops of sustained computational speed and hundreds of terabytes of main memory. The performance of future (peta- and exascale) computers will presumably not

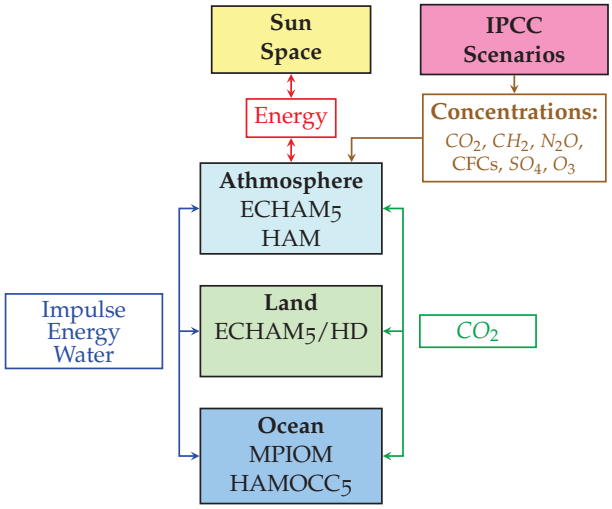


Figure 2: Schematic illustration of the COSMOS model used in the IPCC report. ECHAM5 includes an atmosphere model as well as a model for the surface of the earth with an outlet (HD) and an aerosol (HAM) model. MPIOM models the oceans with sea-ice and HAMOCC5 the marine biochemistry. The coupling between these three models is managed by OASIS3. Besides anthropogenic influences, also naturally occurring processes (e.g., volcanoes, variations of insolation) can be taken into account [1].

be reached by increasing processor speed but by massively increased parallelism only, i.e., climate models will "require unprecedented levels of parallel scalability in all components" [136]. This implies significant software engineering considerations. For overall scalability of a global climate model each component of the model in itself as well as the coupling and communication between the model components and also the I/O need to be scalable. Each component may need a special distribution and partitioning of data to guarantee optimal utilization of local fast memory, minimal global communication and optimal load balancing.

1.2 THE SCALES PROJECT AND ITS GOALS

The project "Scalable-Earth-System-Models for high productivity climate simulations" (ScalES) was funded from 2009 to 2011 by the German "Bundesministerium für Bildung und Forschung" (BMBF No. 01IH08004E) to address these design issues. Central goal of the project is to provide generic solutions for problems and tasks common to all climate and earth system models in form of a library that can easily be used by model developers to increase scalability and efficiency of their models. As a test suite and development platform, one of the global climate models being used for the simulations of the IPCC ARs [72] was used; the COSMOS model system [26], encompassing a global ocean/sea-ice model (MPIOM) [91], an atmosphere model

(ECHAM) [118, 128] and the OASIS coupler [114]. Figure 2 illustrates the interplay between these models and their inputs defined by IPCC scenarios.

The scalability bottlenecks of COSMOS were analyzed by the project partners and several recurring patterns were identified: Many components of COSMOS solve stencil based systems of linear equations obtained by finite difference or finite volume discretization, like the barotropic subsystem in the ocean model. Another recurring task is the determination of a proper partitioning based on the current computational workload to assure a good load balance. For instance, an optimal grid partitioning for the fluid dynamics of the atmosphere model ECHAM may not suit the ocean component well, which, in case of the Max-Planck-Institute ocean model (MPIOM), has many "dry" grid points defined over land. An obvious conclusion that comes with these observations is the need for tools to easily and efficiently distribute and transpose data between processes.

The goals and challenges of the ScalES project can be summarized as following:

- fast parallel storage and access of large data sets,
- efficient, parallel coupling of the model's components,
- dynamic load balancing to evenly distribute the workload,
- efficient usage of the given hardware architecture by modern mathematical methods.

The ScalES consortium is formed by the Deutsches Klimarechenzentrum GmbH (DKRZ) in Hamburg which is taking the coordination role, the Alfred-Wegener-Institute für Polar und Meeresforschung (AWI) in Bremerhaven, the Max-Planck-Institut für Meteorologie (MPI-M) in Hamburg, the Max-Planck-Institut für Chemie (MPI-C) in Mainz, IBM Germany and the Engineering Mathematics and Computing Lab (EMCL) at the Karlsruhe Institute of Technology (KIT). The task of the EMCL is the performance improvement of MPIOM on High-Performance Computing (HPC) hardware architectures by means of improved preconditioned linear solvers to reduce the necessary number of iterations and thereby the communication costs. Modern programming techniques are applied in order to develop an efficient solver library tailored to the special needs of legacy earth-system-model components like MPIOM.

1.3 THE MPI-OCEAN/SEA-ICE MODEL

MPIOM is an Ocean General Circulation Model (OGCM) based on the ocean primitive equations on a curvilinear C-grid with z-coordinates and free surface [91]. The first version of MPIOM was released in

1997 as a serial program written in Fortran 77. In 2000 the code was then parallelized for the NEC SX6 vector parallel supercomputer [103], accompanied by a switch to Fortran 90. Today the code encompasses roughly 40,000 lines of Fortran 90/95 code and uses the Message Passing Interface (MPI) library [51] for parallelization.

It is the successor of the Hamburg Ocean Primitive Equation (HOPE) model [142]. While the horizontal discretization in HOPE was based on a staggered Arakawa E-grid [7], MPIOM makes use of a curvilinear C-grid [7]. There are mainly two reasons for this transition. Firstly, the C-grid is computationally more efficient than the staggered E-grid, because a higher horizontal resolution can be achieved with the same number of grid points. Secondly, the E-grid model required additional horizontal numerical diffusion in order to achieve convergence. MPIOM has been applied in numerous scientific studies investigating different aspects of the ocean/sea-ice dynamics and the ocean's role in Earth System dynamics. Simulations with the coupled ESM ECHAM-MPIOM have contributed to the IPCC AR4 and will also provide data for IPCC AR5.

Although MPIOM can be considered "old fashioned" with regard to the mathematical methods (like finite differences) and programming techniques, it is an validated, heavily used ocean model encompassing many aspects of the ocean physics and therefore bearing valuable and extensive legacy code.

1.4 OUTLINE

The thesis is organized as follows. Following this introduction and motivation from **Chapter 1**, we present in the first section of **Chapter 2** an introduction to fluid dynamics by deriving the Navier-Stokes equations as well as the advection-diffusion equation. Based on this, we develop in Section 2.2 the mathematical model behind MPIOM, the so called ocean primitive equations. In Section 2.3, we derive the barotropic subsystem and its discretization in the succeeding section.

Chapter 3 introduces the solver for the barotropic subsystem that was used by MPIOM when the ScalES project started. The following sections describe solvers and preconditioners that were implemented, as part of our contribution to the ScalES project, in order to improve the scalability and performance of MPIOM. The Chapter closes with numerical experiments and their analysis to evaluate the presented solvers and preconditioners.

Chapter 4 presents a largely unknown technique to construct and analyze algebraic preconditioners; the *support theory*. This theory is particularly applicable to the diagonal dominant Stieltjes matrices of the discretized barotropic subsystem. Therefore we study and extend this theory to analyze a special kind of block-Jacobi preconditioners. In the first two sections an introduction to graph theory is given and a

bridge to symmetric, diagonal dominant matrices is established. Section 4.3 elaborates the fundamental definition and theorems of support theory followed by additions to this field of research. Section 4.5 gives an introduction to flows in a network and presents an algorithm for a special type of network flow problems. The final section gives an outlook of how the techniques developed in this chapter can be used to construct preconditioners that are aware of the computer cluster's network topology.

Chapter 5 tackles the subject of preconditioning from a hardware perspective. Herein, the usage of reconfigurable computing as accelerators for preconditioners is investigated. This is necessary since new technologies like reconfigurable computing pose opportunities for HPC but special considerations need to be taken regarding their utilization. To evaluate this technology we solve a Laplace-like problem which can be understood as a simplified barotropic subsystem. After a short introduction to reconfigurable computing, we show how this technology can be accessed with the help of a converter that translates C code to a hardware description language which is needed to program reconfigurable hardware. The potential of this approach is examined by solving a model problem with the conjugate gradient method that is accelerated by a preconditioner running on reconfigurable hardware. The chapter closes with a thorough analysis of some numerical experiments and an outlook regarding this technology.

Chapter 6 concludes this thesis by summarizing the results obtained so far as well as providing an outlook to further topics for additional studies.

Appendix A discusses other developments in the ScalES project besides the solvers and preconditioners from Chapter 3. On the one hand, we give a short overview of the UniTrans library for transposition and exchange of data in climate models and, on the other hand, of a hierarchical partitioner for dynamic load balancing.

Appendix B lists the configuration details of the Blizzard cluster at the DKRZ and elaborates on his main features.

2 MATHEMATICAL MODELS

Nowadays we are used to reckon with weather forecasts which are reliable to some extent and therefore highly relevant for our modern society. The same is true for climate projections that help to assess the anthropological impact on our climate over a century. For the simulation of such physical phenomena mathematical models are necessary that adequately capture the behavior of physical processes. At the same time, a model needs to be simple and reduced to a minimal description of the influential parameters and their relationships which cause the physical effects that are of interest. In order to reduce the complexity of natural processes to a mathematical model, assumptions need to be taken that contribute to the characteristics of a model, e.g., its accuracy and its scope of application. It is highly important to know all assumptions, simplifications and conditions that were applied to derive a model since only with this knowledge the results obtained by this model can be scientifically interpreted. Besides the pure knowledge of this, also the certainty is indispensable that all aspects of a model, e.g., the boundary conditions, are sound in physical terms to allow conclusions about the physical processes that are described by the model. Only then, numerical mathematics can be employed which further shapes the characteristics of a *numerical model* by applying methods and techniques to discretize and solve a model that often has no closed-form solution.

Following this idea, we will rigorously elaborate in this chapter the fundamental mathematical models that are employed in the ocean model of the Max-Planck Institute [91] (MPIOM) to lay the basis for our numerical considerations in the following chapters. We will do this in a mathematically strict and precise way by clearly stating all assumptions and employed simplifications in MPIOM which has not been available in this form before.

In MPIOM, as in most atmospheric and oceanic models, a set of nonlinear partial differential equations is used to describe changes of state in space and time due to flows [52, 57, 70]. The most basic equations found in almost every climate model are derived from physical axioms, i.e., conservation of momentum, mass and energy. They were first written down by Vilhelm Bjerknes for the atmosphere and are called *primitive equations* [99]. The first numerical approaches to solve these equations were undertaken by Richardson [116] in 1922 which can be seen as the advent of modern weather and climate forecast [89].

From this short historical note, we start with an elementary introduction to fluid dynamics following standard references [8, 14, 46, 101, 113, 140]. A nomenclature of the terminology used in this chapter can be found on page 131.

2.1 BASIC EQUATIONS OF FLUID DYNAMICS

From a physical point of view, a fluid is composed of an extremely large number of interacting molecules which in their entirety define the velocity, density and other properties of the fluid. This *microscopical view* is an inherently discrete way of examining and describing a fluid since the properties of every atomic element of a fluid are taken into account. In contrast, fluid dynamics is concerned with the fluid as a whole leading to a *macroscopical view* where a fluid is considered a homogeneous continuum that exhibits its physical properties at every point. We state this notion in the following hypothesis that is intrinsic to fluid dynamics.

Assumption 1 (Continuum hypothesis)**.** *Physical properties (e.g. density, temperature, ...) are well-defined, continuous functions (e.g.* ρ*,* T*, ...) on a domain* $\Omega \subset \mathbb{R}^3$*.*

2.1.1 *Lagrangian and Eulerian Specification*

We further assume a time invariant Cartesian coordinate system on a domain $\Omega \subset \mathbb{R}^3$ in order to identify a point in Ω with $x = (x_i)_{i=1}^3$. The fluid in Ω is defined by a non-empty, connected set $\Omega_t \subset \Omega$ of *material points* $\xi \in \Omega_t$ that is dependent on time $t \in [0, \infty)$. We assume that the position x of ξ changes smoothly in t so that its movement can be represented by a sufficiently continuously differentiable function $x = x(\xi, t)$. Additionally we require that each point $x \in \Omega$ is occupied by at most one material point ξ at any given time t so that $x = x(\xi, t)$ is invertible. To identify ξ we take its position at time $t_0 = 0$ and write $\xi := x(\xi, 0)$. The velocity $v = (v_i)_{i=1}^3$ of a material point ξ at point x and time t is given by $v(x, t) = v(x(\xi, t), t) = \partial_t x(\xi, t)$.

Describing the configuration of a fluid by means of Cartesian coordinates x is called *Eulerian specification* of a fluid whereas the description by the position $x(\xi, t)$ of a material point ξ at time t is called *Lagrangian specification* of a fluid.

2.1.2 *Transport Theorem*

Let ϕ be a physical quantity (e.g. density, temperature, ...) of a material point ξ at point x and time t. The relation $\phi = \phi(x, t) = \phi(x(\xi, t), t)$ links the Eulerian and Lagrangian specification. The local change in time t of ϕ at a fixed point x is $\partial_t \phi = \partial_t \phi(x, t)$ in the

Eulerian specification. In the Lagrangian specification, the movement of ξ needs to be taken into account which leads to the definition of the *total derivative* (or *material derivative*)

$$\frac{d}{dt}\phi = d_t\phi = d_t\phi(x(\xi,t),t) = \partial_t\phi + v\cdot\nabla_x\phi \tag{2.1}$$

which describes the local change of ϕ at a material point ξ.

Considering a material volume $V(t)$, the total amount of the quantity ϕ contained in $V(t)$ is determined by $\int_{V(t)}\phi\,dx$. The following theorem states the relation of the change in time (or transport) of a material volume's quantity to the change of ϕ and v.

Theorem 2 (Transport theorem)**.** *Let $\phi = \phi(x,t)$ be a sufficiently smooth and scalar function. For a material volume we have that*

$$\frac{d}{dt}\int_{V(t)}\phi\,dx = \int_{V(t)}\partial_t\phi + \nabla\cdot(\phi v)\,dx.$$

Sketch of the proof. Utilization of the requirement that $x = x(\xi,t)$ is invertible and consequently that its functional determinant is positive. This is then used to transform $V(t)$ on the reference volume $V(0)$ where basic transformation rules of differential calculus are applied. See Feistauer [46, p. 29] for details. □

2.1.3 *Conservation of Mass*

Given the density $\rho = \rho(x,t)$ of material points at point x and fixed time t, the mass of a material volume $V = V(t)$ is

$$m(V) = \int_V \rho\,dx.$$

Since the material volume is defined by the same material points at any time we can state the *conservation of mass* as

$$d_t m(V) = 0. \tag{2.2}$$

Applying Theorem 2 yields

$$\int_V \partial_t\rho + \nabla\cdot(\rho v)\,dx = 0 \tag{2.3}$$

for arbitrary V. Using the continuity of the integrand we conclude the *continuity equation*

$$\partial_t\rho + \nabla\cdot(\rho v) = 0 \tag{2.4}$$

which means that inside a material volume no mass is created or destroyed. Assuming a control volume, i.e., a fixed volume with

respect to the space and applying the divergence theorem [93, p. 83] to the second term of the left side of (2.3), we have that

$$\int_V \partial_t \rho \, dx - \int_{\partial V} n \cdot (\rho v) \, dS = 0,$$

where n denotes the unit outer normal to the volume's surface ∂V. This equation states the fact that all changes of the control volume's mass are caused by the flow of mass elements over the boundary of the volume.

2.1.4 *Conservation of Momentum*

Newton's second law of motion states that the rate of change of a material volume's impulse $I(V)$ is equal to the force $F(V)$ acting on V, formally

$$d_t I(V) = F(V). \tag{2.5}$$

The impulse of a volume V is given by the velocity v of its material points and density ρ in V, thus

$$I(V) = \int_V \rho v \, dx.$$

The action on V can be classified into two kinds of forces:

- *Volume forces* $F_{vol}(V)$ like gravitation and inertial force are acting on all material points contained in a volume. Let $f = f(x,t)$ be a vector-valued function, called *density of volume force*, describing the volume force in relation to unit of mass, we have

$$F_{vol}(V) = \int_V \rho f \, dx.$$

- *Surface forces* $F_{sur}(V)$ like pressure which are acting on the surface of the volume. These forces are described by a *density of surface force* $\sigma(x,t)$ in relation to the normal n and area of ∂V. Here, we take the simplest example of a surface force and let $\sigma = (\sigma_{ij})_{i,j=1}^3$ be the *stress tensor* (see [8, p. 101] for details) which lets us write

$$F_{sur}(V) = \int_{\partial V} n \cdot \sigma \, dS.$$

The total force $F(V)$ acting on V is just the sum of those two kinds. Substituting this into (2.5), we have that

$$\frac{d}{dt} \int_V \rho v \, dx = \int_V \rho f \, dx + \int_{\partial V} n \cdot \sigma \, dS. \tag{2.6}$$

Applying the divergence theorem to the second term on the right hand side results in

$$\frac{d}{dt}\int_V \rho v\, dx = \int_V (\rho f + \nabla \cdot \sigma)\, dx. \tag{2.7}$$

From Theorem 2 with $\phi = \rho v_i$ for $i = 1, 2, 3$ applied to the left hand side, we have that

$$\frac{d}{dt}\int_V \rho v_i\, dx = \int_V (\partial_t(\rho v_i) + \nabla \cdot (\rho v_i v))\, dx. \tag{2.8}$$

Regarding the fact that (2.7) and (2.8) hold for all V, we have altogether

$$\partial_t(\rho v_i) + \nabla \cdot (\rho v_i v) = \rho f_i + (\nabla \cdot \sigma)_i.$$

Written in a vectorial notation this becomes the *conservative form* of the equation of conservation of momentum

$$\partial_t(\rho v) + \nabla \cdot (\rho v \otimes v) = \rho f + \nabla \cdot \sigma,$$

where the operator $v \otimes v := (v_i v_j)_{i,j=1}^3$ denotes the dyadic product. Using the continuity equation (2.4) the divergence term on the left hand side can be eliminated in order to get the *non-conservative form*

$$\rho \partial_t v + \rho(v \cdot \nabla)v = \rho f + \nabla \cdot \sigma \tag{2.9}$$

with an advection term $\rho(v \cdot \nabla)v$.

2.1.5 *Viscosity Model*

In order to further describe the stress tensor σ we have to make certain assumptions about the fluid, namely

Assumption 3 (Stokesian fluid)**.** *The stress on a fluid in a rest state is spherically symmetric depending on the pressure, meaning*

$$\sigma_{|v=0} = -pI$$

with pressure p and unit tensor $I := \delta_a^b$ with the Kronecker symbol δ_a^b.

Following this assumption we can write

$$\sigma = -pI + \tau$$

with a tensor τ describing shear stress. If we further assume *conservation of angular momentum* then τ is symmetric. For a Stokesian fluid the non-conservative form of the momentum equation (2.9) becomes

$$\rho \partial_t v + \rho(v \cdot \nabla)v = \rho f - \nabla p + \nabla \cdot \tau. \tag{2.10}$$

The shear stress tensor τ can now be related with the tensor of strain stress $\epsilon := \frac{1}{2}\left(\nabla v + \nabla v^T\right)$ through the constitutive relation

$$\tau = F(\epsilon),$$

where $F = (F_{ij})_{i,j=1,2,3}$ is a continuous and tensor-valued function. Assuming a linear constitutive relation F, we state the following approximation:

Assumption 4 (Newtonian fluid)**.** *The constitutive relation F is linear in ϵ and conclusively the tensor τ has necessarily the form*

$$\tau = 2\mu\epsilon + \lambda \mathrm{tr}(\epsilon) I \tag{2.11}$$

with the material constants μ for shear viscosity and λ for volume viscosity.

Following [113, p. 34] by assuming constant temperature in the fluid we can relate μ and λ by

$$3\lambda + 2\mu = 0, \quad \mu \geq 0,$$

and substituting this into (2.11) leads to

$$\tau = \mu\left(\nabla v + \nabla v^T\right) - \frac{2}{3}\mu(\nabla \cdot v) I.$$

Applying this to the conservation of momentum (2.9) results in

$$\rho \partial_t v + \rho(v \cdot \nabla)v - \mu\Delta v - \frac{1}{3}\mu\nabla(\nabla \cdot v) + \nabla p = \rho f. \tag{2.12}$$

Together with (2.4) these equations are known as the *compressible Navier-Stokes* equations. Assuming an incompressible fluid which reduces the continuity equation (2.4) to $\nabla \cdot v = 0$ leads to the impulse equation of the *incompressible Navier-Stokes* equations

$$\partial_t v + (v \cdot \nabla)v - \mu\Delta v + \nabla p = f. \tag{2.13}$$

2.1.6 *Conservation of Material Properties*

Let φ be a scalar function describing a physical property of a fluid like temperature or a concentration of material particles (tracers) inside a fluid. We assume that φ is conserved, meaning that changes of φ in a control volume can only occur by flux through its boundary or by sources and sinks. This conservation principle can be stated as

$$\frac{d}{dt}\int_V \rho\varphi \, dx = \int_{\partial V} f \cdot n \, dS + \int_V q \, dx \tag{2.14}$$

with a flux vector f and a source/sink term q. In the spirit of the continuum hypothesis, we can assume for the flux f the following relationship to φ:

Assumption 5 (Fick's first law)**.** *The diffusion flux f is proportional to the gradient of a quantity's concentration φ, formally*

$$f = D\,\nabla\varphi$$

with a diffusion coefficient $D = D(x,t)$.

Under this assumption and after applying Theorem 2 to the left hand side of (2.14), we have

$$\int_V \partial_t(\rho\varphi) + \nabla\cdot(\rho\varphi v)\,dx = \int_{\partial V}(D\,\nabla\varphi)\cdot n\,dS + \int_V q\,dx.$$

Transforming the surface integral by the divergence theorem and keeping in mind that the former equation holds for every V, we conclude

$$\partial_t(\rho\varphi) + \nabla\cdot(\rho\varphi v) = \nabla\cdot(D\,\nabla\varphi) + q$$

which is known by the name *advection-diffusion equation*, misleadingly *convection-diffusion equation* or simply *scalar transport equation*. By using the total derivative (2.1) and the continuity equation (2.4) this shortens to

$$\rho\,d_t\varphi = \nabla\cdot(D\,\nabla\varphi) + q. \tag{2.15}$$

2.2 OCEAN PRIMITIVE EQUATIONS

The basic equations of fluid dynamics presented in the previous section are of general nature. For the application of these equations in an ocean model several more aspects, e.g., external forces, the properties of the fluid and boundary conditions, need to be considered. This section establishes the necessary theoretical background in order to state the intrinsic equations of ocean modeling, the *Ocean Primitive Equations*.

To present the mathematical background for ocean modeling, we will adopt the notation common in this field of research and denote vectors by boldface symbols. This presentation of ocean modeling follows standard references [52, 57, 70, 132].

2.2.1 *Geographical Coordinate System*

The most dominant external force for fluid dynamics in the earth's oceans is the geopotential force $\mathbf{g}$ which is the concluding force from gravitational and centrifugal force (introduced in Subsection 2.2.4). Therefore, a suitable orthogonal coordinate system for the earth is naturally found by first requiring $\mathbf{g}/\|\mathbf{g}\|$ to be the vertical basis di-

rection whereby the horizontal directions are separated from $\mathbf{g}$ by orthogonality. Together with the fact that the geopotential surface of the earth can be well approximated by an oblate spheroid with equatorial radius larger then polar radius, oblate spherical coordinates seem the best choice to fulfill this requirement. The downside to this approach is that the metric functions to measure distances are more complicated to handle for oblate spherical coordinates than for spherical coordinates, where $\mathbf{g}$ would exhibit an unintentional tangential component. However, it is possible, to a high order of accuracy, to maintain both advantages by a combination of spherical and oblate spherical coordinates.

We start with defining geographical coordinates (r, ϕ, λ), which are a variant of spherical coordinates, as a transformation to Cartesian coordinates (x^1, x^2, x^3) with

$$\begin{aligned} x^1 &= r\cos\phi\cos\lambda, \\ x^2 &= r\cos\phi\sin\lambda, \\ x^3 &= r\sin\phi, \end{aligned}$$

where we call r the *radial coordinate*, $\phi \in [-\frac{1}{2}\pi, \frac{1}{2}\pi]$ the *latitude* and $\lambda \in [0, 2\pi]$ the *longitude*. It should be noted that geographical coordinates transform to classical spherical coordinates (r_s, ϕ_s, λ_s) by letting $\phi_s = \frac{1}{2}\pi - \phi$. Although strictly not correct, the lateral directions latitude and longitude are sometimes referred to as horizontal directions. This terminology conceals the fact that the geometry on a sphere is curved and therefore non-Euclidean. Basic differential calculus shows how the metric on a sphere relates to the Euclidean metric, i.e., the squared infinitesimal distance between two points on a sphere is given as

$$(ds)^2 = (r\cos\phi\, d\lambda)^2 + (r\, d\phi)^2 + (dr)^2.$$

Regarding the earth as a sphere would oversimplify the fact that the earth's geopotential surface is approximately an oblate spheroid, i.e., an ellipse with semi-major axis $a = 6378.139\,\text{km}$ and semi-minor axis $b = 6356.754\,\text{km}$ rotated around one of these axes. A good approximation of this spheroid by a sphere allows us to use geographical coordinates to approximate the metric functions of the oblate spherical coordinates at the earth's surface to a high degree of accuracy and still regard r as geopotential surface [52].

The radius r of such an approximating sphere can be found by different means. One is to find

$$\min_{R>0} \int_0^{\frac{\pi}{2}} \left(R_e^2(t) - R^2\right)^2 dt$$

with the norm of a parametrized ellipse $R_e(t) = \|(a\cos(t), b\sin(t))^t\|$ which results in $R = 6367.456\,\text{km}$. Another common approach is to

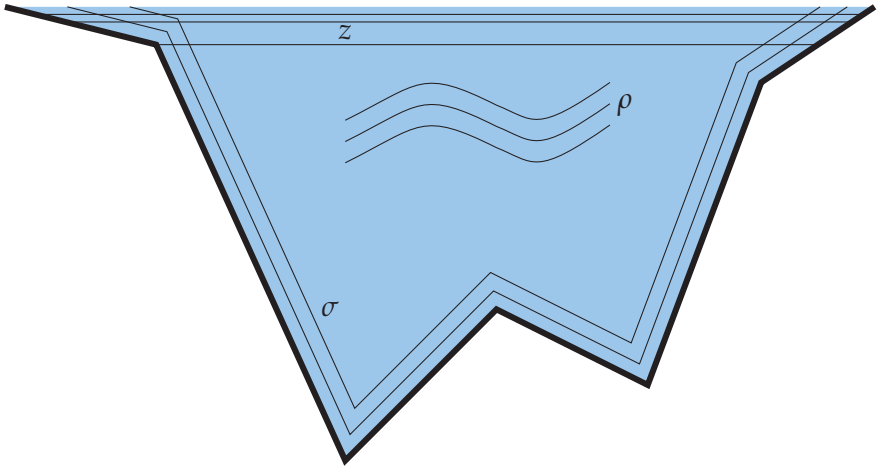


Figure 3: Vertical section of an ocean basin illustrating the three fundamental regimes of ocean dynamics and their corresponding vertical coordinates. The surface mixed layer is naturally represented by z-coordinates and the bottom topography with σ-coordinates. The ocean's interior where most transport processes occur along layers of constant potential density is naturally represented by ρ-coordinates [57].

use the radius of a sphere with the same volume as the Earth resulting in $R = 6371\,\text{km}$.

Since we let R represent the geopotential surface of the earth's sea level, the vertical coordinate r can be expressed as deviation from R, formally

$$r = R + z.$$

This allows us to use z as vertical coordinate instead of r. Ocean models which use this form of vertical coordinates are termed *z-coordinates* or *geopotential coordinates* ocean models. Besides the widely used z-coordinates, there are mainly two other classes of vertical coordinates in ocean models, namely *isopycnal* or *ρ-coordinates* and *σ-coordinates*. A surface of constant vertical coordinate in *ρ-coordinates* describes a surface of constant density whereas in the case of *σ-coordinates* a surface of constant distance to the ocean's bottom topography is described. These three different vertical coordinates are illustrated in Figure 3. Although the choice of vertical coordinates is one of the most fundamental aspects in the design of an ocean model, oceanologists have not yet come to a final conclusion which vertical coordinate is the most appropriate for ocean modeling [58].

2.2.2 *General Lateral Orthogonal Coordinates*

The geographical coordinates as defined in Subsection 2.2.1 are often impractical for ocean modeling, since the poles ($\phi = -\frac{\pi}{2}, \frac{\pi}{2}$) inherently lead to singularities if they are part of the considered domain. A common way to resolve this problem is the introduction of general

orthogonal lateral coordinates (ξ^1, ξ^2) instead of (ϕ, λ). In this case, the squared infinitesimal length of a line element takes the general form

$$(ds)^2 = (h_1 d\xi^1)^2 + (h_2 d\xi^2)^2 + (dr)^2$$

with the metric functions $h_1(\xi^1, \xi^2, r)$ and $h_2(\xi^1, \xi^2, r)$. It should be noted that the lack of mixed terms $d\xi^1 d\xi^2$ is due to the stipulated orthogonality. If r varies only a little relative to a constant R, it can be substituted by R in h_1 and h_2 without losing much accuracy which is a reasonable assumption when comparing the maximal depths in the oceans to the radius of the earth.

Assumption 6 (Shallow ocean approximation)**.** *Compared to the radius of the earth the extent of the ocean layer is small and therefore the parameter variations in r can be neglected. Therefore, r can be substituted by the constant earth radius R in the metric functions h_1 and h_2.*

Regardless of this assumption, the application of the basic equations of fluid dynamics from Section 2.1 to a spherical geometry should be done with caution because they are in general not independent of the choice of coordinates. For instance, this can be seen when we consider the total derivative (2.1) on a spherical geometry. Thereby, we have

$$\rho \frac{d\mathbf{v}}{dt} = \partial_t(\rho \mathbf{v}) + \nabla \cdot (\rho \mathbf{v} \otimes \mathbf{v}) + \mathcal{M}(\mathbf{r} \times \rho \mathbf{v}),$$

where

$$\mathcal{M} = v\partial_x ln\, dy - u\partial_y ln\, dx$$

is referred to as *advective metric frequency* or in [70] as *metric terms* or *curvature effects* [57, p. 53].

2.2.3 *Coriolis Acceleration*

Newton's first law of motion states that in the absence of any force the uniform motion of a material volume relative to a fixed coordinate system does not change. This is referred to as *inertial motion*, because no acceleration or deacceleration occurs. Even if the coordinate system itself was changing in space throughout time with an *inertial* motion the material volume's motion would still be regarded as *inertial.* For instance, if the coordinate system is moving along with a material volume with the same uniform motion and direction, the material volume would be at rest relative to this local coordinate system. A coordinate system with this property is called *inertial frame of reference*. If the coordinate system is changing in a *non-inertial* way, e.g., a coordinate system fixed on the surface of a rotating sphere, it is called *non-inertial reference frame*. A material volume at rest or in uniform motion in such a non-inertial reference frame experiences non-inertial motion with respect to a fixed coordinate system. To compensate

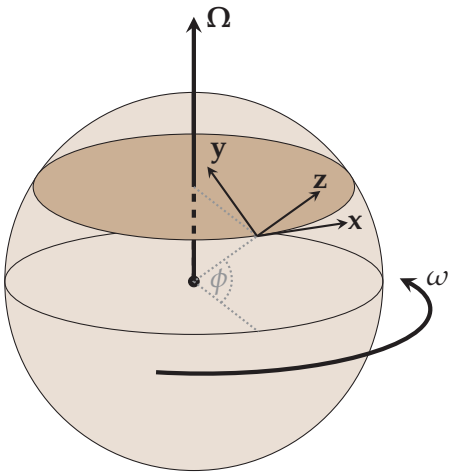


Figure 4: A local Cartesian coordinate system $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ on the surface of a sphere rotating around axis $\mathbf{\Omega}$ with angular velocity ω. The angle ϕ is defining the direction of $\mathbf{\Omega}$ in the local coordinate system.

for that a *fictitious force* (or *pseudo force*) needs to be introduced that describes the acceleration of a material volume with an inertial motion in a non-inertial reference frame with respect to an inertial reference frame. On a sphere rotating around an axis $\mathbf{\Omega} = \omega(0, \cos(\phi), \sin(\phi))^T$ with angular velocity ω this compensating fictitious force, acting as an accelerating volume force, is called *Coriolis acceleration* and is illustrated in Figure 4.

We follow the derivation of the Coriolis acceleration as in [52, p. 73]. With the subscript f we describe a quantity relative to a fixed frame and with r relative to a rotating frame. A fixed point $\mathbf{x}_r$ in a rotating frame has velocity $\mathbf{\Omega} \times \mathbf{x}_r$ in the fixed frame, i.e.,

$$\frac{d\mathbf{x}_f}{dt} = \mathbf{\Omega} \times \mathbf{x}_r.$$

If $\mathbf{x}_r$ is moving in the rotation frame with velocity $d\mathbf{x}_r/dt$ this adds up to

$$\frac{d\mathbf{x}_f}{dt} = \frac{d\mathbf{x}_r}{dt} + \mathbf{\Omega} \times \mathbf{x}_r.$$

Repeating the last arguments for the acceleration of $\mathbf{x}_f$, we have

$$\begin{aligned} \frac{d^2\mathbf{x}_f}{dt^2} &= \frac{d}{dt}\left(\frac{d\mathbf{x}_r}{dt} + \mathbf{\Omega} \times \mathbf{x}_r\right) + \mathbf{\Omega} \times \left(\frac{d\mathbf{x}_r}{dt} + \mathbf{\Omega} \times \mathbf{x}_r\right) \\ &= \frac{d^2\mathbf{x}_r}{dt^2} + 2\mathbf{\Omega} \times \frac{d\mathbf{x}_r}{dt} + \mathbf{\Omega} \times (\mathbf{\Omega} \times \mathbf{x}_r). \end{aligned}$$

By rewriting the last term in geographical coordinates and velocity $\mathbf{v} = (u, v, w)$, we have

$$\frac{d\mathbf{v}_f}{dt} = \frac{d\mathbf{v}_r}{dt} + 2\mathbf{\Omega} \times \mathbf{v}_r - \frac{1}{2}\nabla_r(\omega r \cos\phi)^2,$$

where ∇_r is the gradient with respect to the geographical coordinates. The term $\frac{1}{2}\nabla(\omega r \cos\phi)^2$ is called *centrifugal acceleration* and $-2\boldsymbol{\Omega} \times \mathbf{u}$ is termed *Coriolis acceleration* with components

$$\mathbf{a}_c = -2\boldsymbol{\Omega} \times \mathbf{v} = 2\omega \begin{pmatrix} v\sin\phi - w\cos\phi \\ -u\sin\phi \\ u\cos\phi \end{pmatrix}.$$

The shallow ocean assumption influences the Coriolis acceleration in the way that two points at the same lateral coordinate with different vertical coordinates exhibit the same angular momentum in the ocean. This means that motion in the vertical direction does not affect angular momentum and therefore the non-radial component of $\boldsymbol{\Omega}$ needs to be eliminated. Under the shallow ocean assumption, this results in the Coriolis acceleration

$$\tilde{\mathbf{a}}_c = -f\,\mathbf{z} \times \mathbf{v} \tag{2.16}$$

with *Coriolis parameter* $f = 2\omega\sin\phi$.

2.2.4 *Effective Gravitational Force*

On a sphere with radius R and mass M_s the potential energy Φ of a material volume with mass m at radius $r = z + R$ due to gravitation is

$$\begin{aligned} m\Phi &= -\frac{GM_s m}{z+R} \\ &= -\frac{GM_s m}{R} + \frac{GM_s}{R^2}mz + \left(\frac{z}{R}\right)^2 \frac{GM_s}{(z+R)}, \end{aligned}$$

where $G = 6.67 \times 10^{-11}\,\mathrm{Nm^2kg^{-2}}$ is Newton's gravitational constant and mass $M_s = 5.98 \times 10^{24}\,\mathrm{kg}$ for the Earth. Dropping the last term because of the shallow ocean assumption ($z \ll R$) and defining $g_s = GM_s/R^2$ leaves

$$m\Phi = C_\Phi + g_s m z, \tag{2.17}$$

where $C_\Phi = -GM_s m/R$ is a geopotential constant. A vector describing the gravitational acceleration is therefore given as

$$\nabla\Phi = \nabla(zg_s) = \nabla(rg_s).$$

Combined with the centrifugal acceleration, the effective gravitational acceleration is

$$\begin{aligned} \mathbf{g} &= -\nabla(rg_s - \frac{1}{2}(\omega r\cos\phi)^2) \\ &= -(g_s - r\omega^2\cos^2\phi)\mathbf{r} - (r^2\omega^2\cos\phi\sin\phi)\boldsymbol{\phi}. \end{aligned} \tag{2.18}$$

Because of the parts in (2.18) caused by rotation, this effective gravitational force is not orthogonal to the surface of a sphere. Assuming

again an oblate spheroid that compensates for this non-orthogonality, we can drop the non-radial last term of (2.18) to obtain an easier representation of the effective gravitational force. Furthermore, we tare the geopotential to zero for $z = 0$ so that the surface of an ocean at rest would have no geopotential energy, i.e., we drop C_Φ from (2.17) that is irrelevant for dynamics. Although the effective gravitational acceleration depends on ϕ, what is obviously to be seen in the first term of (2.18), we can assume the latitude $\phi = 45°$ without much loss of accuracy. Taking also into account the angular velocity of the earth $\omega = 7.2921 \times 10^{-5}\,\mathrm{s}^{-1}$ and the shallow ocean assumption, i.e., $r = R = 6371\,\mathrm{km}$, we have that $g = g_s - r\omega^2 \cos^2\phi$ is a constant and consequently

$$\mathbf{g} \;=\; -g\mathbf{z}$$

with $g \approx 9.81\,\mathrm{ms}^{-2}$. Ocean models based on Cox [35] apply similar simplifications following Moritz [96] to obtain $g = 9.806\,\mathrm{ms}^{-2}$.

2.2.5 *Hydrostatic Balance*

A fluid at rest maintains a balance between two forces acting on a material volume: pressure and geopotential force. This observation yields the *hydrostatic equation*

$$\partial_z p = -\rho g. \tag{2.19}$$

A vertical scale analysis for the momentum equation (2.9) as in [70, p. 41] shows that the horizontal scale is very large compared to the vertical scale of motions in the ocean. Therefore, we can state the following approximation:

Assumption 7 (Hydrostatic approximation)**.** *The ocean maintains a state of static equilibrium in conformity with the hydrostatic equation* (2.19)*.*

Using this assumption renders the pressure p a *diagnostic variable* that can be determined by integrating the hydrostatic equation which therefore constitutes a *diagnostic equation*. We get

$$p(z) = p_a + g \int_z^\eta \rho(r)\, dr$$

with the pressure of the atmosphere p_a at the ocean surface η.

2.2.6 *Kinematic Boundary Condition*

To derive the boundary conditions for a z-coordinate model, we let $z = 0$ represent the geopotential surface, $z = \eta$ the real surface (top boundary) and $z = -H$ the solid ground surface (bottom boundary). The top and bottom boundary surfaces are free surfaces parametrized

with x, y, i.e., $\eta(x, y)$ and $H(x, y)$. Any surface element can be written as

$$dA_{(\eta)} = |\nabla(-\eta + z)|\, dx\, dy$$

which is directly the definition of the area of a free surface. The normal on the bottom which points inward into the ocean is defined as

$$\hat{\mathbf{n}}(-H) = \frac{\nabla(H + z)}{|\nabla(H + z)|} \tag{2.20}$$

and the normal at the surface pointing outward of the ocean is defined as

$$\hat{\mathbf{n}}(\eta) = \frac{\nabla(-\eta + z)}{|\nabla(-\eta + z)|}.$$

We assume that the bottom boundary of an ocean is static so that $\partial_t \hat{\mathbf{n}}(-H) = 0$. Additionally we request that the bottom surface is impermeable, formally

$$\hat{\mathbf{n}}(-H) \cdot \mathbf{v} = 0, \quad z = -H.$$

Using the definition (2.20) we get that

$$\mathbf{u} \cdot \nabla H + w = 0, \quad z = -H, \tag{2.21}$$

where the three-dimensional velocity field is split into a horizontal and vertical part $\mathbf{v} = (\mathbf{u}, w)$ with $\mathbf{u} = (u, v)$. This equation is known as the *kinematic boundary condition* for the bottom boundary.

In order to derive the surface kinematic boundary condition, we consider an arbitrary cuboid with the upper side being the ocean surface η and the opposite side being the bottom H as illustrated in Figure 5. The mass M of the cuboid C is

$$\begin{aligned} M &= \int_C \rho\, dV \\ &= \int_{A_c} \int_{-H}^{\eta} \rho\, dz\, dA, \end{aligned}$$

where A_c is the orthogonal cross section of C. The change of mass in C is therefore

$$\partial_t M = \int_{A_c} \partial_t \left(\int_{-H}^{\eta} \rho\, dz \right) dA. \tag{2.22}$$

Using the divergence theorem we can express the incoming fluxes over the vertical sides ∂C_v of C as

$$\begin{aligned} -\int_{\partial C_v} \rho \mathbf{u} \cdot \mathbf{n}\, dA &= -\int_{\partial A_c} \left(\int_{-H}^{\eta} \rho \mathbf{u}\, dz \right) \cdot \mathbf{n}\, ds \\ &= -\int_{A_c} \nabla \cdot \int_{-H}^{\eta} \rho \mathbf{u}\, dz\, dA \end{aligned} \tag{2.23}$$

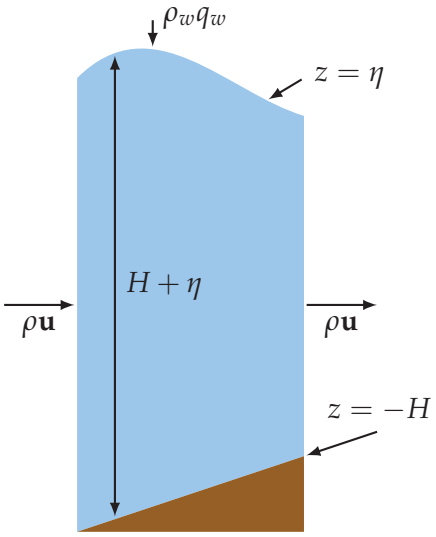


Figure 5: The balance of mass in a cuboid spanning from the solid earth boundary to the surface of the ocean. The change in mass in the cuboid is determined by the flux of seawater $\rho\mathbf{u}$ over its side boundaries, the time-dependent fluctuations of the free surface $z = \eta$ and the input of fresh water $\rho_w q_w$ over its top boundary [57].

with outer normal $\mathbf{n}$. At any point on the surface area of C, fresh water fluxes can occur and we represent this as a vector $\mathbf{n}_w$ with its length determining the amount of fresh water. Thereby, the overall amount of fresh water transiting over the surface boundary is just the projection of $\mathbf{n}_w$ onto $\hat{\mathbf{n}}(-\eta)$ integrated over the surface area. Formally, we have for the mass of fresh water Q_w that

$$Q_w = \int_\eta \rho_w \mathbf{n}_w \cdot \hat{\mathbf{n}}(-\eta)\, dA = \int_\eta \rho_w q_w \, dA \tag{2.24}$$

with q_w being the actual volume element of fresh water passing over the ocean surface boundary per unit time per unit horizontal cross-sectional area and ρ_w its in situ density. From (2.24) and (2.23), we have

$$\partial_t M = \int_{A_c} \left(\rho_w q_w - \nabla \cdot \int_{-H}^{\eta} \rho\mathbf{u}\, dz \right) dA.$$

Together with (2.22), it follows that

$$\int_{A_c} \partial_t \left(\int_{-H}^{\eta} \rho\, dz \right) dA = \int_{A_c} \left(\rho_w q_w - \nabla \cdot \int_{-H}^{\eta} \rho\mathbf{u}\, dz \right) dA$$

and due to the fact that C is arbitrary, we have

$$\partial_t \int_{-H}^{\eta} \rho\, dz + \nabla \cdot \int_{-H}^{\eta} \rho\mathbf{u}\, dz = \rho_w q_w.$$

Evaluating the divergence and time derivative in the last equation, we have

$$\rho(\eta)(\partial_t + \mathbf{u}(\eta)\cdot\nabla)\eta + \mathbf{u}(-H)\cdot\nabla H + \int_{-H}^{\eta}(\partial_t\rho + \nabla\cdot(\rho\mathbf{u}))\,dz = \rho_w q_w.$$

Applying the continuity equation (2.4) and bottom boundary condition (2.21), we have

$$(\partial_t + \mathbf{u}\cdot\nabla)\,\eta = \left(\frac{\rho_w}{\rho}\right) q_w + w, \quad z = \eta. \tag{2.25}$$

This is known as the *surface boundary condition*. Setting the fresh water flux q_w to zero and evaluating the $u\cdot\nabla$ operator, we get

$$\partial_t\eta + u\partial_x\eta + v\partial_y\eta = w, \quad z = \eta.$$

For many applications the nonlinear terms in the last equation are of second order and neglecting them provides a good approximation. This is known as the linearized kinematic boundary condition

$$\partial_t\eta = w|_{z=\eta}. \tag{2.26}$$

In a practical ocean model the z-coordinate is discretized into a number of, in general not equidistant, layers according to the chosen resolution with the topmost layer at the geopotential surface $z = 0$. Since the distance between the actual ocean surface $z = \eta$ and the zero geopotential surface $z = 0$ is generally quite small compared to the distance of the two topmost layers, the ocean surface $z = \eta$ is often not resolved. Following [105, p. 56], this is stated as:

Assumption 8 (Non-rigid lid approximation)**.** *The boundary conditions at the ocean's surface are stated at $z = 0$ instead of $z = \eta$ to avoid the necessity of an additional layer with non-fixed size.*

Taking this assumption, we have that the vertical velocity at the ocean surface equals the velocity at the zero potential surface so that

$$\partial_t\eta = w|_{z=0}. \tag{2.27}$$

2.2.7 *Boussinesq Fluid*

According to [139, p. 118], the mean density of sea water is near 1.025 kg/m^3 and varies less than 2% from 1.035 kg/m^3 [52, p. 47]. This observation justifies the often applied approximation in ocean models to use a constant reference density ρ_w for water if ρ_w is not multiplied by g. This approximation is attributed to Boussinesq [23] and states:

Assumption 9 (Boussinesq approximation)**.** *Under the condition that variations of density in a fluid are relatively small, treating density as a*

constant, whenever not multiplied by a factor of larger order of magnitude (like the geopotential g), gives a suitable approximation.

Under this assumption the continuity equation (2.4) simplifies to

$$\nabla \cdot v = 0, \tag{2.28}$$

and we say that v is *divergence-free* or *solenoidal*. Furthermore, there is no longer any *prognostic equation* to allow the treatment of ρ as a *prognostic variable*. Since the variations of ρ from ρ_w occur with respect to temperature, salinity and pressure it is possible to define a diagnostic equation to determine ρ by

$$\rho = \rho(S, T, p), \tag{2.29}$$

where S is the salinity and T the *in situ temperature* [48]. A function which determines the density by these parameters is generally known as *equation of state*.

2.2.8 *Eddy Viscosity*

In large scale flows it is often impractical to resolve all motions of a fluid down to the smallest scale. Using a resolution that does not resolve small-scale motions like turbulent eddies gives raise to the question how to treat those effects. A common approach is to model the momentum by eddies the same way as the momentum by the motion of molecules, i.e., the viscosity presented in Subsection 2.1.5. This is stated in the following assumption:

Assumption 10 (Boussinesq eddy viscosity)**.** *The turbulent transfer of momentum by eddies can be analogously modeled as the molecular viscosity caused by the transfer of momentum by the motion of molecules as*

$$\tau = \mu \left(\nabla v + \nabla v^T \right) - \frac{2}{3} \mu (\nabla \cdot v) I$$

in case of a Newtonian fluid. It is therefore referred to as eddy viscosity*.*

Consequently the corresponding term in the incompressible conservation of momentum equation (2.13) is $\mu \Delta v$. Following [122], we decompose the eddy viscosity for the lateral velocity $\mathbf{u}$ in lateral $\mathbf{F}_H$ and vertical $\mathbf{F}_V$ components

$$\mathbf{F}_H = A_H \Delta_H \mathbf{u},$$
$$\mathbf{F}_V = A_V \frac{\partial^2}{\partial z^2} \mathbf{u},$$

where $\Delta_H := \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ and A_H, A_V are the lateral and vertical viscosity parameters which are allowed to be different.

Going back to the work of Holland [69] as well as Semtner and Mintz [122], it is now taken as a well established fact that eddy viscosity given by a biharmonic operator that acts more strongly on small scales and less on large scales than the Laplacian operator is better suited in eddy-resolving ocean models. Taking into account that only resolution of lateral eddies are of interest, we define

$$\tilde{\mathbf{F}}_H = -B_H \Delta^2 \mathbf{u}$$

with viscosity parameter B_H. It should be noted, that modeling viscosity with a biharmonic operator is not directly motivated by physical laws and can therefore be considered purely heuristical. As pointed out by Delhez and Deleersnijder [37], the usage of a biharmonic operator in lateral diffusion of momentum and tracers can lead to over-shootings and spurious oscillations and should be applied with caution.

2.2.9 *Governing Equations*

At this point, we have everything at hand to define the governing equations of an ocean model. Rewriting the conservation of momentum equation (2.10) under the Boussinesq approximation (Assumption 9) with constant density ρ_w, we have

$$\partial_t \mathbf{v} + (\mathbf{v} \cdot \nabla)\mathbf{v} = -\frac{1}{\rho_w}\nabla p + \mathbf{f} + \frac{1}{\rho_w}\nabla \cdot \tau. \tag{2.30}$$

With the hydrostatic approximation (Assumption 7) the pressure p can be decomposed into two additive parts, that is the pressure p_0 with the ocean surface at the geopotential $z = 0$ and the pressure $p_\eta = \rho_w g \eta$ caused by the deviation of the free boundary η from $z = 0$. Furthermore, we let $\mathbf{v} = (\mathbf{u}, w)^T = (u, v, w)^T$ and restrict (2.30) to the lateral directions $\mathbf{u}$ so that the geopotential force is perpendicular and the arbitrarily external force $\mathbf{f}$ only consists of the Coriolis force (2.16) under the shallow ocean approximation (Assumption 6). This results in

$$\partial_t \mathbf{u} + (\mathbf{u} \cdot \nabla_H)\mathbf{u} + w\partial_z \mathbf{u} + f(\mathbf{z} \times \mathbf{u}) = -\frac{1}{\rho_w}\nabla_H(p_0 + \rho_w g\eta) + \mathbf{F}_H + \mathbf{F}_V \tag{2.31}$$

with Coriolis parameter f, the normal to the geopotential surface $\mathbf{z}$, the horizontal gradient operator ∇_H and the horizontal and vertical viscosity parametrization $\mathbf{F}_H$ and $\mathbf{F}_V$. Using the total derivative, this shortens to

$$\frac{d}{dt}\mathbf{u} + f(\mathbf{z} \times \mathbf{u}) = -\frac{1}{\rho_w}\nabla_H(p_0 + \rho_w g\eta) + \mathbf{F}_H + \mathbf{F}_V \tag{2.32}$$

which describes the horizontal momentum balance for a hydrostatic Boussinesq fluid on a sphere [91, p. 93] with prognostic treatment of $\mathbf{u}$ and η. Density ρ will be treated diagnostically by means of the equation of state $\rho = \rho(S, T, p)$ with salinity S, pressure p and in situ temperature T according to the equation of state polynomial formula by the Joint Panel on Oceanographic Tables and Standards [48]. The diagnostic calculation of pressure p_0 is done by the hydrostatic equation (2.19) under the hydrostatic approximation (Assumption 7). The vertical velocity w is determined by the horizontal velocity field utilizing the incompressible continuity equation (2.28) to obtain

$$\partial_z w = -\nabla_H \cdot \mathbf{u}. \tag{2.33}$$

The vertical velocity at the ocean surface is consequently

$$w|_{z=\eta} = -\nabla_H \cdot \int_{-H}^{\eta} \mathbf{u}\, dz,$$

and finally the ocean's surface linearized kinematic boundary condition (2.27) yields $\partial_t \eta$.

The *potential temperature* Θ and salinity S are treated as tracers determined by the transport equation (2.15). Taking these quantities per volume together with a source/sink term $q = 0$ and the Boussinesq approximation to justify dropping density, we have

$$\begin{aligned} d_t \Theta &= \nabla \cdot (D\, \nabla \Theta), \\ d_t S &= \nabla \cdot (D\, \nabla S), \end{aligned}$$

where the tensor D is a subgrid-scale parametrization of lateral and vertical diffusion. The surface boundary condition is stated by a Robin boundary condition

$$\begin{aligned} D_v \partial_z \Theta &= \lambda(\Theta_* - \Theta), \quad z = \eta, \\ D_v \partial_z S &= \lambda(S_* - S), \quad z = \eta, \end{aligned}$$

where D_v is the vertical diffusion coefficient, λ a relaxation parameter and Θ_*, S_* given surface fields. At the bottom and lateral boundaries a no-flux condition is applied.

At first sight, these governing equations seem to exhibit the same complexity as the three-dimensional Navier-Stokes equations which yet lack a proof for the global existence of the strong solution*. On the contrary, because of the diagnostic treatment of vertical velocity z, the ocean primitive equations are of a much simpler nature. For a simplified variant of the ocean primitive equations, i.e., with disregard of the fluid's density and on a cylindrical domain $M \times (-h, 0)$ with M being a smooth bounded domain in $\mathbb{R}^2$, the existence and uniqueness of a strong solution can be proved [29].

*http://www.claymath.org/millennium/Navier-Stokes_Equations/

2.3 BAROCLINIC AND BAROTROPIC VELOCITIES

In order to solve the ocean model's momentum equation (2.32), we decompose the lateral velocity fields $\mathbf{u}$ into its barotropic $\bar{\mathbf{u}}$ and baroclinic $\tilde{\mathbf{u}}$ components

$$\begin{aligned} \bar{\mathbf{u}} &= \int_{-H}^{0} \mathbf{u}\, dz, \\ \tilde{\mathbf{u}} &= \mathbf{u} - \frac{\bar{\mathbf{u}}}{H}, \end{aligned}$$

where H is the local depth of the sea. Following [124], we can now reformulate the momentum equation 2.31 into a *barotropic subsystem*

$$\partial_t \bar{\mathbf{u}} + \bar{A}(\mathbf{u},\mathbf{u}) + f(\mathbf{z}\times\bar{\mathbf{u}}) = -gH\nabla_H \eta - \int_{-H}^{0} \nabla_H \bar{p}\, dz + \bar{\mathbf{F}}_H + \bar{\mathbf{F}}_V \tag{2.34}$$

and a *baroclinic subsystem*

$$\partial_t \tilde{\mathbf{u}} + A(\mathbf{u},\mathbf{u}) - \bar{A}(\mathbf{u},\mathbf{u}) + f(\mathbf{z}\times\tilde{\mathbf{u}}) = \frac{1}{H}\int_{-H}^{0} \nabla_H \hat{p}\, dz - \nabla_H \hat{p} + \tilde{\mathbf{F}}_H + \tilde{\mathbf{F}}_V, \tag{2.35}$$

where $\hat{p} = p/\rho_w$ denotes the internal pressure divided by the constant density and the advection term given by

$$A(\mathbf{u},\mathbf{u}) = (\mathbf{u}\cdot\nabla)\mathbf{u} + w\partial_z \mathbf{u}$$

and analogously defined barotropic $\bar{A}(\mathbf{u},\mathbf{u})$, $\bar{\mathbf{F}}_H$, $\bar{\mathbf{F}}_V$ as well as baroclinic $\tilde{\mathbf{F}}_H$ and $\tilde{\mathbf{F}}_V$. Partially updating $\tilde{\mathbf{u}}$ and $\bar{\mathbf{u}}$ with respect to the advection terms $A(\mathbf{u},\mathbf{u})$, $\bar{A}(\mathbf{u},\mathbf{u})$ and viscosity terms $\bar{\mathbf{F}}_H$, $\bar{\mathbf{F}}_V$, $\tilde{\mathbf{F}}_H$, $\tilde{\mathbf{F}}_V$ by means of *operator splitting techniques* [107] allows further reduction of the barotropic and baroclinic subsystems.

2.3.1 *Baroclinic Subsystem*

The equations of the baroclinic subsystem with partially updated baroclinic velocities are given as

$$\partial_t \tilde{u} - f\tilde{v} = \frac{1}{H}\int_{-H}^{0} \partial_x \hat{p}\, dz - \partial_x \hat{p}, \tag{2.36}$$

$$\partial_t \tilde{v} + f\tilde{u} = \frac{1}{H}\int_{-H}^{0} \partial_y \hat{p}\, dz - \partial_y \hat{p}. \tag{2.37}$$

Taking the *small-amplitude* approximation [52, p. 129], we can assume a linearized form of the density conservation $d_t\rho = 0$, namely

$$\partial_t \rho = -w\partial_z \rho. \tag{2.38}$$

Substituting the right hand side of (2.38) into the time derivative of the hydrostatic equation (2.19), i.e.,

$$\partial_t \partial_z \hat{p} = -\frac{g}{\rho_w} \partial_t \rho,$$

concludes an equation for the time evolution of the internal pressure, i.e.,

$$\partial_t \partial_z \hat{p} = \frac{wg}{\rho_w} \partial_z \rho. \tag{2.39}$$

Applying the continuity equation (2.33) to the baroclinic velocities results in the baroclinic continuity equation

$$\partial_z \tilde{w} = -\nabla_H \tilde{\mathbf{u}} \tag{2.40}$$

which can be used to obtain $\tilde{w}$.

The equations of the baroclinic subsystem (2.44), (2.45) and (2.39) can now be discretized by means of a relaxed Crank-Nicolson scheme (or more generally as a one-step Θ-Scheme [74]) in the spirit of Wolff et al. [142]. Denoting the time step by superscripts and the spatial partial derivatives by subscripts, we have

$$\begin{aligned} \tilde{u}^{n+1} - \tilde{u}^n = {} & \alpha \Delta t \left(f \tilde{v}^{n+1} - \hat{p}_x^{n+1} + \frac{1}{H} \int_{-H}^{0} \hat{p}_x^{n+1} \, dz \right) \\ & + (1-\alpha) \Delta t \left(f \tilde{v}^n - \hat{p}_x^n + \frac{1}{H} \int_{-H}^{0} \hat{p}_x^n \, dz \right), \end{aligned} \tag{2.41}$$

$$\begin{aligned} \tilde{v}^{n+1} - \tilde{v}^n = {} & \alpha \Delta t \left(-f \tilde{u}^{n+1} - \hat{p}_y^{n+1} + \frac{1}{H} \int_{-H}^{0} \hat{p}_y^{n+1} \, dz \right) \\ & + (1-\alpha) \Delta t \left(-f \tilde{u}^n - \hat{p}_y^n + \frac{1}{H} \int_{-H}^{0} \hat{p}_y^n \, dz \right), \end{aligned} \tag{2.42}$$

$$\hat{p}_z^{n+1} - \hat{p}_z^n = \frac{g}{\rho_w} \Delta t \rho_z \left(\beta \tilde{w}^{n+1} + (1-\beta) \tilde{w}^n \right), \tag{2.43}$$

with relaxation parameters α and β. To solve these equations a fixed point iteration with parameter l for $\hat{p}^{n+1,l}$ and start value $\hat{p}^{n+1,1} = \hat{p}^n$ is applied. The equations (2.41) and (2.42) are solved with fixed $\hat{p}^{n+1,l}$ for $\tilde{u}^{n+1,l+1}$ and $\tilde{v}^{n+1,l+1}$. Then (2.40) is used to get $\tilde{w}^{n+1,l+1}$ and consequently with (2.43) an updated $\hat{p}^{n+1,l+1}$ is obtained. This progress can be repeated until a convergence criteria is met.

2.3.2 *Barotropic Subsystem*

The momentum equations of the barotropic subsystem with partially updated barotropic velocities are

$$\partial_t \bar{u} - f\bar{v} + gH\partial_x \eta + \int_{-H}^{0} \partial_x \hat{p}\, dz = 0, \tag{2.44}$$

$$\partial_t \bar{v} + f\bar{u} + gH\partial_y \eta + \int_{-H}^{0} \partial_y \hat{p}\, dz = 0. \tag{2.45}$$

Integrating the continuity equation (2.33) in z-direction and applying the surface boundary condition (2.27) concludes the barotropic continuity equation

$$\partial_t \eta + \partial_x \bar{u} + \partial_y \bar{v} = 0.$$

Analogously as in Section 2.3.1, these equations can be discretized in time to get

$$\begin{aligned} &\bar{u}^{n+1} - \bar{u}^n - f\Delta t \left(\alpha \bar{v}^{n+1} + (1-\alpha)\bar{v}^n\right) \\ &+gH\Delta t \left(\alpha \eta_x^{n+1} + (1-\alpha)\eta_x^n\right) + \Delta t \int_{-H}^{0} \hat{p}_x^{n+1}\, dz = 0, \end{aligned} \tag{2.46}$$

$$\begin{aligned} &\bar{v}^{n+1} - \bar{v}^n - f\Delta t \left(\alpha \bar{u}^{n+1} + (1-\alpha)\bar{u}^n\right) \\ &+gH\Delta t \left(\alpha \eta_y^{n+1} + (1-\alpha)\eta_y^n\right) + \Delta t \int_{-H}^{0} \hat{p}_y^{n+1}\, dz = 0, \end{aligned} \tag{2.47}$$

$$\eta^{n+1} - \eta^n + \Delta t \left(\beta \bar{u}_x^{n+1} + (1-\beta)\bar{u}_x^n + \beta \bar{v}_y^{n+1} + (1-\beta)\bar{v}_y^n\right) = 0 \tag{2.48}$$

with relaxation parameters α and β. Equations (2.46) and (2.47) are now solved for $\bar{u}^{n+1}$ and $\bar{v}^{n+1}$ and substituted into (2.48) to attain an equation for η^{n+1} and its spatial derivatives η_x^{n+1} and η_y^{n+1}.

2.4 DISCRETIZATION

Depending on the focus of interest in the climate projections, different horizontal coordinates on a sphere are applied. In general, those coordinates are orthogonal and curvilinear to avoid complicated metrical functions as explained in Subsection 2.2.1. A common problem of such coordinates are the existence of points where longitudes and latitudes converge, e.g., the poles for geographical coordinates. This is avoided by introducing sophisticated compositions of orthogonal curvilinear grids in order to move the singularities over landmass as illustrated in Figure 6. A monograph on these grids, which are also used in the Max-Planck Institute ocean model, is written by Murray [98].

When finite differences are used to discretize the ocean primitive equations different possibilities arise how to position the prognostic variables relative to each other which leads to a staggered grid. A thorough analysis of different kinds of staggered grids for the ocean

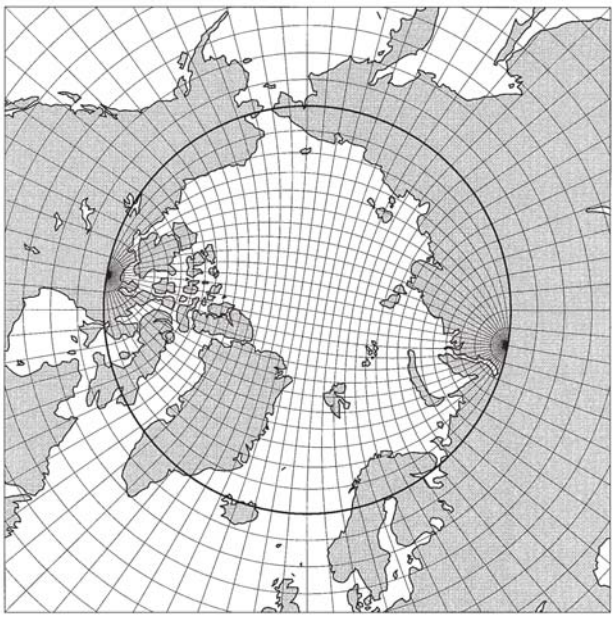


Figure 6: A grid composed of a geographical coordinates grid (outside of the bold circle) and an asymmetric bipolar grid (inside of the bold circle) to avoid the singularity at the north pole caused by a standard geographical coordinates grid [98]. In MPIOM this grid is referred to as *tripolar* grid.

primitive equations was conducted by Arakawa and Lamb [7] whereon the conclusion for the Max-Planck Institute ocean model was based that the Arakawa C-Grid, as illustrated in Figure 7, is the most appropriate. Since the barotropic subsystem is a two dimensional problem as shown in Subsection 2.3.2, only one grid layer (namely the topmost layer) is necessary. In contrast, the baroclinic subsystem uses up to 80 vertically stacked layers of the illustrated Arakawa C-Grid.

Using finite differences on a staggered Arakawa C-Grid for the spatial discretization of the barotropic subsystem (2.46), (2.47), (2.48) with dimension m in zonal and n in meridional direction results in a symmetric, positive definite, diagonal dominant matrix with non-positive off-diagonal entries and positive diagonal entries $A = (a_{ij}) \in \mathbb{R}^{N\times N}$ with $N = mn$ and a right hand side $b \in \mathbb{R}^N$. Due to the periodic boundary conditions in zonal direction and the tripolar grid which introduces *doubly periodic* boundary conditions [98] at the north pole, the structure of A strongly depends on the ordering of the unknowns.

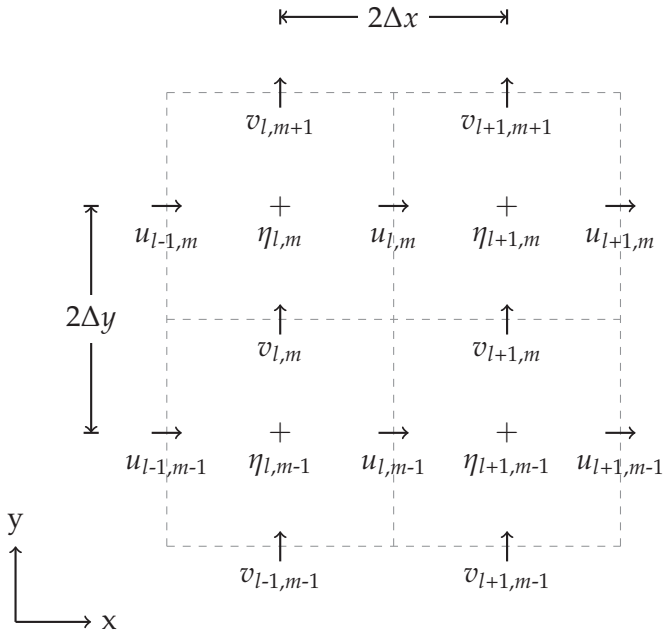


Figure 7: A staggered Arakawa C-Grid with meridional velocity v, zonal velocity u and sea-surface elevation η [7]. The discretization length in x-direction (resp. y-direction) is denoted by Δx (resp. Δy).

Applying a lexicographical ordering, i.e., starting at the north pole and going in east then south direction, we have that

$$A = \begin{pmatrix} D_1 & S_1 & & & \\ S_1 & D_2 & S_2 & & \\ & S_2 & \ddots & \ddots & \\ & & \ddots & \ddots & S_n \\ & & & S_n & D_n \end{pmatrix}$$

with diagonal matrices $S_k \in \mathbb{R}^{m\times m}$ and matrices $D_k = (d_{ij}) \in \mathbb{R}^{m\times m}$, $k = 1, \ldots, n$ with

$$D_1 = \begin{pmatrix} d_{11} & d_{12} & & & & & d_{1,n\text{-}1} \\ d_{21} & d_{22} & d_{23} & & & d_{n\text{-}2,2} & \\ & d_{32} & \ddots & \ddots & \iddots & & \\ & & \ddots & \ddots & \ddots & & \\ & & \iddots & \ddots & \ddots & \ddots & \\ & d_{n\text{-}2,2} & & & \ddots & \ddots & d_{n,n\text{-}1} \\ d_{n\text{-}1,1} & & & & & d_{n\text{-}1,n} & d_{nn} \end{pmatrix}$$

and

$$D_k = \begin{pmatrix} d_{11} & d_{12} & & & d_{1,n\text{-}1} \\ d_{21} & d_{22} & d_{23} & & \\ & d_{32} & \ddots & \ddots & \\ & & \ddots & \ddots & d_{n,n\text{-}1} \\ d_{n\text{-}1,1} & & & d_{n\text{-}1,n} & d_{nn} \end{pmatrix} \text{ for } k \neq 1.$$

A detailed derivation of the components of A and b can be found in Wolff et al. [142, p. 35].

The matrix A can be simplified by extending it with additional dependent variables at the boundary, i.e., if x_i is a variable at the periodic boundary, we add a variable $x'_i = x_i$, in order to get an extended matrix A'. In this way, the matrix A' exhibits a tridiagonal structure, i.e.,

$$A' = \begin{pmatrix} D'_1 & S'_1 & & & \\ S'_1 & D'_2 & S'_2 & & \\ & S'_2 & \ddots & \ddots & \\ & & \ddots & \ddots & S'_n \\ & & & S'_n & D'_n \end{pmatrix}, \tag{2.49}$$

with diagonal matrices S'_i and tridiagonal matrices D'_i , $i = 1, \ldots, n$.

2.5 SUMMARY AND CONCLUSION

In this chapter we pointed out all necessary assumptions and conditions to derive the ocean primitive equations from a set of physical axioms. Based on that, we showed how the velocity field in these equations is split into baroclinic and barotropic components that can be solved consecutively. We then discretized the Laplace-like barotropic subsystem to obtain a system of linear equations that can be solved by a computer. Within MPIOM, one of the main challenges is to solve this system of equations efficiently. If we consider a quite coarse earth grid with a resolution of about 10 km sidelength, the numbers of unknowns already rise in the order of millions. At the present day, solving such systems repeatedly requires large compute clusters to become feasible. Additionally, in large models like MPIOM, the shear amount of data is so huge that one compute node in a cluster can hold only a small fraction of the whole data set. To exemplify this, Table 1 shows a bold and simple comparison of the computing power and memory between a standard MacBook Pro laptop and the Blizzard compute cluster at the DKRZ. We see from this comparison that a compute cluster surmounts a standard laptop by a factor of 10,000 in terms of computing power and memory. Theoretically, this would allow Blizzard to perform the same task as the MacBook Pro laptop 10,000 times faster or, from another perspective, a 10,000 times more

	MacBook Pro	Blizzard	Ratio
Computing power	100 Gigaflop/s	158 Teraflop/s	$6.33 \cdot 10^4$
Memory	4 GB	21 TB	$1.86 \cdot 10^4$

Table 1: An illustrative comparison of a standard 17 inch Apple MacBook Pro laptop (fall 2011 model) and the Blizzard cluster. More details on Blizzard can be found in Appendix B.

complex problem in the same time. If we imagine that today huge climate simulations already have a runtime of several weeks to months on large compute cluster, we can grasp the importance of HPC.

The real challenge now lies in the exploitation of this computing power by a solver for the barotropic subsystem. An obvious and necessary condition for this is a parallel algorithm that is able to run efficiently on thousands of cores. Here, also the *scalability* of the algorithm becomes an important property. In the context of HPC, we differentiate between *strong scaling* and *weak scaling*. In the former case we examine how the runtime of an algorithm with a fixed input data set changes when the number of processors increases. In the latter case, we examine the runtime when the input data increase as the number of processors increases such that the work of one processor is fixed. The behavior of an algorithm under these conditions determines his scalability.

An important aspect of a cluster is the interconnection of its compute nodes. Typically, there is a large gap between the computing power of a cluster and the bandwidth of its interconnection [64]. This leads to bad scalability of an algorithm that stresses the interconnection too much and therefore keeps the processors waiting for new data to process. It is an important point to consider this aspect when choosing an iterative solver for a large sparse system of linear equations like the barotropic subsystem. In the following chapter we will address this challenge by presenting and evaluating different approaches to this problem.

3

PARALLEL SOLVERS AND PRECONDITIONERS FOR THE OCEAN/SEA-ICE MODEL MPIOM

The barotropic subsystem as introduced in Subsection 2.3.2 is an elliptic partial differential equation that commonly emerges as an intermediate step in solving the ocean primitive equations as derived in Chapter 2 [124]. This chapter addresses the challenge of numerically solving the discretized barotropic subsystem efficiently on HPC hardware. This is done in the context of the ScalES project with its goal to improve the solver of the barotropic subsystem with respect to its runtime.

In the case of MPIOM, the traditional solver for the barotropic subsystem was hard-wired into the ocean model. From this, the challenge was defined to separate the solver from the model itself into a dedicated library. This library, which we call *ScalES-Lib*, also helps to improve MPIOM regarding its flexibility. Since ScalES-Lib implements various solver techniques it is possible to select the most suitable solver and preconditioner depending on the scale of the problem and hardware setup, i.e., the number of compute nodes and processors.

The particular feature of ScalES-Lib compared to other libraries like PETSc [12] and the Trilinos Project [66] is its orientation towards stencil-based symmetric positive definite systems in legacy ocean models like MPIOM. With this focus it was possible to use special data structures and to avoid any overhead that comes with larger, more general libraries. Additionally, the barotropic subsytem in MPIOM is only implicitly defined by stencil operations and complex boundary exchange functions. Interfacing with a solver library that expects a matrix representation for certain operations, e.g., preconditioning, and uses its own structure for distributed arrays is a complex endeavor with uncertain outcome. For these reasons the application of an external solver library was not pursued in the ScalES project.

We will start with introducing the solvers of ScalES-Lib, followed by the preconditioners and conclude the chapter with numerical benchmarks. A special emphasize was put on a cache efficient implementation of the solvers and preconditioners that will be presented. The cache is a small and fast memory component, compared to the main memory of a processing unit, that transparently stores read values as well as intermediate results in order to speed up their later retrieval. If a data element is stored in the cache, we have a *cache hit* in lieu of a *cache miss* if data need to be retrieved from the slower main memory. Cache-efficient programming is characterized by the attempt to achieve as many cache hits as possible in an algorithm. Therefore,

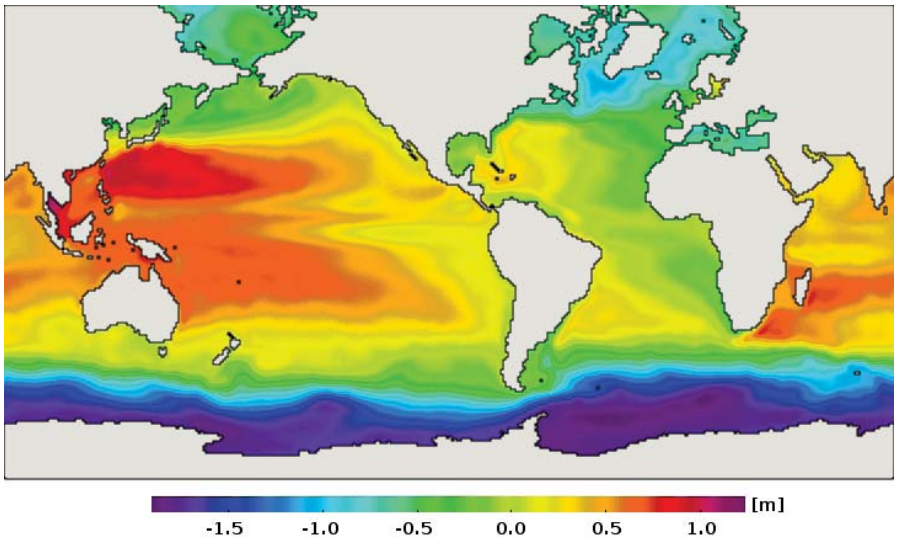


Figure 8: The sea-surface height above sea level in meter as calculated in each timestep in the barotropic subsystem.

certain assumptions about the inner workings of the cache need to be taken. For instance, data that were accessed recently or frequently are stored in the cache.

A special aspect of a parallel program running on a large supercomputer is the fact that the speed of data being sent over a network interconnect is determined by the network's *latency* and *bandwidth*. The latency can be defined as the time a small message, i.e., a message with a single data element, needs to travel from source to destination. On the other hand the bandwidth of a network determines the amount of data in bits that can be transferred in a second. For the communication of large data sets the bandwidth is the decisive factor and the latency can almost be neglected. Obviously, this is not the case for the communication of small data sets where the latency becomes the crucial factor. In case of our parallel solvers, the amount of data that needs to be transferred during an exchange operation and especially for an inner product is small and thus we are typically bound by latency. Frequent communication of small messages will have a negative impact on the performance of a parallel program.

To give a foretaste of what is to come, Figure 8 shows the visualized solution, i.e., the sea-surface elevation, of the barotropic subsystem.

3.1 CURRENT SOLVERS OF MPIOM

In Section 2.4, we showed that the discretized barotropic subsystem with dimension m in zonal and n in meridional direction is a sym-

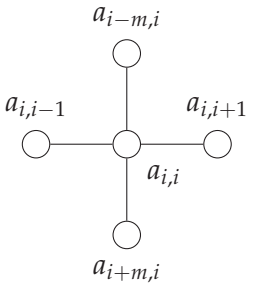


Figure 9: Five-point stencil of the barotropic subsystem.

metric, positive definite, diagonal dominant matrix $A = (a_{ij}) \in \mathbb{R}^{N \times N}$ with $N = mn$ that has block tridiagonal structure, i.e.,

$$A = \begin{pmatrix} D_1 & S_1 & & & \\ S_1 & D_2 & S_2 & & \\ & S_2 & \ddots & \ddots & \\ & & \ddots & \ddots & S_n \\ & & & S_n & D_n \end{pmatrix} \tag{3.1}$$

with diagonal matrices $S_i \in \mathbb{R}^{m \times m}$ and tridiagonal matrices $D_i \in \mathbb{R}^{m \times m}$, $i = 1, \ldots, n$.

Due to this structure, A can also be formulated as a five-point stencil as illustrated in Figure 9. This allows storing A with the help of simpler data structures compared to more general sparse matrix formats like CSR or CSC [119] as well as the realization of the structured Arakawa C-Grid by means of 2D arrays.

Following this approach, in MPIOM the desired solution of the barotropic subsystem η^{n+1} is stored in an array `Z10(i,j)` with $i = 1, \ldots, m$ and $j = 1, \ldots, n$, where i represents a zonal (west–east) and j a meridional (north–south) coordinate. The corresponding barotropic stencil is defined as a set of three 2D arrays; `FF` for the central part, `UF` for the zonal arms and `VF` for the meridional arms of the stencil at coordinate (i, j). Keeping the staggered Arakawa C-Grid in mind an application of the stencil to `Z10(i,j)` translates to

$$\begin{aligned} \texttt{FF(i,j)} * \texttt{Z10(i,j)} &- \texttt{UF(i,j)} * \texttt{Z10(i+1,j)} \\ &- \texttt{UF(i-1,j)} * \texttt{Z10(i-1,j)} \\ &- \texttt{VF(i,j)} * \texttt{Z10(i,j+1)} \\ &- \texttt{VF(i,j-1)} * \texttt{Z10(i,j-1)}. \end{aligned}$$

Parallelization in MPIOM is accomplished by a uniform block decomposition of the 2D arrays into $z = x \cdot y$ rectilinear partitions for a total of z processes with x partitions in zonal direction and y partitions

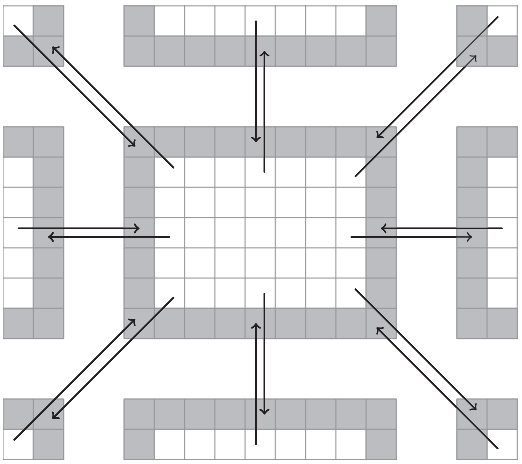


Figure 10: Boundary exchange of one partition (white area) with its neighbors. The gray area represents one or more boundary halos [15].

in meridional direction. Consequently any partition has four direct neighbors sharing an edge and four diagonally neighboring partitions as illustrated in Figure 10. Communication is performed through additional boundary halos, implemented by appropriately enlarged arrays, that overlap the neighboring partitions and are updated with the according values each time a boundary exchange function is called. In case of only one boundary halo, communication is only necessary with four neighboring partitions due to the structure of the stencil. Depending on the application, using more halos can reduce the number of necessary boundary exchanges. This results in shorter communication time per data, because of fewer communication operations and therefore fewer latencies.

Traditionally, the barotropic subsystem was solved with the SOR method in MPIOM which is a splitting method and will be elaborated in Subsection 3.3.1. For a linear equation $Ax = b$, SOR is based on the splitting $\omega A = (D + \omega L) - ((1 - \omega)D - \omega U)$, where D is the diagonal of A, L its strict lower part, $U = L^T$ its strict upper part and ω a relaxation parameter with $\omega \in (0, 2)$. Hence, the iteration scheme for x_k is

$$(D + \omega L)x_{k+1} = ((1 - \omega)D - \omega L^T)x_k + \omega b$$

which can be easily translated to a stencil formulation.

Parallelization of SOR is accomplished by a *red-black* or *checkerboard* ordering [119] that allows parallel treatment of points with the same color as shown in Figure 11. The drawback of this ordering is the necessity of two boundary exchanges per iteration if only one boundary halo is used. Therefore, two boundary halos are used in MPIOM following the recommendations of Beare and Stevens [15]. The most important relaxation parameter ω, with regard to the rate of

convergence, is estimated by a number of test calculations measuring the rate of convergence followed by manual fine tuning. While SOR proved to be quite efficient for small problems an ever increasing number of unknowns due to finer grids distributed on an increasing number of processes exposed its lack of scalability. The reason for this is that the number of necessary SOR iterations to approximate the solution rapidly increases with the size of the problem. This renders the interconnect to be the major bottleneck because in each iteration all neighboring processes communicate. Still, on a small model problem running on a single cluster node an iterative method like SOR with a large number of necessary iterations can be the method of choice given that the interprocess communication is fast.

Another property of SOR is that it provides no implicit way to check for the quality of the current iteration, meaning that additionally the calculation of a residual would be needed to assure the quality of the solution. In the current SOR implementation this is omitted, meaning that one needs to preset a fixed number of iterations. A major task in the ScalES project was to resolve these limitations by replacing SOR with the help of a library that provides parallel solvers and preconditioners.

3.2 NEW SOLVERS FOR MPIOM

3.2.1 *Conjugate Gradient Method*

For solving a sparse symmetric positive definite linear system, the conjugate gradient (CG) method is one of the best known iterative techniques [119]. It belongs to the class of projection methods onto Krylov subspaces and was first proposed in 1952 by Hestenes and Stiefel [67] as a possible direct solver as well as iterative solver [102]. But it was first until the work of [115] that the CG method became a standard method for solving large symmetric, positive definite systems

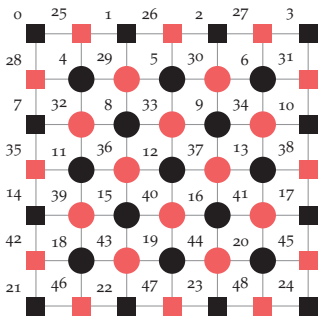


Figure 11: Example of a *red-black* or *checkerboard* ordering on a 7×7 grid. The rectangles denote preset boundary values. The five-point stencil is applied to all inner grid points, i.e., circles.

Algorithmus 3.1 Conjugate gradient method

1: $r_0 = b - Ax_0$
2: $p_0 = z_0$
3: **for** $k = 0, 1, \ldots, k_{max}$ **do**
4: $\quad \alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}$
5: $\quad x_{k+1} = x_k + \alpha_k p_k$
6: $\quad r_{k+1} = r_k - \alpha_k A p_k$
7: $\quad$ **if** $r_{k+1}^T r_{k+1} < TOL$ **then**
8: $\quad\quad$ exit loop
9: $\quad$ **end if**
10: $\quad \beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
11: $\quad p_{k+1} = r_{k+1} + \beta_k p_k$
12: **end for**

on parallel computers. For a thorough derivation of the CG method the reader is kindly referred to [4, 119, 123].

The CG method, as shown in Algorithm 3.1, consists basically only of three building blocks: vector operations, matrix-vector multiplications and dot products, whose efficient implementation determines much of the resulting scalability of the algorithm. Besides the operations in each iteration, the overall number of iterations needed to satisfy a given tolerance is of utmost importance. According to the convergence theory of CG, this number depends on the condition number $\kappa(A)$ through the relation

$$\begin{aligned} \frac{\|e_k\|_A}{\|e_0\|_A} &\leq 2 \left[\left(\frac{\sqrt{\kappa(A)}+1}{\sqrt{\kappa(A)}-1} \right)^k + \left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1} \right)^k \right]^{-1} \\ &\leq 2 \left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1} \right)^k, \end{aligned} \tag{3.2}$$

where $e_k = x_k - x$ is the error in the k^{th} iteration and $\|\cdot\|_A$ the energy norm [63]. By this inequality we know that in order to reduce the initial error e_0 by a factor of ϵ a maximum of

$$\frac{1}{2}\sqrt{\kappa(A)}\ln(\frac{2}{\epsilon}) + 1 \tag{3.3}$$

iterations are needed [9]. We will call an x_k that satisfies $e_k \leq \epsilon e_0$ an *ϵ-approximate solution*. It should also be noted that this bound is overly pessimistic since depending on the clustering of the eigenvalues of A even a superlinear rate of convergence can be observed in practice [16].

This inequality justifies the application of a preconditioner M where $\kappa(M^{-1}A) \ll \kappa(A)$ and conclusively fewer iterations are necessary to solve the equivalent system $M^{-1}Ax = M^{-1}b$. The preconditioner M needs to be symmetric and positive definite, so that L exists with $M =$

Algorithmus 3.2 Preconditioned conjugate gradient method

$r_0 = b - Ax_0$
$z_0 = M^{-1}r_0$
$p_0 = z_0$
for $k = 0, 1, \ldots, k_{max}$ **do**
  $\alpha_k = \frac{r_k^T z_k}{p_k^T A p_k}$
  $x_{k+1} = x_k + \alpha_k p_k$
  $r_{k+1} = r_k - \alpha_k A p_k$
  $z_{k+1} = M^{-1} r_{k+1}$
  if $r_{k+1}^T z_{k+1} < TOL$ **then**
    exit loop
  end if
  $\beta_k = \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k}$
  $p_{k+1} = z_{k+1} + \beta_k p_k$
end for

LL^T, to sustain those properties for the CG method [119]. Without a preconditioner the number of necessary iterations for an *ill-conditioned* matrix A, i.e., $\kappa(A)$ is large, is too high and the resulting number of network communications impairs the scalability and overall runtime.

In Algorithm 3.2 the pseudo code of the preconditioned conjugate gradient (PCG) method is illustrated. The implementation in MPIOM was done by the author in a generic way, meaning that the five-point stencil of the barotropic subsystem is provided as a function parameter to the PCG function so that it is independent of the actual stencil implementation. Each process performs the stencil operation on its local partition followed by a boundary exchange to update the boundary halos. In the same way it is possible to have different kinds of preconditioners M^{-1} that will be presented in Section 3.3. To calculate the dot product local dot products are summed up using the sum reduction (`MPI_Allreduce`, `MPI_SUM`) of the MPI library. It is possible to switch to routines from the Basic Linear Algebra Subprograms (BLAS) library for vector operations and dot products with a preprocessor flag. Our modular implementation allows to facilitate the reuse of the code in other projects.

3.2.2 *Chebyshev Iteration*

The Chebyshev iteration can be directly motivated from the convergence proof of the CG method. Therein, the upper bound (3.2) for the k^{th} iteration of the CG method is derived with the help of the Chebyshev polynomials $T_n(x) = \cos(n \arccos(x))$, $x \in [-1, 1]$ as well as the fact that $f(x) = 1/2^{n-1} T_n(x)$ is the minimal polynomial of degree n with leading coefficient 1 on $[-1, 1]$ in the ∞-norm [119, 130]. This motivates the direct application of Chebyshev polynomials to reduce

Algorithmus 3.3 Preconditioned Chebyshev iteration

1: $\theta = (\lambda_{max} + \lambda_{min})/2$
2: $\delta = (\lambda_{max} - \lambda_{min})/2$
3: $\sigma = \theta/\delta$
4: $r_0 = b - Ax_0$
5: $z_0 = M^{-1}r$
6: $\rho_0 = 1/\sigma$
7: $d_0 = \frac{1}{\theta}z_0$
8: **for** $k = 0, 1, \ldots, k_{max}$ **do**
9: $\quad x_{k+1} = x_k + d_k$
10: $\quad r_{k+1} = r_k - Ad_k$
11: $\quad z_{k+1} = M^{-1}r_{k+1}$
12: $\quad$ **if** $z_{k+1}^T z_{k+1} < TOL$ **then**
13: $\quad\quad$ exit loop
14: $\quad$ **end if**
15: $\quad \rho_{k+1} = (2\sigma - \rho_k)^{-1}$
16: $\quad d_{k+1} = \rho_{k+1}\rho_k d_k + \frac{2\rho_{k+1}}{\delta}z_{k+1}$
17: **end for**

the initial error which results in the *Chebyshev iteration,* as shown in Algorithm 3.3, that also exhibits the same rate of convergence (3.2) as the CG method.

Since the Chebyshev iteration uses no inner products, it is especially suited for massively parallel computers where internode communication is usually expensive [60]. As a downside, good approximations of the eigenvalues λ_{min} and λ_{max} of the symmetric, positive definite matrix A are needed. Overestimating (resp. underestimating) λ_{max} (resp. λ_{min}) impairs the convergence rate while the underestimation of λ_{max} (resp. overestimation of λ_{min}) can even result in divergence. An easy way to obtain these estimations is given by the power method:

$$v_{k+1} = Aw_k, \quad w_{k+1} = \frac{v_{k+1}}{\|v_{k+1}\|}, \quad \lambda_{k+1} = v_{k+1}w_{k+1}.$$

If a symmetric, positive definite matrix A has only one unique eigenvalue λ_{max} of maximum modulus and w_0 has a component associated with the subspace of λ_{max}, than λ_k converges to λ_{max} and w_k to the corresponding eigenvector [54]. Having obtained an approximation of λ_{max}, we define $B = \lambda_{max}I - A$ where I is the identity matrix. Obviously, the largest eigenvalue of B is $\lambda_{max} - \lambda_{min}$ and therefore we can apply the power method to B in order to obtain λ_{min}.

The implementation of the Chebyshev iteration in MPIOM was analogously done as the CG method to allow for varying preconditioners and optional BLAS optimizations. To preserve the advantage of not needing inner products, we implemented two parameters that allow to control the frequency in which the norm of the current approximate solution is checked for a given tolerance. The first parameter allows

to define a minimum number of iterations in which no convergence check is done to save unnecessary communication. Typically, this number can be easily determined by a number of test runs to obtain an average number of iterations that are needed to solve the barotropic subsystem. The second number allows to specify the frequency of the convergence checks which should be chosen with caution since it increases the number of performed iterations and can therefore out weight the benefits.

3.2.3 *Additive Schwarz Method*

The CG method as well as the Chebyshev method can be used as a *global solver* to solve the system of linear equations of a discretized elliptic partial differential equation, $Au = f$ on a given grid Ω and $u = g$ on $\partial\Omega$, as a whole. The parallelism of the solver is then determined solely by the parallel characteristics of the solver's algorithm. Another approach is to split the grid Ω into overlapping subgrids $\Omega_1, \Omega_2, \ldots$ which dates back to Schwarz [121] in 1870. Following standard references [109, 125, 134], we derive the *additive Schwarz* method. For the sake of simplicity, we assume two subdomains Ω_1, Ω_2.

We let $\Gamma_1 = \partial\Omega_1 \cap \Omega_2$ (resp. $\Gamma_2 = \partial\Omega_2 \cap \Omega_1$) be the artificial boundary of Ω_1 (resp. Ω_2) in Ω and denote with an index $i = 1, 2$ the restrictions on Ω_i, e.g., $u_1 = u|_{\Omega_1}$. Furthermore, let R_i be the rectangular restriction matrix for the grid Ω_i, i.e., $u_i = R_i u$.

Using a *Richardson iteration* approach, we let $r = f - Au$ and define two subproblems:

$$A_i u_i = r_i \text{ in } \Omega_i, \qquad u_i = 0 \text{ on } \partial\Omega_i$$

for $A_i = R_i A R_i^T$ and $i = 1, 2$. For an initial value u^0, we define the additive Schwarz iteration

$$u^{n+1} = u^n + \sum_{i=1}^{2} R_i^T A_i^{-1} R_i (f - Au^n) \tag{3.4}$$

which only converges if $\rho(I - BA) < 1$ where $B = \sum_{i=1}^{2} R_i^T A_i^{-1} R_i$ and I is the unity matrix. In general, an appropriate damping factor ω for BA must be chosen to assure convergence. The calculation of $A_i^{-1} r_i^n$, $i = 1, 2$ can be done in parallel since A_1 and A_2 are completely decoupled. Calculating first u_1^{n+1} and updating with this the values in $\Omega_1 \cap \Omega_2$ before calculating u_2^{n+1} results in the *Multiplicative Schwarz* method that converges faster but does no longer allow for parallel treatment. The relation between Multiplicative and Additive Schwarz has therefore many similarities to the relation between the Gauß-Seidel and Jacobi method.

As a *local solver* for A_i^{-1} a direct solver or an iterative method like CG method can be chosen. Since B can be seen as a preconditioner

for A in the Richardson iteration, an obvious possibility is to replace the Richardson iteration with another iteration method that exhibits a better rate of convergence like the Chebyshev or CG method.

Finally, it should be noted that the generalization of the additive Schwarz method to more than two subgrids is straightforward and was omitted for notational simplification. In MPIOM, the author implemented the Richardson iteration with a general solve function which then calls the solver and preconditioner depending on some configuration parameters. For the implementation of the overlapping halos we could reuse the `sethaloN` function which is used by the SOR solver to setup its two boundary halos and for boundary exchange.

3.3 PARALLEL PRECONDITIONERS

To reduce the number of necessary iterations of an iterative solver like the CG method and the Chebyshev iteration a symmetric, positive definite preconditioner B is used. Traditionally, we demand the following properties of B:

- strong reduction of the number of iterations,
- fast application to a vector, i.e., $B^{-1}r$ is easy and fast to compute,
- low memory profile, i.e., B or B^{-1} is sparse.

With the advent of highly parallel supercomputers our second postulation basically translates to:

- $B^{-1}r$ is highly parallel to compute in order to exploit parallel hardware.

With increasing parallelization laid out in a hierarchy (cluster, nodes, CPUs, cores, ...) the speed of the interconnect plays a more and more import role. For each CG iteration the necessary matrix-vector multiplication, i.e., the application of the stencil, obviously demands communication on all levels of the interconnect hierarchy. For the application of the preconditioner this is not necessarily the case. Most preconditioners can be applied in a block-Jacobi fashion, i.e., the unknowns of A are partitioned with no coupling between two partitions leading to a block structure of the preconditioner B. This strategy saves communication overhead and allows for more parallelism since each block can be treated independently. If we number

the unknowns block by block and lexicographically inside a block we have $B = (b_{ij}) \in \mathbb{R}^{N\times N}$ with $N = mn$. The structure of B is

$$B = \begin{pmatrix} B_1 & & & & \\ & B_2 & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & B_n \end{pmatrix}, \quad B_i = \begin{pmatrix} D_1 & S_1 & & & \\ S_1 & D_2 & S_2 & & \\ & S_2 & \ddots & \ddots & \\ & & \ddots & \ddots & S_{k_i} \\ & & & S_{k_i} & D_{k_i} \end{pmatrix}$$

with diagonal matrices $S_i \in \mathbb{R}^{l_i\times l_i}$ and tridiagonal matrices $D_i \in \mathbb{R}^{l_i\times l_i}$, where l_i is the horizontal length and k_i the vertical length of the i^{th} Jacobi block for $i = 1, \ldots, n$. The downside to this approach is that dropping couplings mostly impairs the reduction of the number of iterations.

We have implemented several types of parallel preconditioners in MPIOM which are to be presented in the remaining section following [13, 119]. In MPIOM, we apply these preconditioners inside Jacobi blocks which are naturally given by the partitioning of the unknowns that the model code imposes. Therefore, one processing unit works on exactly one partition, i.e., one Jacobi block, which allows us to use the boundary exchange functions for the solver as illustrated in Figure 10. The downside is that it impedes the possibility to apply parallel techniques inside one Jacobi block for the preconditioner with the help of OpenMP [19] for instance. Additionally, changing the given partitioning in the way that two ore more processing units work together on one partitioning would impede the execution of other parts of MPIOM which cannot exploit more than one processing unit per partition.

3.3.1 *Matrix Splitting Preconditioners*

The first iterative solvers were based on the idea of consecutively altering one unknown of a linear system with the others fixed by an approximate solution x_n in order to obtain an new approximation x_{n+1}. This process is consequently called *relaxation*. By splitting the matrix of a symmetric linear system $Ax = b$ into $A = D + L + U$, where D is the diagonal of A, L its strict lower part, $U = L^T$ its strict upper part, this idea can be formally expressed as:

$$Dx_{k+1} = -(L + L^T)x_k + b$$

and is known as *Jacobi iteration*. Instead of using only x_n when calculating a component of x_{n+1}, it is possible to take into account the already determined components of x_{n+1} where available which results in the *Gauß-Seidel* iteration:

$$(D+L)x_{k+1} = -L^T x_k + b.$$

Relaxing the i^{th} unknown $x_i^{(k)}$ in the k^{th} iteration with Gauß-Seidel, we obtain the i^{th} component of the new approximate x_i^{GS}. This can be considered as a non-optimal step with step length 1 in direction $x_i^{GS} - x_i^{(k)}$ from $x_i^{(k)}$. Another enhancement is achieved by the idea of introducing a generic step length ω, which leads to the SOR formulation

$$x_i^{(k+1)} = \omega x_i^{GS} + (1-\omega)x_i^{(k)}$$

for $i = 1, \ldots, n$. In matrix notation, we have

$$(D+\omega L)x_{k+1} = ((1-\omega)D - \omega L^T)x_k + \omega b$$

for the new approximation x_{k+1}. Evidently, the Gauß-Seidel method is a special case, i.e., $\omega = 1$, of the SOR method. From the theory of SOR we know that it diverges if $\omega \notin (0,2)$ [76] and that an optimal ω_{opt} can be found with the help of the spectral radius ρ of the Jacobian iteration matrix $B = -D^{-1}(L+U)$ [13, p. 11] and

$$\omega_{opt} = \frac{2}{1+\sqrt{1-\rho^2}}. \tag{3.5}$$

The splitting methods explained so far belong to the class of *linear stationary iterative methods of first degree* with general form

$$x_{k+1} = Gx_k + k,$$

where G is called the *iteration matrix* [143, p. 64]. A necessary and sufficient condition for the convergence of these methods is that the spectral radius $\rho(G)$ satisfies $\rho(G) < 1$. If A is strictly diagonal dominant or diagonal dominant and irreducible, this condition is necessarily satisfied [25, p. 186]. For the SOR and Gauß-Seidel ($\omega = 1$) iteration, we additionally have that if A is symmetric and $0 < \omega < 2$, the SOR iteration converges if and only if A is positive definite by the Ostrowski-Reich theorem [25, p. 188].

In order to use these methods as preconditioners, a reformulation as *error-correction* method is necessary, i.e., we set $\Delta x_{k+1} = x_{k+1} - x_k$ and determine in each iteration Δx_{k+1} instead of x_{k+1}, resulting in $\Delta x_{k+1} = M^{-1} r_k$ where M is the *preconditioning matrix* or just the *preconditioner*. For Jacobi, Gauß-Seidel and SOR, we have:

$$\begin{aligned} M_{JA} &= D, \\ M_{GS} &= D + L, \\ M_{SOR} &= \omega^{-1}(D + \omega L). \end{aligned}$$

Since M_{GS} and M_{SOR} are not symmetric, both preconditioners do not qualify for the application in the CG method. In both methods the asymmetry occurs due to the ordering in which the components of the new approximation are updated. In our case, a lexicographical ordering, i.e., $1, 2, \ldots n$ was applied which is termed *forward sweep*. By inverting the ordering we obtain a *backward sweep*, which applied after a forward sweep results in the *symmetric Gauß-Seidel* (SGS) and resp. the *symmetric SOR* (SSOR) method. We therefore have

$$M_{SGS} = (D + L^T)D^{-1}(D + L),$$
$$M_{SSOR} = (\omega(2 - \omega))^{-1}(D + \omega L^T)D^{-1}(D + \omega L).$$

In MPIOM a block-Jacobi SSOR preconditioner was implemented by the author. Due to the stencil formulation, the implementation was straightforward and therefore the details are left out here. The optimal relaxation parameter was determined with (3.5) as suggested in [110] with the help of the power method to determine ρ.

3.3.2 *Incomplete Factorization Preconditioners*

Incomplete factorization techniques are based on the idea of performing an incomplete Gaussian elimination of a matrix A in the sense that the factors L and U are sparse with respect to certain predefined constraints. We therefore have $A = LU - R$, where L is a sparse lower triangular, U a sparse upper triangular and R the residual matrix [13, 119]. Dropping R now leads to the *incomplete LU* (ILU) preconditioner $M = LU$ which are guaranteed to exist if A is an M-matrix [31]. Applying the preconditioner to a vector r involves doing a forward and backward substitution, i.e., solving $Ly = r$ followed by $Ux = y$.

If we let L (resp. U) have the same sparsity pattern as the lower (resp. upper) triangular part of A, we have have no additional *fill-in* and write ILU(0). Increasing the fill-in, which can be specified in a *level of fill-in*, reduces the nonzero components of R and improves the quality of M with the disadvantage of more needed storage and work when applying the preconditioner.

Since we are dealing with symmetric, positive-definite matrices, we will use the incomplete Cholesky decomposition (ICC) which reduces the memory consumption compared to ILU where two triangular matrices are needed to be stored. An exception is ILU(0) that allows for only storing the diagonal elements and reconstruct the off-diagonal elements during the substitution step [119, p. 303] and was therefore implemented in ScalES-Lib as well.

The ICC preconditioner as implemented in MPIOM consists of two steps: a setup step where the incomplete decomposition is calculated and its actual application in the CG algorithm. These steps are detailed in the following two subsections.

3.3.2.1 *Incomplete Cholesky Factorization*

Given that $A = (a_{ij})$ is a symmetric and positive definite matrix, there exists a unique factorization $A = LL^T$, where $L = (l_{ij})$ is a lower diagonal matrix with positive diagonal entries [54]. By applying the column-wise proceeding Cholesky-Crout algorithm, for $i = 1, \ldots, N$ we have

$$\begin{aligned} l_{ii} &= \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}, \\ l_{ji} &= \frac{1}{l_{ii}} \left(a_{ji} - \sum_{k=1}^{i-1} l_{jk} l_{ik} \right), \quad \text{for } j > i. \end{aligned} \tag{3.6}$$

The choice of a column-wise instead of a row-wise traversal is motivated by the fact that Fortran stores arrays column-wise, resulting in a higher cache-hit rate when accessed in the same manner, as we will see later. A complete factorization would lead to a dense matrix L, that must be avoided because of memory limitations and high computational costs, when doing forward and backward substitution. Therefore, we introduce for l_{ij} the common definition [119] of the initial level of fill-in

$$\text{lev}_{ij} = \begin{cases} 0 & \text{if } a_{ij} \neq 0 \text{ or } i = j \\ \infty & \text{otherwise} \end{cases},$$

and update the current level lev_{ij} when l_{ij} is updated according to

$$\text{lev}_{ij} = \min \left(\text{lev}_{ij}, \min_{l=1,\ldots,i-1} (\text{lev}_{jl} + \text{lev}_{il} + 1) \right).$$

When lev_{ij} exceeds a predefined maximum level of fill-in p we set $l_{ij} = 0$. Let L^p be the factor L due to the incomplete Cholesky decomposition with level of fill-in p. Analyzing the resulting sparsity pattern of L^p (see Figure 12) with the help of a naïve implementation in Matlab, we can easily derive a rule to predict the sparsity pattern. Let $\text{diag}_k(L) = (l_{i+k\,i})_{i=1,\ldots n-k}$ be the k^{th} lower diagonal. L^0 has the same sparsity pattern as the lower diagonal matrix of A meaning the non-zero elements are in $\text{diag}_k(L)$ with $k = 0, 1, m$, whereas L^1 gains additional elements in $\text{diag}_{m-1}(L)$. In case of L^p with $p \geq 2$ we have non-zero elements in $\text{diag}_k(L)$ with $k = 0, \ldots, p-1$ and $k = m-p, \ldots, m$. We can extend this rule to include the somewhat exceptional cases L^0 and L^1 to obtain in all

$$l_{ij}^p = 0 \text{ if } l_{ij}^p \notin \text{diag}_k(L) \text{ with } k = 0, \ldots, \max(1, p-1), m-p, \ldots, m. \tag{3.7}$$

The diagonals $\text{diag}_k(L)$ with $k = 0, \ldots, \max(1, p-1)$ will be denoted with *secondary diagonals* and $\text{diag}_k(L)$ with $k = m-p, \ldots, m$ with *outer diagonals*.

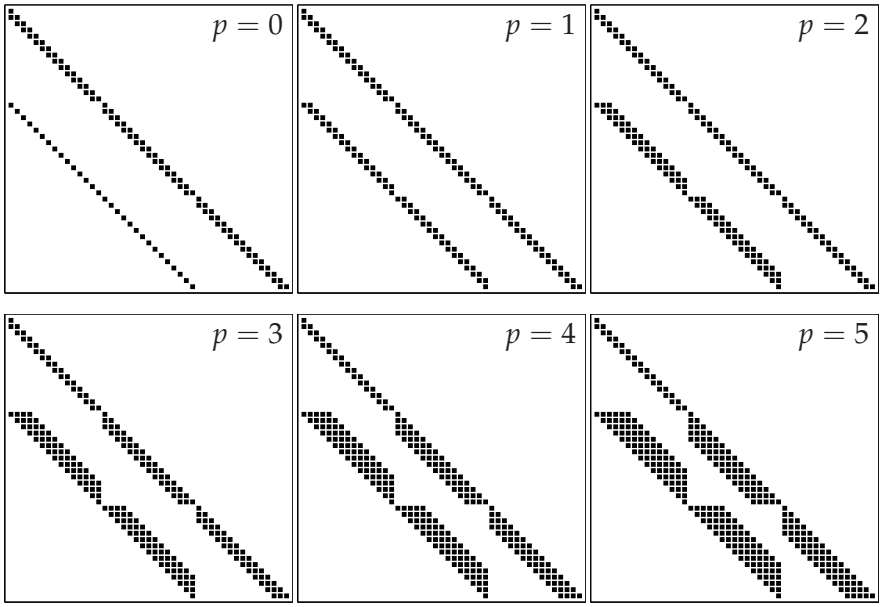


Figure 12: Fill-in pattern for ICC(p) with $p = 0, \ldots, 5$ in matrix representation.

Taking into account the diagonal growth of this pattern for increasing fill-in, the DIAG format [119] is a suitable storage format. The DIAG format stores elements along a diagonal in a two-dimensional array `DIAG(1:n,1:Nd)`, where `Nd` is the number of diagonals with nonzero entries. The offset of each diagonal from the main diagonal is saved in the array `IOFF(1:Nd)`. As in the Fortran 95 language the colon notation `A(i:j)` is used to account for the dimension of an array A or a slice consisting of the ordered elements `A(i), A(i+1),..., A(j)`. By adding an increment of `-1` as in `A(i:j:-1)` the order of the elements is reversed.

Considering the fact that we are dealing with a stencil, we modified the DIAG storage scheme into a more stencil compliant form. Transforming the matrix L^p to a stencil representation results in an ICC stencil as pictured in Figure 13. The western arm $l_{ii-1}, \ldots, l_{ii-(p-1)}$ of this stencil is stored in the three-dimensional array `ICC_W(1:MAX(1,p-1), 1:m,1:n)`, the inverse of the central element l_{ii} in the array `ICC_C(1:m, 1:n)` and the southern arm $l_{i+mi}, \ldots, l_{i+mi+p}$ in `ICC_S(1:p+1,1:m, 1:n)`. Again, the indexing of these arrays was chosen such that the element access in the forward and backward substitution step is most cache efficient. By virtue of storing $\frac{1}{l_{ii}}$ in `ICC_C` we can later, when applying the preconditioner, use a multiplication instead of a division which can be computed in 1 cycle compared to about 30 cycles on the IBM POWER6.

Calculating the ICC(p) decomposition with regard to the arrays of the ICC stencil is then accomplished by the implementation of (3.6) and a simple transformation that maps an element e_{ij} of a matrix to the according stencil at (x, y) and vice versa. To restrict the decomposition

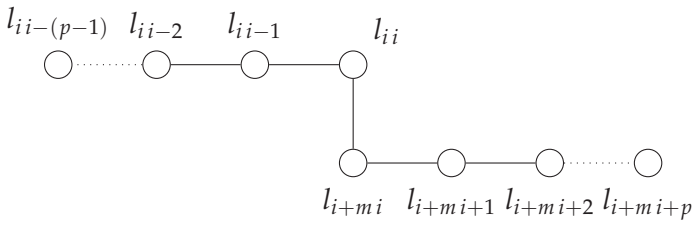


Figure 13: Stencil of the ICC(p) factorization

to a fill-in of level p and to avoid unnecessary calculations, only the non-zero elements according to (3.7) are considered as displayed in Algorithm 3.4.

To compensate for the discarded elements in an incomplete decomposition, a popular strategy is to add their sum to the diagonal such that the row sums of A and the product of the incomplete factors LL^T are the same [119, p. 305]. This can increase the quality of the preconditioner in reducing the number iterations [119]. Since we have $A = LL^T - R$ from [87], the i^{th} row sum is given as

$$a_{i*}e = \sum_{k=1}^{i} l_{ik} l_{*k} e - r_{i*}e, \tag{3.8}$$

where $e = (1, 1, \ldots, 1)^T$ and $*$ is used to denote a row or column vector. In order to eliminate the second residual term $r_{i*}e$, the diagonal components of the decomposition (3.6) need to be adjusted as

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2 - (r_{i*}e)}.$$

This modification establishes that $Ae = LL^T e$ and is therefore called *modified ICC* (MICC) which was additionally implemented in MPIOM.

3.3.2.2 *Applying the ICC(p) Preconditioner*

For a given residual $r \in \mathbb{R}^N$, where $r(i, j)$ with $i = 1, \ldots, m$ and $j = 1, \ldots, n$ presents the component at the grid point (i, j), we can now define the forward and backward substitution step in terms of the ICC stencil. In the following, the arms `ICC_W`, `ICC_S` and the inverse of the central element `ICC_C` of the ICC stencil will be abbreviated with W, S and C respectively.

The forward substitution for $Ly = r$, more precisely

$$y_i = \frac{1}{l_{ii}} \left(r_i - \sum_{k=1}^{i-1} l_{ik} y_k \right)$$

Algorithmus 3.4 Main loop of the ICC(p) decomposition. A and L as defined in (3.1) and (3.6).

```
DO i = 1,N
    ! treat main diagonal
    tmp = get_value_of_A(i,i)
    DO k = MAX(1,i-m),i-m+p
        tmp = tmp - get_value_of_L(i,k)**2
    ENDDO
    DO k = MAX(1,i-MAX(1,p-1)),i-1
        tmp = tmp - get_value_of_L(i,k)**2
    ENDDO
    tmp = SQRT(tmp)
    CALL set_value_of_L(i,i,tmp)

    ! treat secondary diagonals
    DO j = i+1,i+MAX(1,p-1)
        IF ( j > n ) EXIT
        tmp = get_value_of_A(j, i)
        DO k = MAX(1,i-m),i-m+p ! outer diagonals
            tmp = tmp - get_value_of_L(j,k) * get_value_of_L(i,k)
        ENDDO
        DO k = MAX(1,i-MAX(1,p-1)),i-1 ! secondary diagonals
            tmp = tmp - get_value_of_L(j,k) * get_value_of_L(i,k)
        ENDDO
        tmp = tmp / get_value_of_L(i,i)
        CALL set_value_of_L(j, i, tmp)
    ENDDO

    ! treat outer diagonals
    DO m_j = m_i+m_m-p,m_i+m_m
        ! same loop body as before
    ENDDO
ENDDO
```

Algorithmus 3.5 Applying the ICC(p) preconditioner

```
s_l = p + 1 ! length of southern arm of ICC stencil
w_l = MAX(1,p-1) ! length of western arm of ICC stencil

! 1.) Forward substitution
! first column
r(1,1) = r(1,1)*ICC_C(1,1)
DO i=2,m
    r(i,1) = (r(i,1) - ICC_W(1,i,1)*r(i-1,1))*ICC_C(i,1)
ENDDO
! all other columns
DO j = 2,n
    DO i = 1,m
        r(i,j) = ( r(i,j) - ICC_S(1:s_l,i,j)*r(i:i+s_l-1,j-1) &
               - ICC_W(1:w_l,i,j)*r(i-1:i-w_l:-1,j) )*ICC_C(i,j)
    ENDDO
ENDDO

! 2.) Backward substitution
! all but first column
DO j = n,2,-1
    DO i = m,1,-1
        r(i,j) = r(i,j)*ICC_C(i,j)
        r(i-1:i-w_l:-1,j) = r(i-1:i-w_l:-1,j) - r(i,j)*ICC_W(1:w_l,i,j)
        r(i:i+p,j-1) = r(i:i+p,j-1) - r(i,j)*ICC_S(1:s_l,i,j)
    ENDDO
ENDDO

! first column
DO i = m,2,-1
    r(i,1) = r(i,1)*ICC_C(i,1)
    r(i-1,1) = r(i-1,1) - r(i,1)*ICC_W(1,i,1)
ENDDO
r(1,1) = r(1,1)*ICC_C(1,1)
```

for $i = 1, \ldots, N$, can then be transformed into a stencil formulation, that is

$$y(i,j) = C(i,j) \cdot r(i,j) - \left(\sum_{k=1}^{p+1} S(k,i,j) \cdot r(i-1+k, j-1) - \sum_{k=1}^{\beta} W(k,i,j) \cdot r(i-k,j) \right) \tag{3.9}$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, n$, where $\beta = \max(1, p-1)$. We traverse $y(i,j)$ in a cache-friendly way, so that the inner loop iterates over i, to ensure optimal cache utilization. Treating the case $j = 1$ separately without the second term on the right-hand-side of (3.9) eliminates unnecessary calculations for the first column where no S (resp. no outer diagonals) exist. The actual code is shown in the first part of Algorithm 3.5. It should be noted that the result $y(i,j)$ is saved in $r(i,j)$ again to avoid storing an additional array.

Backward substitution, $L^T x = y$, is more challenging to accomplish in a cache efficient way, because the first index of W and S now present columns in L^T. To overcome this, the substitution is reordered to result in a column-wise operation. Instead of

$$x_i = \frac{1}{l_{ii}} \left(y_i - \sum_{k=i+1}^{n} l_{ki} x_k \right)$$

for $i = N, \ldots, 1$ we perform partial updates of x_i. We start by setting $x_i = y_i$ for all i. At first, the element x_N is updated to its final value by $x_N \leftarrow \frac{x_N}{l_{ii}}$. After this, the remaining x_i, $i = N-1, \ldots, 1$ get updated according to the N^{th} column by virtue of $x_i \leftarrow x_i - l_{ki} x_N$, which is an efficient saxpy operation. The whole process is now repeated for x_{N-1} until x_1.

This principle can be conveyed to the stencil formulation with some modifications. The saxpy operation can be split into two saxpy operations by S and W elements and is performed such that first $x(i,j)$ is multiplied by $C(i,j)$, then $x(i-1 : i-\max(1,p-1) : -1, j)$ gets updated due to W followed by an update of $x(i : i+p, j+1)$ with respect to S. Applying this in the described column-wise fashion results in cache efficient chunk-wise updates of x as seen in line 20 to 26 of the Algorithm 3.5. Like in the forward substitution the values in the first column $x(i,1)$ need to be updated only with respect to W which is addressed in lines 29 to 33. Furthermore, it should be noted that the code avoids conditional statements and dependent iteration variables at the expense of a little extra work to help utilizing optimized vector operation units of the processor. This can be seen for instance in line 14 for $i = 1$ and $j = 2$, where the elements $r(0,2)$, $r(-1,2), \ldots$ (which wrap to $r(m,1)$, $r(m-1,1), \ldots$) are multiplied with 0 only entries in $W(1 : \max(1,p-1), 1, 1)$. Using the Hardware Performance Monitor (HPM) library [86] we could determine that the cache hit rate of our implementation on a POWER6 architecture (as described in Section 3.5) on a test problem is 99.945%.

3.4 MULTI-PRECISION ITERATIVE REFINEMENT

With the rise of *Floating Point Units* (FPUs) that perform single precision (32 bit) calculations faster than double precision (64 bit) calculations, the idea was born to use iterative refinement in order to achieve double precision accuracy while performing the most time-consuming computations in single precision.

The application of iterative refinement techniques for increasing the accuracy of computational calculations goes back to the work of Wilkinson et al. [24, 92, 141] and are therefore nearly as old as computers themselves. Since a matrix A is a linear operator, we can find a correction c for an approximate solution x_0 to the equation

Algorithmus 3.6 Iterative refinement to solve $Ax = b$.

1: $x_0 \leftarrow$ Initial guess
2: $r_0 = b - Ax_0$
3: **repeat**
4: $\quad Ac_i = r_i$
5: $\quad x_{i+1} = x_i + c_i$
6: $\quad r_{i+1} = b - Ax_{i+1}$
7: **until** $\|r_{i+1}\| \leq \epsilon$

system $Ax = b$ by solving $Ac = r = b - Ax_0$. A better approximation x_1 is then given by $x_1 = x_0 + c$. Repeating this procedure leads to the iterative refinement as outlined in Algorithm 3.6. It should be noted that this technique can also be motivated by considering the Newton's method with $f(x) = b - Ax$. The theory regarding convergence and stability of iterative refinement with respect to rounding errors and floating point formats is well established and the reader is kindly referred to [38, 68].

Traditionally, x86-based FPUs on CPUs use an 80 bit extended precision floating point format to compute all floating point calculations independent of the operands being single or double precision. With the introduction of short vector extensions like the Streaming SIMD* Extensions (SSE) on Intel CPUs, special FPUs take advantage of reduced precision and therefore allowing faster single precision computations [112]. Also highly parallel processing units like *Graphical Processing Units* (GPUs) utilize single precision FPUs that consume only a fourth of the space a double precision FPU would need. This consequently allows for more parallel units on the same die. Similar argumentation conveys to *Field Programmable Gate Arrays* (FPGAs) where the die real estate is the major limiting factor of parallel processing units. Another positive aspect is due to the fact that a single precision number is transferred twice as fast from the memory to the processing unit than double precision. Since on modern architectures the gap between computational speed and memory access is a major bottleneck, single precision reduces the negative consequences of memory access and at the same time allows for twice as many numbers in local cache.

We adapt this notion behind multi-precision techniques by performing all calculations of Algorithm 3.6 in double precision but the most time-consuming part in Line 4 in single precision. This is illustrated in Figure 14.

Multi-precision solvers based on this approach and beyond are discussed in [10, 27, 28, 61] and convergence analysis can be found in [83]. In [117], a thorough analysis of the question was conducted on how much the precision of the low precision format can be reduced in case

*Single Instruction Multiple Data, based on Flynn's taxonomy [47].

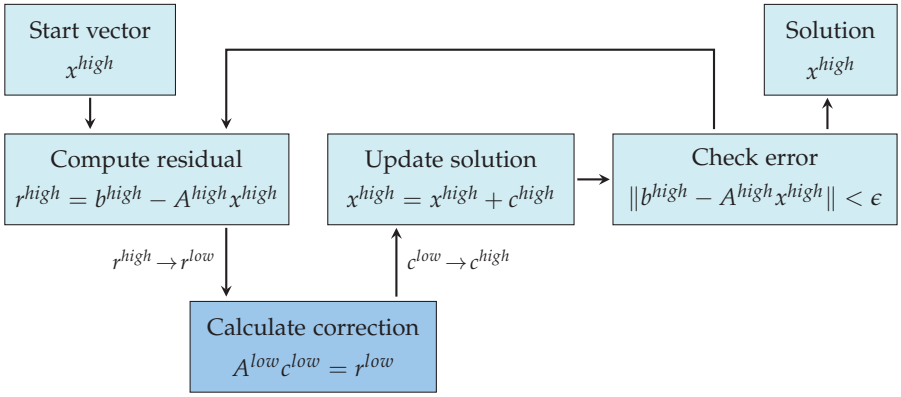


Figure 14: Multi-precision iterative refinement. In practice c^{high} is a double precision (64 bit) and c^{low} a single precision (32 bit) number. Typecasts between two formats are denoted by $\to$.

of a matrix stemming from an elliptic operator without introducing an error that is larger than the discretization error itself.

In MPIOM the algorithm depicted in Figure 14 was implemented so that an arbitrary solver preconditioner combination can be chosen to calculate an approximate correction c^{low}. Due to the fact that Fortran 90/95 has no template facilities as C++ does and *token concatenation* that allows to write macros that act like templates is not available on all preprocessors, the implementation was quite cumbersome. The solution was to separate the interface declaration and the definition of all functions belonging two a module into two files. In the definition file each function name as well as each floating point data type is a preprocessor macro. The declaration file sets the data type macro to single precision and the function name macro to the according function name with a suffix specifying the current precision. Then the definition file is textually included into the declaration file. After this, all macros are unset and this process is repeated for double precision. An additional interface block in the declaration file allows calling a function depending on the operands in a polymorphic manner. This approach is illustrated in Algorithm 3.7.

3.5 NUMERICAL EXPERIMENTS

Projections of future climate changes depend strongly on the accuracy of the simulated ocean circulation, therefore a realistic description of oceanic processes is required. A major factor which influences the accuracy is the resolution of the mesh.

For the simulations in IPCC AR4, that was published in 2007, a grid with a resolution of 256 grid points in longitude by 220 points in latitude was used. A limiting factor for increasing the resolution is the convergence of meridians at the North Pole, which is a source of numerical instabilities. For this reason, in the current calculations

Algorithmus 3.7 Declaration and definition file of the solver module including the double and single precision preconditioned CG.

```
MODULE solvers
    ! ...
    INTERFACE precond_cg_method
        MODULE PROCEDURE precond_cg_method_sp
        MODULE PROCEDURE precond_cg_method_dp
    END INTERFACE
    ! ...
    CONTAINS
#define PREC sp
#define PRECOND_CG_METHOD precond_cg_method_sp
! ...
#include "solvers_multi.f90"
#undef PREC
#undef PRECOND_CG_METHOD
! ...
#define PREC dp
#define PRECOND_CG_METHOD precond_cg_method_dp
! ...
#include "solvers_multi.f90"
! ...
END MODULE solvers
```

```
! solvers_multi.f90
FUNCTION PRECOND_CG_METHOD(A, b, x, ext_x, precond,\
    exchange, tol_opt, maxiter_opt) RESULT(kiter)
    REAL(PREC),   INTENT(IN)           :: b(:,:)
    REAL(PREC),   INTENT(INOUT)        :: x(:,:)
    TYPE(extent), INTENT(IN)           :: ext_x(:)
    REAL(PREC),   OPTIONAL, INTENT(IN) :: tol_opt
    INTEGER,      OPTIONAL, INTENT(IN) :: maxiter_opt
! ...
END FUNCTION PRECOND_CG_METHOD
```

for IPCC AR5, a new type of grid, i.e., the *tripolar* grid [98], has been introduced. Hereon, the North Pole is no longer a single point, but it has been expanded to a line with one pole at each of its endpoints. Therefore, we have two poles on the northern hemisphere and one on the southern hemisphere. The resolution for the IPCC AR5 runs is 0.4 deg (802×404 grid points). Nevertheless, basin-scale ocean models with a resolution of about 0.1 deg or higher are found to produce more realistic simulations. The main reason for such high resolution is the necessity for a proper representation of meso-scale ocean eddies which play a crucial role in determining the mean flow.

For the experiments presented in this work, we used a tripolar model with a resolution of 0.1 deg (3602×2394 grid points) which yields a system of about 8.6 million equations. To measure the actual runtime the function call to the solver of the barotropic system is surrounded by calls to `MPI_Barrier` and `MPI_Wtime`. The SOR solver uses a fixed number of 1200 iterations and reaches a relative residual with the order of magnitude 10^{-11}. As start value the null vector was chosen for all tests. The SOR relaxation parameter in all benchmarks was set to 1.934, which was hand tuned by a large number of test calculations. In order to compare our solvers with SOR, we set them to reach the same order of magnitude in the relative residual. In the following tables and figures SOR denotes the traditional red-black ordered SOR method in MPIOM, CG the new implemented conjugate gradient method with no preconditioning and Chebyshev the Chebyshev iteration with no preconditioning. The CG method and Chebyshev iteration were also analyzed with different preconditioners. In detail, Jacobi denotes the diagonal scaling preconditioner, ILU(0) the incomplete LU decomposition with zero fill-in according to [119, p. 303], SSOR the symmetric SOR method and (M)ICC(p) the (modified) incomplete Cholesky decomposition with p fill-in. Each solver was used to solve the barotropic subsystem in 30 consecutive time steps of the ocean model to get an average number of necessary iterations and an average runtime in seconds. This benchmark was repeated three times and averaged to compensate for effects like non-optimal process distribution in the cluster. All averages were calculated with the arithmetic mean. The setup time for the preconditioners and the SOR parameter was neglected. The partitioning is always given as $x \times y$ meaning x partitions in zonal and y partitions in meridional direction with one core per partition.

All benchmarks were performed on the new DKRZ high-performance supercomputer named Blizzard. The Blizzard cluster is an IBM p575 POWER6 system consisting of 264 nodes with 16 dual core CPUs per node, hence reaching a total of 8448 cores. Each core has a peak performance of 18.8 Gigaflop/s giving a total system peak performance of 158 Teraflop/s. The aggregate bandwidth of the InfiniBand Fat CLOS Tree interconnect is 7.6 Terabyte/s. A detailed description

of the IBM POWER6 microarchitecture can be found in Appendix B and [85]. For our benchmarks Simultaneous Multithreading (SMT) was disabled because experience gained at the DKRZ has shown that for symmetric memory access patterns, as they are found in many numerical simulations, SMT has at best no benefit. This concludes that on one node a total of 32 cores with 32 MPI processes were exclusively taken.

On the software side the IBM xlf Fortran compiler in version 13.1.0.7 running on IBM AIX 6.1.5.2 with IBM's Parallel Environment in version PE 5.2.2.2 (which includes POE/MPI) and disabled OpenMP support was used to compile and run MPIOM. The important compiler flags which were used are `O3` and `qhot`. These flags promise to enhance the performance of the code. The drawback is that the optimization is very aggressive which alters the results of MPIOM as tests have shown. For this reason the `qstric`, `qxflag=nvectver` (suppresses vector versioning) and `qxflag=nsmine` (suppresses strip mining) are needed in order to get correct results.

3.5.1 *Conjugate Gradient Method and Chebyshev Iteration*

In the tables and figures we will denote the unpreconditioned CG method with CG and the preconditioned CG method with only the name of the preconditioner for the sake of simplicity. Table 2 shows a comparison of SOR and the CG method with different preconditioners on 32×16 cores regarding the number of iterations and runtime. The CG method with no preconditioning needs more than 50% more iterations than SOR and since each CG iteration is more costly than a SOR iteration, the CG method performs much worse than SOR. We also see that with the help of a preconditioner the number of iterations can be drastically decreased. Since the communication overhead is a major limiting factor, this results in a shorter overall runtime compared to SOR if the overhead of preconditioning is overcompensated. One should note that one iteration of CG needs three communications: one for the matrix-vector multiplication, i.e., the stencil operation, and two for calculating dot products. Consequently the number of iterations in CG needs to be significantly lower than 400 iterations to make up for the single communication that SOR needs per iteration. The decisive factor in the communication is the latency for which reason the two-halo-boundary exchange of SOR is almost double as fast as two consecutively performed one-halo-boundary exchanges. Regarding the ICC preconditioner, we see that with an increase of fill-in the number of iterations decreases. Since this also increases the cost of the preconditioner, various test calculations are typically needed to figure out an optimal fill-in parameter with minimal runtime. For this setup on 32×16 cores, we can conclude that only the CG method with the modified ICC(4) preconditioner is slightly better than SOR.

method	iterations	runtime [s]	speedup
SOR	1200.0	0.1895	1.00
CG	1844.8	0.4583	0.41
Jacobi	1035.1	0.2743	0.69
ILU(0)	371.0	0.3418	0.55
SSOR	269.9	0.2629	0.72
ICC(4)	147.4	0.1949	0.97
ICC(6)	133.8	0.2028	0.93
ICC(8)	128.6	0.2257	0.84
MICC(4)	142.6	0.1885	1.01
MICC(6)	132.2	0.2004	0.95
MICC(8)	128.1	0.2241	0.85

Table 2: Number of iterations and runtime in seconds for SOR and CG method with various preconditioners on $\mathbf{32 \times 16}$ partitions and cores.

We also see that the modified ICC preconditioner needs less iterations than the standard ICC preconditioner for the same fill-in parameter.

Table 3 shows the same comparison for 32×32 cores. Here, we can see that with the help of the modified ICC preconditioner the CG method is up to 24% faster than the SOR method. This is due to the fact that CG with MICC(4) scales much better (speedup 1.70) compared to SOR (speedup 1.38) from 32×16 to 32×32 cores. It can also be seen that the number of iterations increase in case of preconditioning since we apply our preconditioners in a block-Jacobi way.

In Table 4, the largest setup with 64×32 cores is shown. Doubling the number of cores again has led to a speedup of 1.1 for SOR while CG with modified ICC(4) sees a speedup of 1.49. It should be noted that we are dealing with *strong scaling*, i.e., we increase the number of cores with a fixed problem size. On this setup, CG with MICC(4) is about 67% faster than SOR. Figures 15 and 16 provide a survey of the change of runtime and the number of iterations over all setups. Table 5 shows the scalability by comparing the speedup of setups with 32×32 and 64×32 cores to the setup with 32×16 cores. In case of perfect scaling the speedup of the 32×32 setup should be 2 and for the setup with 64×32 cores 4 which is hard to achieve for strong scaling. This is due to the fact that with an increase in processing units the number of operations per processor decreases but the necessary communication increases. Therefore, the communication becomes a larger factor in the overall runtime. We see that the CG method with an appropriate preconditioner scales better than the traditional SOR method. Peculiarly, on the 64×32 setup the CG method with Jacobi

method	iterations	runtime [s]	speedup
SOR	1200.0	0.1374	1.00
CG	1844.9	0.3399	0.40
Jacobi	1035.1	0.2003	0.69
ILU(0)	374.8	0.2212	0.62
SSOR	276.6	0.1656	0.83
ICC(4)	156.2	0.1146	1.20
ICC(6)	141.5	0.1176	1.17
ICC(8)	136.2	0.1460	0.94
MICC(4)	151.4	0.1111	1.24
MICC(6)	139.9	0.1156	1.19
MICC(8)	135.5	0.1271	1.08

Table 3: Number of iterations and runtime in seconds for SOR and CG method with various preconditioners on $\mathbf{32} \times \mathbf{32}$ partitions and cores.

preconditioner performs even worse than on 32×32 cores which could be caused by a non-optimal process distribution in the cluster during the test runs but needs to be further examined.

Based on the results for the CG method, we tested the Chebyshev iteration with no preconditioning and with modified ICC(4) preconditioning. The results of these benchmarks are shown in Table 6. Compared to the CG method, the Chebyshev iteration needs more iterations but benefits from the absence of dot products which include global sums. The costs of a boundary exchange and a global sum are shown in Table 7. From this it can be drawn that a global sum operation becomes more expensive with an increase in the number of cores while the cost of a boundary exchange stays roughly the same. This is due to the fact that the MPI global sum operation is often implemented as a tree structure of partial sums with parallel work in $O(\log n)$ while a neighbor to neighbor boundary exchange is always $O(1)$. As a last point, we can conclude that the Chebyshev iteration with modified ICC(4) preconditioner is 95% faster than the traditional SOR solver on the largest setup.

3.5.2 *Additive Schwarz Method*

Our next benchmark is the additive Schwarz method with the CG method as local solver. According to our former results, we decided to use the modified ICC preconditioner with fill-in 4 to accelerate the CG method. By a set of test runs we determined that setting the local solver to reach a relative residual of 10^{-1} within a maximum number of 20 iterations is a proper choice for our problem. With

method	iterations	runtime [s]	speedup
SOR	1200.0	0.1247	1.00
CG	1844.8	0.3247	0.38
Jacobi	1035.1	0.2401	0.52
ILU(0)	379.8	0.1356	0.92
SSOR	297.5	0.1099	1.13
ICC(4)	165.6	0.0769	1.62
ICC(6)	151.3	0.0939	1.33
ICC(8)	146.0	0.0840	1.48
MICC(4)	160.9	0.0747	1.67
MICC(6)	149.9	0.0924	1.35
MICC(8)	145.6	0.0836	1.49

Table 4: Number of iterations and runtime in seconds for SOR and CG method with various preconditioners on $\mathbf{64} \times \mathbf{32}$ partitions and cores.

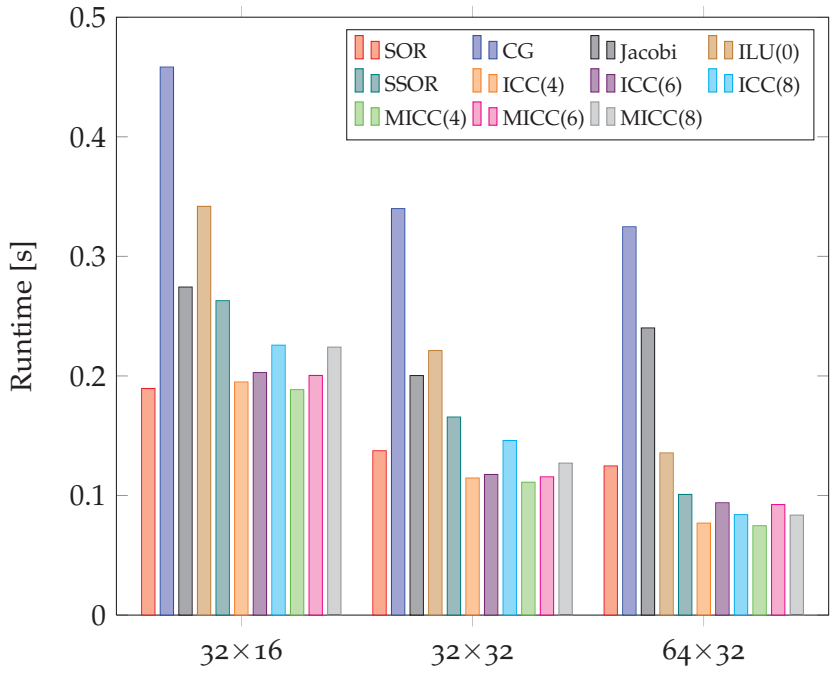


Figure 15: Runtime comparison of SOR with CG and various preconditioners on different number of cores.

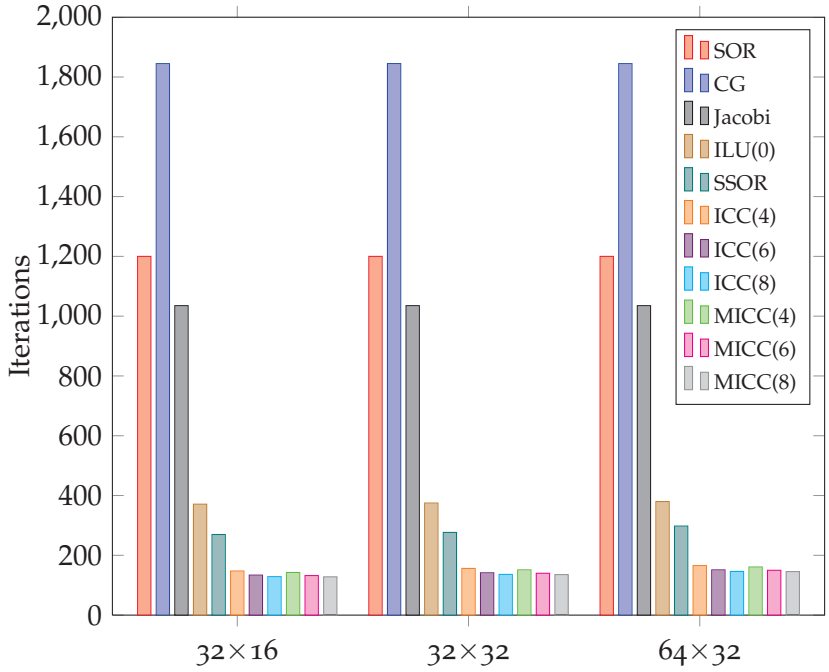


Figure 16: Number of iterations of SOR and CG with various preconditioners on different number of cores.

method	32×32	64×32
SOR	1.38	1.49
CG	1.35	1.41
Jacobi	1.37	1.14
ILU(0)	1.55	2.52
SSOR	1.59	2.39
ICC(4)	1.70	2.53
ICC(6)	1.72	2.16
ICC(8)	1.55	2.69
MICC(4)	1.70	2.52
MICC(6)	1.73	2.17
MICC(8)	1.76	2.68

Table 5: The gained speedup of the runtimes on 32×32 cores (Table 3) and 64×32 (Table 4) compared to 32×16 cores (Table 2).

	32×16	32×32	64×32
SOR iteration	1200	1200	1200
SOR runtime	0.1895	0.1374	0.1247
Chebyshev iterations	3763.9	3764.7	3764.3
Chebyshev runtime [s]	0.6955	0.4941	0.4516
Chebyshev speedup	0.27	0.28	0.28
MICC(4) iterations	164.0	172.0	183.8
MICC(4) runtime [s]	0.1931	0.1054	0.0639
MICC(4) speedup	0.98	1.30	1.95

Table 6: Number of iterations and runtime in seconds of the Chebyshev iteration with no preconditioner and MICC(4) preconditioning on different number of cores. For comparison the number of iterations and the runtime of SOR as well as the speedup compared to SOR is provided.

	32×16	32×32	64×32
Boundary Exchange [s]	$3.19 \cdot 10^{-5}$	$2.90 \cdot 10^{-5}$	$3.09 \cdot 10^{-5}$
Global Sum [s]	$2.82 \cdot 10^{-5}$	$3.37 \cdot 10^{-5}$	$4.16 \cdot 10^{-5}$

Table 7: Measured time in seconds for one boundary exchange and global sum on various numbers of partitions determined by 100 000 performed operations.

	halos	32×16	32×32	64×32
iterations	8	93.8	93.6	107.9
	16	55.9	64.4	61.6
	24	44.7	51.3	53.3
runtime [s]	8	0.8291	0.4498	0.2955
	16	0.6724	0.4351	0.2744
	24	0.6651	0.4492	0.3065
SOR	2	0.1895	0.1374	0.1247

Table 8: Number of iterations and the runtime of the Schwarz method with varying number of overlapping halos and on different number of cores. As local solver the CG method with modified ICC(4) preconditioner was used and set to reach a relative residual of 10^{-1} with 20 being the maximum number of iterations. For comparison the SOR runtime is provided. The iteration number of SOR is 1200 for all setups.

these parameters we run our benchmark with a varying number of overlapping halos, i.e., 8, 16, 24, and on different number of cores. This means that the gray area in Figure 10 consists of 8, 16 or 24 halos which overlap the area of the neighboring partitions. Table 8 shows our results.

The first observation is that the overall runtime compared to the CG method or the Chebyshev iteration is much higher, albeit the number of iterations of the Schwarz method is generally lower than in the case of the CG method or Chebyshev iteration. This is due to several facts. First, the communication of only one halo as in the CG method is less expensive than communicating up to 24 halos. Additionally, starting with 2 halos, it is necessary to communicate with 8 neighbors instead of 2 which can be seen in Figure 10. Second, in the case of 64×32 cores, partitions can be as small as $56 \times 74 = 4144$ grid points. With 24 additional halos this number increases to 12688 which is more than 3 times as much. This leads to more computation per partition compared to the CG method on the same setup. Looking at the scaling behavior, we see that the speedup when doubling the number of cores decreases with an increase in halos. This can also be explained with the fact that the ratio of the size of a partition with additional halos and of the original partition becomes higher with an increase in halos and cores.

3.5.3 *Multi-Precision Iterative Refinement*

The benchmarks for our multi-precision iterative refinement solver in MPIOM were performed on an Intel Core i7 920 with 4 cores, 2.67

	MIR+CG	CG	Ratio
CG Iterations	480.3	470.5	1.02
Runtime [s]	0.2488	0.3759	0.66

	MIR+PCG	PCG	Ratio
PCG Iterations	49.9	43.1	1.16
Runtime [s]	0.2673	0.2269	1.18

Table 9: Runtime in seconds and the number of necessary iterations needed to solve the barotropic subsystem by the multi-precision iterative refinement (MIR) with nested CG method and the CG method. The second table shows the analogue for the modified ICC(7) preconditioned CG method. The MIR needed in all cases about 6 outer iterations while the inner CG solver was set to stop if the current relative residual is less than 10^{-2}.

GHz, SSE 4.2, 8 MB cache and 6 GB RAM desktop system. This is due to the fact that the Intel SSE extensions are far more advanced than the vector multimedia extension (VMX) of the POWER6 architecture [43] and are more likely to be exploited by a modern compiler which is in our case the Intel Fortran compiler 12.0.0. To allow for MPIOM to be run on a desktop system we used the tripolar model TP10L40 with a resolution of 1.9 deg (392×162 grid points) and 40 vertical levels. This setup leads to an approximate RAM usage of 2 GB for MPIOM. The same 30 time steps benchmark as described above was performed on a single core to not further reduce the workload per core. The important compiler flags which were used are `O3` and `xsse4.2`. Table 9 shows the runtime and iterations numbers for solving the barotropic subsystem comparing the multi-precision iterative refinement (MIR) with CG and the pure CG method.

The speedup of the mixed precision approach is about 1.5 compared the pure double precision CG method and significantly lower than the optimal to be expected speedup of 2. This is due to the fact that the vectors we are dealing with in the CG method have a length of $392 \cdot 162 = 63504$ which results in less then 0.5 MB memory for double precision. Since the number of vectors in the CG method and the stencil operations are less than 16, the 8 MB cache holds almost all operands and the speedup for single precision is not to be gained by the doubled transfer rate from RAM to the CPU. Consequently, our benchmark is inherently compute-bound since a possibly doubled transfer rate from the cache to the CPU registers is not substantial as our results show. The processing of single precision numbers is only twice as fast as double precision if the SSE units of the CPU are used. Separate time measurements for the operations of the CG methods yielded that all vector valued operations are performed twice as fast

in single precision. For the matrix-vector multiplication a speedup of 1.73 and for the dot product a speedup of 1.55 was measured. This indicates that the compiler was not able to fully exploit SSE. The overall speedup is further reduced by the overhead of the iterative refinement, i.e., type conversions, additional matrix-vector and vector operations and the boundary exchange function which converts the single precision operands to double precision before performing the actual exchange in double precision. It was decided not to adapt this function to single precision due to its complexity and deep integration into MPIOM. In the future, the boundary exchange function will be completely replaced by the UniTrans* library which allows for direct usage of single precision floating-point numbers.

The second part of Table 9 shows the benchmark results using a preconditioned CG with a modified ICC preconditioner with fill-in 7. We chose a fill-in of 7 to have partial dot products of length 8 in line 14 of Algorithm 3.5 which should lead to two SSE operations. In this case the MIR performs worse than the pure CG method because the preconditioning operation did not benefit from single precision arithmetic. Since the preconditioning step was measured to cause 81% of the runtime per CG iteration in double precision and even more in single precision, it has the strongest impact on the overall runtime. Several efforts were undertaken to allow the compiler to use SSE for the vector and dot product operations in Algorithm 3.5. For this purpose, one important aspect is the data structure alignment. The XMM registers which are needed for SSE operations need a 128 bit alignment, i.e., four 32 bit single precision numbers need to be consecutively stored in memory with the address of the first number divisible by 128. Since the Fortran 90/95 standard provides no facilities to allocate aligned memory, Intel specific code annotations, i.e., `!DEC$ ATTRIBUTES ALIGN: 16` were used for the operands. Additionally, higher dimensional arrays were increased in the first dimension to assure 128 bit alignment which is called data structure padding. The fill-in parameter p was declared constant as well as several notations for the vector operations were tested. With all our attempts the compiler could not exploit SSE in the preconditioning step albeit it is theoretically possible. The only feasible recourse seems to be a C implementation which can directly apply SSE commands with the help of the `mmintrin.h` header file. This approach was not pursued due to the complexity of interfacing Fortran with C code.

3.6 SUMMARY AND CONCLUSION

As one of the goals in the ScalES project, we implemented a solver component called *ScalES-Lib* for stencil-based symmetric positive definite

*The **uni**versal data **trans**position (UniTrans) library was developed as part of the ScalES project. More information can be found in Appendix A.

systems. Therein, we provide different iterative methods combined with different preconditioners which allow for shifting the workload from the interconnect to the CPUs by reducing the number of iterations. Currently we provide the CG method and the Chebyshev iteration which, in contrast to CG, uses no inner products that cause communication and is therefore especially suited for high scalability. Each solver can be combined with different preconditioners ranging from Jacobi, ILU(0) and SSOR through to the more sophisticated ICC(p) and modified ICC(p) preconditioners. Additionally, these solvers and preconditioners can be applied inside an additive Schwarz method to further reduce the necessary number of iterations and consequently communication. A multi-precision iterative refinement was also implemented to allow the application of all solvers and preconditioners in single precision to reduce the runtime while still achieving double precision accuracy. Depending on the given problem size and employed hardware, the user can choose an appropriate solver/preconditioner combination with the help of the ScalES-Lib. By virtue of our focus on stencil-based systems it was possible to obtain a highly cache-optimized implementation, especially for our ICC(p) preconditioner.

We saw that the CG method and Chebyshev iteration combined with a preconditioner like the (modified) ICC(p) can almost halve the runtime when solving the barotropic subsystem on a large setup with 2048 cores. This is a major improvement which was only possible by the usage of an appropriate preconditioner. Consequently, we saw that parallel preconditioners facilitate the efficient usage of iterative methods in large scale computing.

Although the actual runtime to solve the barotropic subsystem is already a fraction of a second, there is still much need for further improvement. Considering the fact that one simulated scenario for the IPCC AR5 spans a simulation time of 100 years, resulting in millions of time steps where each has a barotropic subsystem to solve, one can easily see the leverage effect we are dealing with. The bright side of solving one linear system with varying right-hand-sides many times is that the setup time of a preconditioner or solver can be neglected.

In the future more complex models with higher accuracies will be needed to better supplement the research in climatology. This development will go hand in hand with a continuously raising demand for better mathematical methods, solvers and preconditioners on high-performance hardware to make more complex simulations feasible. With the help of ScalES-Lib one big leap was accomplished in improving the scalability of a valuable legacy ocean model whose grid resolution can now be further increased. This also allows climate scientists to efficiently use MPIOM probably even on larger hardware than the currently employed Blizzard cluster.

In the next chapter, we will deal with a more recent and rather unknown mathematical theory with respect to preconditioners. Our objective is to better judge the quality of a preconditioner in terms of the reduction in the number of iterations as well as its parallel properties. This will be important in the future of HPC where compute clusters will have a drastically increased number of parallel processing units and a more complex interconnect hierarchy.

4

SUPPORT THEORY BASED PRECONDITIONERS

In this chapter we will elaborate on the subject of support theory and its application to preconditioning. The foundation for this is laid in the first two sections by a small introduction to graph theory which is a necessary requirement to comprehend support theory. Section 4.3 then provides us with the main definition and theorems from support theory which we will use in Section 4.4 to analyze Steiner graph preconditioners. Section 4.5 will provide us with the necessary tools from the domain of network flow problems for the author's main contribution to the field of support theory: the estimation of the condition number of a system that is preconditioned with a block-Jacobi Steiner tree preconditioner. Based on this result, a model for a hardware-aware preconditioner is proposed in Section 4.6.

4.1 INTRODUCTION TO GRAPH THEORY

This section provides a brief overview of the most basic definitions, notations and terminology of graph theory which were taken mostly from [22, 39]. The first two sections will serve us as a basis for establishing the support theory.

4.1.1 *Basics and Notation*

For a given set A we call a set $\mathcal{A} = \{A_1, \ldots, A_n\}$ of pairwise disjoint and nonempty subsets $A_i \subseteq A$ a *partition* of magnitude $|\mathcal{A}| := n$ if $A = \bigcup_{i=1}^{n} A_i$. Another partition $\mathcal{A}' = \{A'_1, \ldots, A'_m\}$ *refines* the partition $\mathcal{A}$ if each A'_i is a proper subset of an A_j. Since we will be interested in 2-element subsets instead of ordered tuples we define in the spirit of the Cartesian product $\times$ an unordered variant as following $A \otimes B := \{\{x, y\} \mid x \in A, y \in B\}$.

$\mathcal{A}$

$|\mathcal{A}|$

$A \otimes B$

An *undirected, weighted graph* $G = (V, E, \omega)$ consists of a set of vertices V and a set of edges $E \subseteq V \otimes V$ as well as a *weighting function* $\omega : e \in E \to \mathbb{R}^+$. If $\omega(e) = 1$ for all $e \in E$ then G is *unweighted*. An edge $\{v_i, v_i\}$ is called a *loop* on the vertex v_i. We call an unweighted, undirected graph with no loops a *simple graph*. In contrast to an undirected graph with edges in $V \otimes V$, a *directed graph* has edges in $V \times V$ which subsequently exhibit a direction.

$\omega(e)$

To simplify the notation we define the *capacity* $c : V \otimes V \to \mathbb{R}_0^+$,

$c(e)$

$$c(e) := \begin{cases} \omega(e) & e \in E \\ 0 & e \notin E \end{cases},$$

as an extension of ω onto $V \otimes V$. If not stated otherwise we use just graph to refer to an undirected, weighted graph with a finite vertex set V. The vertex set of a graph G is denoted by $V(G)$ and resp. the edge set by $E(G)$. The number of vertices will be denoted by $|G|$.

$V(G)$ $E(G)$ $|G|$

Given two graphs $G = (V, E, \omega)$ and $G' = (V', E', \omega')$, if $V' \subseteq V$ and $E' \subseteq E$, then G' is *subgraph* of G (and G a *supergraph* of G') which we denote by $G' \subseteq G$. If $G' \subseteq G$ and every edge $e = \{u, v\} \in E$ with $u, v \in V'$ is also an edge in G', i.e., $e \in E'$, that satisfies $\omega(e) = \omega'(e)$, we call G' an *induced subgraph* of G. We say that the vertex set V' *induces* G' in G and write $G[V'] := G'$.

$G[V']$

We say a vertex v is *incident* with an edge e if $v \in e$. Two vertices u, v are *adjacent* and therefore *neighbors* if there exists an edge $e = \{u, v\}$. A *complete graph* is a graph where all vertices are adjacent. The *volume* $\mathrm{vol} : v \in V \to \mathbb{R}_0^+$ of a vertex v is the sum of the capacity of all edges incident to v, i.e.,

$\mathrm{vol}(v)$

$$\mathrm{vol}(v) := \sum_{e \in v \otimes V} c(e).$$

The *degree* of v, denoted by $\deg(v)$, is the number of incident edges to v. We extend the definition of the volume of a vertex to a set of vertices A by

$\deg(v)$ $\mathrm{vol}(A)$

$$\mathrm{vol}(A) := \sum_{v \in A} \mathrm{vol}(v)$$

as well as the capacity of an edge to a set of edges $E \subseteq X \otimes Y$ with $X, Y \subseteq V$ by

$\mathrm{cap}(X, Y)$

$$\mathrm{cap}(X, Y) := \sum_{e \in X \otimes Y} c(e).$$

Furthermore, we define $\mathrm{out}(X)$ as a shorthand for $\mathrm{cap}(X, V \setminus X)$ representing the *outflow capacity* of the subgraph $G[X]$ in G.

$\mathrm{out}(X)$

If $X \subseteq V$ is a set of vertices, the *contraction* of the vertices X in $G = (V, E, \omega)$ results in a new graph $G/_X = (\bar{V}, \bar{E}, \bar{\omega})$ with a new node $\bar{x} \notin V$, vertices $\bar{V} = V \setminus X \cup \{\bar{x}\}$, edges $\bar{E} = E(G[V \setminus X]) \cup \{\{v, \bar{x}\} \mid \{v\} \otimes X \in E\}$ and weight function $\bar{\omega}$ defined for an edge $\{u, v\} \in \bar{E}$ as

$G/_X$

$$\bar{\omega}(\{u, v\}) = \begin{cases} \omega(\{u, v\}) & u, v \in V \setminus X \\ \mathrm{cap}(X, \{v\}) & u = \bar{x}, v \in V \setminus X \\ \mathrm{cap}(\{u\}, X) & u \in V \setminus X, v = \bar{x} \end{cases}.$$

Given a partition $\mathcal{A} = \{A_1, \ldots, A_n\}$, we will denote by $G/_{\mathcal{A}}$ the graph with n vertices that is obtained by consecutively contracting each $A_i \in \mathcal{A}$.

$G/_{\mathcal{A}}$

4.1.2 *Paths and Embeddings*

A *walk* is a sequence of edges $(e_1, \ldots, e_k)$ with $e_i = \{v_i, v_{i+1}\} \in E$ that starts from vertex v_1 and ends in vertex v_k. If all vertices of a walk are distinct it is a *path*. The *length* of a path is the number of its edges. If the first and last vertex of a path are the same, i.e., $v_1 = v_k$, the path is called a *cycle*.

A graph is *connected* if for any two vertices v_i, v_j there exists a path from v_i to v_j. Given a connected graph H and a graph $G \subseteq H$, an *embedding* is an injective mapping of the vertices $V(G)$ onto $V(H)$ and edges $E(G)$ onto paths in H. With $\text{path}(e)$ we denote the path in H the edge $e \in G$ was mapped to. The *dilation* of the edge e is the length of $\text{path}(e)$ while the *congestion* of e is defined as the sum of the weight of all edges whose paths include e divided by the weight of e, formally

$\text{path}(e)$

$$\frac{1}{\omega(e)} \sum_{f : e \in \text{path}(f)} \omega(f).$$

In case of an unweighted graph this is just the number of paths which include e.

4.1.3 *Forests, Trees and Stars*

If a graph G contains no cycles, i.e., *acyclic*, it is called a *forest*. A connected forest is called a *tree*. By definition each vertex in a tree is linked to any other vertex by one unique path. A vertex with degree 1 in a tree is called a *leaf* whereas a vertex with degree larger than 1 is labeled *internal*. In order to induce a partial ordering on $V(T)$ of a tree T, we define one vertex as the *root* r of T and write $u \leq v$ for $u, v \in V(T)$ if $u \in e$ for an edge e in the unique path from r to v. Based on this partial ordering we define the *height* or *level of* a vertex u as the length of the path from r to u and denote this by $\text{lev}(u)$. The set of all vertices with height k is called the k-th *level* of T. The height of T is defined as the maximum height over its vertices and we write $\text{height}(T)$. If all leaves of a tree are at the same level, the tree is *balanced*.

$\text{lev}(u)$

$\text{height}(T)$

A *star graph* or just *star* S is a tree that has only one vertex r with $\deg(r) \geq 2$ which is therefore considered the root of S.

Given a connected graph $G = (V, E, \omega)$, we define a *spanning tree* $T = (V', E', \omega')$ in G such that $V' = V$, $E' \subseteq E$ and $\omega'(e) = \omega(e)$ for all $e \in E'$. The spanning tree of G with maximal sum of the weights of its edges is called *maximum-weight spanning tree*.

Another special kind of a spanning tree, called *low-stretch spanning tree,* was defined in [5]. The *stretch* of an edge $e = \{u, v\}$ in a graph G with respect to a spanning tree T is given by

$$\mathrm{st}_T(e) = \omega(e) \sum_{i=1}^{k} 1/\omega(e_i),$$

where $(e_1, \ldots, e_k)$ is the unique path from u to v. The average stretch of whole G is then defined by

$$\mathrm{st}_T(G) = \frac{1}{|E|} \sum_{e \in E} \mathrm{st}_T(e).$$

A low-stretch spanning tree is the spanning tree T which minimizes $\mathrm{st}_T(G)$ or, more relaxed, T for which $\mathrm{st}_T(G)$ is reasonably small. Low-stretch spanning trees have a large field of applications [44] like preconditioners [127] amongst others.

4.1.4 *Cuts*

A set of edges $C \subset E$ in a graph $G = (V, E, \omega)$ is a k-way *edge cut* if there exists a partition $\mathcal{A} = \{A_1, \ldots, A_k\}$ of V so that $C = \{e \in E \mid e \in A_i \otimes A_j, i \neq j, \}$ with $i, j \in \{1, \ldots, k\}$. A subgraph $G[A_i]$ of G is called a *cluster* in the *graph decomposition* defined by the edge cut and its associated partition.

In case of a 2-way edge cut C with associated partition $\{X, V \setminus X\}$ we define the *sparsity* of C as

$$\phi(X) = \frac{\mathrm{cap}(X, V \setminus X)}{\min\{\mathrm{vol}(x), \mathrm{vol}(V \setminus X)\}}. \tag{4.1}$$

A *sparsest cut* is an edge cut with minimum sparsity over all possible cuts and a *minimum cut* is an edge cut that minimizes $\mathrm{cap}(X, V \setminus X)$.

4.2 LAPLACIANS & SDD MATRICES

In this section we will establish the connection between graphs and matrices. We start with some basic definitions.

Definition 11 (Laplacian)**.** Given a simple, unweighted graph $G = (V, E)$ with $|V| = n$ and an enumeration v_i, $i = 1, \ldots n$ of the elements of V, the *Laplacian* matrix $L \in \mathbb{R}^{n \times n}$ of G is defined as

$$l_{ij} = \begin{cases} \deg(v_i), & i = j \\ -1, & i \neq j, \{v_i, v_j\} \in E \\ 0, & \text{otherwise} \end{cases}$$

for $i, j = 1, \ldots, n$.

This definition provides an isomorphism between simple graphs and Laplacians and can be canonically generalized to weighted, undirected graphs with loops [59].

Definition 12 (Generalized Laplacian matrix)**.** A matrix $L \in \mathbb{R}^{n \times n}$ is a *Generalized Laplacian matrix* if and only if

- L is symmetric,
- $l_{ii} > 0$ for $i = 1, \ldots, n$,
- $l_{ij} \leq 0$ for $i \neq j$, $i, j = 1, \ldots, n$,
- $l_{ii} \geq \sum_{i,j:i \neq j} |l_{ij}|$ for $i = 1, \ldots, n$.

A matrix fulfilling the last requirement, i.e., $l_{ii} \geq \sum_{i,j:i \neq j} |l_{ij}|$, is typically called *diagonal dominant* in this context albeit this conflicts with the more common definition of diagonal dominance, i.e., $|l_{ii}| \geq \sum_{i,j:i \neq j} |l_{ij}|$. Having pointed out this inconsistency, we will use the term diagonal dominance to denote matrices with $l_{ii} > 0$ and $l_{ii} \geq \sum_{i,j:i \neq j} |l_{ij}|$ for $i = 1, \ldots, n$.

We achieve positive definiteness for a Laplacian L, if L is irreducible and $l_{ii} > \sum_{i \neq j} |l_{ij}|$ for at least one row i. In this case we have a diagonal dominant Stieltjes Matrix. The mapping from a generalized Laplacian L to an undirected, weighted graph G_L is given by following rules: G_L

- each row/column i corresponds to a vertex i,
- each off-diagonal entry $l_{ij} \neq 0$ with $i \neq j$ corresponds to an edge from vertex i to j with weight $|l_{ij}|$,
- each diagonal entry l_{ii} is the sum of all incident vertices' weights and $d \geq 0$, i.e., $l_{ii} = \sum_{i \neq j} |l_{ij}| + d$. If $d > 0$ the vertex v_i has a loop with weight d.

Remark 13. While in [55] entries $l_{ii} = \sum_{i \neq j} |l_{ij}| + d$ with $d > 0$ are interpreted as loops, it is also possible to see a loop as an edge to an *implicit zero-valued boundary vertex* [59]. In this case, the Laplacian $L \in \mathbb{R}^{(n+1) \times (n+1)}$ has an additional row/column $n + 1$ with corresponding entries $-d_i \leq 0$ and variable $x_{n+1} = 0$. We also note that L clearly has zero row sum which we can now always assume if necessary, due to this simple transformation.

For a given graph G this mapping can be uniquely reversed to gain a generalized Laplacian L_G. Using this equivalence, we define the addition $A = G + H$ and subtraction $B = G - H$ on two graphs G and H in the way that $L_A = L_G + L_H$ and $L_B = L_G - L_H$, whereas the subtraction is only defined if L_B is a generalized Laplacian. Furthermore, we define a scalar multiplication $A = \alpha G$ with $\alpha \geq 0$ by letting $L_A = \alpha L_G$. L_G $G + H$ $G - H$ αG

The resulting graph G' after the *elimination* of a vertex v_i in the graph G is defined such that $L_{G'}$ is the resulting Laplacian after elimination of the variable x_i in L_G, i.e., performing symmetric Gaussian elimination on row/column i.

Besides the structural aspect of this mapping, also other properties like the positive semidefiniteness convey from a generalized Laplacian L with zero row sum to its corresponding graph G_L. Since $\omega(\{v_i, v_j\}) = -l_{ij}$, we can easily show for all x that

$$x^T L x = \sum_{\{v_i,v_j\}\in E(G)} \omega(\{v_i, v_j\})(x_i - x_j)^2. \tag{4.2}$$

The theory for Laplacians that will be established in the following sections also applies to a wider family of matrices. In Definition 7.2 of Gremban et al. [56, p. 110] is shown how a symmetric, diagonal dominant (sdd) matrix $A \in \mathbb{R}^{n\times n}$ can be transformed into a generalized Laplacian. First, we note that $A = D + A^+ + A^-$, where D is the diagonal part, A^+ the positive off-diagonal elements and A^- the negative off-diagonal elements of A. We now define an extended $A_{\text{ext}} \in \mathbb{R}^{2n\times 2n}$ as

sdd

$$A_{\text{ext}} = \begin{pmatrix} D + A^- & -A^+ \\ -A^+ & D + A^- \end{pmatrix}$$

which is obviously a Laplacian. It can now easily be seen that:

Lemma 14. *Let A be a symmetric, diagonal dominant matrix. We have for all x that*

$$Ax = b \Leftrightarrow A_{ext} \begin{pmatrix} x \\ -x \end{pmatrix} = \begin{pmatrix} b \\ -b \end{pmatrix}.$$

A complete proof can be found in Gremban [55, p. 111]. On the basis of Gremban's lemma, the graph corresponding to A_{ext} is referred to as *Gremban cover* [126].

As a result of this lemma, we can apply the graph theoretical techniques of the following sections to sdd matrices. To the best knowledge of the author, no extension to all symmetric, positive definite (spd) matrices is known. For positive semidefinite matrices we will use the common appreviation psd and for the symmetric case spsd.

spd
psd
spsd

4.3 SUPPORT THEORY

The theory presented in this section goes back to an unpublished manuscript of Vaidya [137], which he presented in a scientific meeting. Therein, he described novel combinatorial techniques to derive preconditioners that have a certain quality in terms of bounding the condition number of the preconditioned system. His work consists of techniques that bridge the gap between spectral analysis and graph

theory, whose tools largely enrich the possibilities to analyze and construct preconditioners. Since Vaidya decided not to publish his work but rather founded the company COMPUTATIONAL APPLICATIONS AND SYSTEM INTEGRATION* instead to market his preconditioners, his initial work was carried on by other researchers.

The theory of Vaidya's preconditioners with all missing proofs was first rigorously written down in [17]. Some results were presented before in the PhD thesis of Gremban [55], wherein he presented some extensions which lead to a new class of preconditioners. Implementation and an experimental study of Vaidya's preconditioners was conducted in [33] and of Gremban's new class of preconditioners in [56].

Another key contribution based on Vaidya's techniques was made by Spielman and Teng who presented in [126] a linear solver that, given an $n \times n$ sdd matrix with m non-zero entries and a right-hand-side b, finds an ϵ-approximate solution $\tilde{x}$ for $Ax = b$ in time $O(m^{1.31} \log(n/\epsilon))$. Their solver is based on a preconditioned Conjugate Gradient or Chebyshev method and a preconditioner B that bounds the condition number of $B^{-1}A$ and therefore the number of iterations.

In [20], the flourishing theory of Vaidya was restated and extended from a more algebraic point of view. We will follow their presentation in this section and omit the proofs.

4.3.1 *Support Number and Basic Properties*

Definition 15 (Generalized eigenvalue)**.** The number λ is a *generalized eigenvalue* of the matrix pencil $(A, B) := A - \lambda B$ if there exists a vector $x \neq 0$ such that $Ax = \lambda Bx$ and $x \neq 0$. The largest generalized eigenvalue is denoted by $\lambda_{\max}(A, B)$ and the smallest by $\lambda_{\min}(A, B)$.

This definition yields a generalization of the condition number.

Definition 16 (Generalized condition number)**.** The *generalized condition number* of two spd matrices $A, B \in \mathbb{R}^{n \times n}$ is defined by $\kappa(A, B)$

$$\kappa(A, B) := \kappa(B^{-\frac{1}{2}} A B^{-\frac{1}{2}}) = \frac{\lambda_{max}(B^{-\frac{1}{2}} A B^{-\frac{1}{2}})}{\lambda_{min}(B^{-\frac{1}{2}} A B^{-\frac{1}{2}})} = \frac{\lambda_{max}(A, B)}{\lambda_{min}(A, B)}.$$

The fundamental part of support theory is the reinterpretation of the eigenvalues of a matrix pencil.

Definition 17 (Support number)**.** The *support number* $\sigma(A, B)$ of a matrix pencil (A, B) where $A, B \in \mathbb{R}^{n \times n}$ is defined by $\sigma(A, B)$

$$\sigma(A, B) = \min \left\{ t \in \mathbb{R} \mid x^T(\tau B - A)x \geq 0, \forall x \in \mathbb{R}^n, \forall \tau \geq t \right\}.$$

*http://www.casicorp.com

If no such t exists we define $\sigma(A, B) = \infty$ and if all t fulfill the requirement, we set $\sigma(A, B) = -\infty$.

The following theorem provides basic relations between the generalized condition number and the support number of a matrix pencil.

Theorem 18. *Let A and B be symmetric matrices.*

1. If B is spd, then $\sigma(A, B) = \lambda_{max}(A, B)$.
2. If B is spsd and $Null(B) \subseteq Null(A)$, then

$$\sigma(A, B) = \max\{\lambda \mid Ax = \lambda Bx,\ Bx \neq 0\},$$

   or, equivalently,

$$\sigma(A, B) = \lambda_{max}(Z^T AZ, Z^T BZ),$$

   where Z is such that the columns of Z span the range of B.
3. If B is not spsd, then $\sigma(A, B)$ is infinite.

Using the first statement of this theorem, we directly obtain the following result.

Proposition 19. *If A and B are both spd, the generalized condition number $\kappa(A, B)$ satisfies $\kappa(A, B) = \sigma(A, B)\sigma(B, A)$.*

Up to this point, we have reformulated the problem of bounding the condition number to that of bounding support numbers without actually simplifying the problem. The next proposition provides means to split the matrices A and B into simpler parts A_i, B_i with $i = 1, \ldots, q$, so that $\sigma(A_i, B_i)$ is easy to compute, in order to bound $\sigma(A, B)$.

Proposition 20 (Splitting)**.** *Split A and B into $A = A_1 + A_2 + \cdots + A_q$ and $B = B_1 + B_2 + \cdots + B_q$. If all B_i are psd, then $\sigma(A, B) \leq \max_{i=1,\ldots,q} \sigma(A_i, B_i)$.*

In our analysis of preconditioners, we will heavily rely on this proposition in order to bound the condition number.

The following propositions show some further basic properties of the support number that we will use later on. For a much more comprehensive survey of support theory, the reader is kindly referred to [20].

Proposition 21. *If B is psd, then*

$$\sigma(A + C, B) \leq \sigma(A, B) + \sigma(C, B).$$

If $C = B$, we have

$$\sigma(A + B, B) = \sigma(A, B) + 1.$$

Proposition 22. *If B and C is psd, then*

$$\sigma(A, B) \leq \sigma(A + C, B).$$

If A and $B - C$ are also psd, then

$$\sigma(A, B) \leq \sigma(A, B - C).$$

Proposition 23 (Triangle inequality)**.** *Let B and C be psd, then*

$$\sigma(A, C) \leq \sigma(A, B)\sigma(B, C).$$

4.3.2 *The Congestion-Dilation Theorem*

The splitting Proposition 20 gives means to regard the support of simpler splittings $\sigma(A_i, B_i)$ with $\sum_{i=1}^{q} A_i = A$ and $\sum_{i=1}^{q} B_i = B$ for $A, B \in \mathbb{R}^{n \times n}$ in order to bound $\sigma(A, B)$ but we have not yet specified what is meant by simpler. For this purpose, we follow the presentation of [59].

We assume A and B to be generalized Laplacians with corresponding graphs $G_A = (V, E_A, \omega_A)$ and $G_B = (V, E_B, \omega_B)$. Now, we consider a splitting of A, B into generalized Laplacians A_i, B_i, $i = 1, \ldots, q$ so that A_i corresponds to a graph G_{A_i} with a single edge $e_i = \{u_i, v_i\}$ and $\omega_{A_i}(e_i) = \omega_A(e_i)$ (or equivalently $G_{A_i} = G_A[\{u_i, v_i\}]$) while B_i corresponds to a graph G_{B_i} consisting of a single path path(e_i) connecting u_i and v_i. The weights $\omega_{B_i}(f)$ of the edges $f \in \text{path}(e_i)$ are yet to be determined. We note that this splitting defines an embedding of G_A into G_B as introduced in Subsection 4.1.2 and develop now the notion that path(e_i) *supports* the edge e_i.

Therefore, we fix i and consider an edge $f \in \text{path}(e_i)$. Observing that f can also be part of other pieces than G_{B_i}, we necessarily have

$$\sum_{i : f \in E(B_i)} \omega_{B_i}(f) = \omega_B(f). \tag{4.3}$$

Denoting the congestion of f in B by γ_f, we set

$$\omega_{B_i}(f) = \frac{\omega_{A_i}(e_i)}{\gamma_f}$$

which clearly fulfills (4.3). To simplify the notation, we assume without loss of generality that path(e_i) in B_i has length j with vertices indexed from 1 to $j + 1$ according to their order along the path and $e_i = \{1, j + 1\}$ in A_i. Since A_i and B_i are Laplacians we can use (4.2) and thus the problem of finding $\sigma(A_i, B_i)$ can be restated to finding τ such that

$$\tau \sum_{k=1}^{j} \frac{\omega_{A_i}(e_i)}{\gamma_k} (x_k - x_{k+1})^2 \geq \omega_{A_i}(e_i)(x_1 - x_{j+1})^2$$

for all $x \in \mathbb{R}^{j+1}$.

The following theorem [40] provides means to find τ that allows the path path(e_i) to support the edge e_i.

Theorem 24 (Congestion-dilation). *For any $x \in \mathbb{R}^{j+1}$, we have for $r = \sum_{i=1}^{j} \gamma_i$ that*

$$r \sum_{i=1}^{j} \frac{1}{\gamma_i}(x_i - x_{i+1})^2 \geq (x_1 - x_{j+1})^2.$$

Proof. By definition of r, we have

$$\begin{aligned}\sum_{i=1}^{j} \gamma_i \sum_{i=1}^{j} \frac{1}{\gamma_i}(x_i - x_{i+1})^2 &= \sum_{i=1}^{j} (\sqrt{\gamma_i})^2 \sum_{i=1}^{j} \frac{1}{(\sqrt{\gamma_i})^2}(x_i - x_{i+1})^2 \\ &\geq \left(\sum_{i=1}^{j} \sqrt{\gamma_i} \frac{1}{\sqrt{\gamma_i}}(x_i - x_{i+1}) \right)^2 = (x_1 - x_{j+1})^2,\end{aligned}$$

where the inequality follows from the Chauchy-Schwarz inequality. □

It should also be noted that one edge in A can also be split in several pieces where each piece is then supported with a path in B. Theorem 24 was implicitly used in the work of Vaidya [137] and Gremban [55] but they neither stated nor proved it. It was thereafter contributed by [17, 59] and generalized by [20, Theorem 4.4] in their rank-1 support theorem for symmetric, positive semidefinite matrices.

An obvious corollary of Theorem 24 can be achieved by setting $r = \delta\gamma$ where δ is the maximum dilation over all paths in B_i, $i = 1, \ldots, q$ and γ is the maximum congestion along all paths. In this case, each $\sigma(A_i, B_i)$ and therefore $\sigma(A, B)$ is bounded by $\delta\gamma$. Figure 17 demonstrates how the techniques developed so far can be applied to bound $\sigma(A, B)$.

It should also be mentioned that support theory also has an interesting interpretation in terms of electrical networks [40] which is outside the scope of this monograph.

4.3.3 *Preconditioners based on Support Theory*

Vaidya applied the techniques of support theory in the following way. Given a generalized Laplacian $A \in \mathbb{R}^{n \times n}$ with corresponding graph G_A possessing m edges, he constructs G_B to be the maximum-weight spanning tree of G_A and uses the corresponding Laplacian B as a preconditioner. This is depicted in Figure 18.

Since G_B is a subgraph of G_A by construction, we have $\sigma(B, A) \leq 1$. In order to analyze $\sigma(A, B)$, we define an embedding by mapping each edge of G_A onto the corresponding unique path in G_B. We then note that a path connecting the endpoints of an edge e possesses only edges with a weight greater than the weight of e. Otherwise, this would

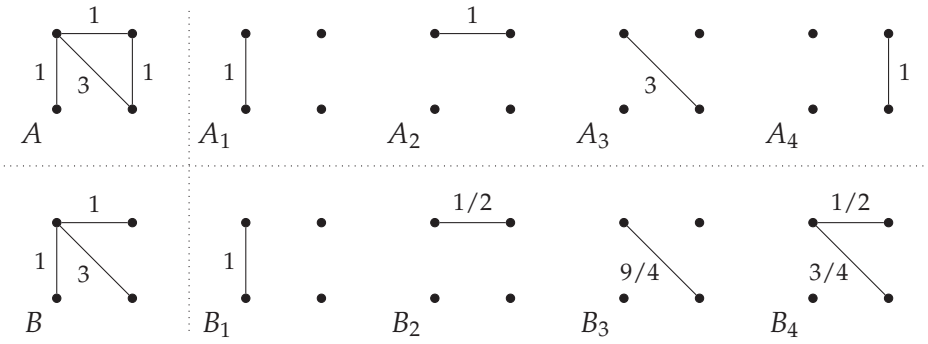


Figure 17: The graph A and B are split into $A = A_1 + A_2 + A_3 + A_4$ and $B = B_1 + B_2 + B_3 + B_4$ so that each edge in A_i is supported by a path in B_i. The maximum dilation of 2 has the path in B_4 while the maximum congestion of 2 is given by the path in B_2. The worst congestion-dilation product is therefore 4 and we have $\sigma(A, B) \leq 4$. Directly applying Theorem 24 to (A_4, B_4) results in $\sigma(A_4, B_4) \leq 2 + \frac{4}{3} = 3\frac{1}{3}$ which is a sharper bound on $\sigma(A, B)$. In this example, we have $\lambda_{\max}(A, B) = 2\frac{1}{3}$.

contradict the assumption that G_B is a maximum-weight spanning tree. Since we have m edges, we use $\frac{1}{m}G_B$ to support each edge resulting in a maximum congestion of m. The maximal path length (i.e., dilation) on n vertices is smaller than n and subsequently $\sigma(A, B) \leq mn$. Applying Proposition 19, we have $\kappa(A, B) = \sigma(A, B)\sigma(B, A) \leq mn$ as a worst case scenario. For an actual matrix the dilation and congestion might be much smaller as well as the bound on the condition number.

The computation of the maximum-weight spanning tree can be done by applying a minimum-weight spanning tree algorithm in $O(\log n)$ time on a linear number of processors [34] or sequentially $O(m \log n)$ time on A' with the reciprocal edge weights of A. Due to the tree structure of G_B, the Laplacian B can be factorized with no fill-in. At this, the degree of parallelism depends on the number of vertices on each level of the tree which also applies for the application of the preconditioner.

Based on this concept of subgraph preconditioners, Vaidya developed more sophisticated preconditioners. Also, Spielmann and Teng used these techniques to develop an ϵ-approximate $O(m^{1.31} \log(n/\epsilon))$ solver [126]. Further improvements resulted in an $\tilde{O}(m \log^2 n \log(1/\epsilon))$* solver which is already close to optimality, i.e., $O(m)$ [82]. With the help of the support theory also already well-known preconditioners like ICC [59], MICC [17] and block-Jacobi [20] were analyzed for their impact on the condition number.

*$\tilde{O}(f(n))$ is called "soft-O of $f(n)$" for a function f and defined as $O(f(n) \log^k f(n))$ for some k.

Figure 18: A regular graph A with its subgraph preconditioner B constructed by removing edges from A in order to get a spanning tree.

4.4 STEINER TREES AND GRAPHS

Gremban and Miller conducted research to develop preconditioners based on support theory that exhibit a higher degree of parallelism than Vaidya's subgraph preconditioners. They observed that parallelism in computation of a subgraph preconditioner, i.e., factorization and application, can be rather limited and remedied this by adding additional vertices and edges to the graph of the preconditioner that are not part of the original graph. In this way, a tree can be constructed with a high number of subtrees which can be treated independently from each other. This novel idea of a preconditioner in a higher dimensional space than the matrix that is to be preconditioned, is the main topic of Gremban's PhD thesis [55]. In this work, the analysis of their new kind of preconditioners was restricted to simple model problems, making them less attractive for general use.

Koutis, under the supervision of Miller, improved upon this work by providing more theoretical tools for analyzing these preconditioners as well as extending the idea of additional vertices to develop combinatorial multigrid methods in his PhD thesis [79]. In this section we will make much use of this work as well as from [80]. We start with a formal definition of this new kind of preconditioners.

4.4.1 *Introduction*

In the renowned work of Bienkowski et al. [18] a laminar decomposition was defined which we introduce here slightly different, similar to [79]. Due to the isomorphism between generalized Laplacians and undirected weighted graphs from Section 4.2 we will simplify our notation by no longer clearly differentiating between a Laplacian and its corresponding graph, i.e., we will use either terms interchangeably.

Definition 25 (Laminar decomposition)**.** Let $H = \{\mathcal{H}_0, \ldots, \mathcal{H}_h\}$ be a set of partitions of the graph $G = (V, E, \omega)$ with $\mathcal{H}_0 = V$. If every partition $\mathcal{H}_i$ refines partition $\mathcal{H}_{i-1}$ for $i = 1, \ldots, h$ and for all $v \in V$ the set $\{v\}$ is an element of a partition $\mathcal{H}_i$ than H is called a *laminar decomposition* of G.

A laminar decomposition naturally induces a tree structure as formally stated below.

Definition 26 (Laminar Steiner tree)**.** Let $H = \{\mathcal{H}_0, \dots, \mathcal{H}_h\}$ be a laminar decomposition of the graph $G = (V, E, \omega)$. We define a *laminar Steiner tree* $T_H = (V_T, E_T, \omega_T)$ with $\sum_{i=0}^{h} |\mathcal{H}_i|$ vertices and height h by letting level i of the tree consist of $|\mathcal{H}_i|$ vertices where each vertex corresponds to exactly one set in $\mathcal{H}_i$ for $i = 0, \dots, h$. Thus, each vertex $t \in V_T$ naturally corresponds to a set $V_t \subseteq V$. A vertex t on level i is connected to the vertex s on level $i-1$ by an edge e if and only if $V_t \subset V_s$. The weight of e is defined to be $\mathrm{out}(V_t)$. The internal vertices of a laminar Steiner tree are called *Steiner vertices*.

Remark 27. The term Steiner tree was coined by Miller [81] referring to the Steiner vertices in Steiner tree problems [71]. In other monographs [17, 21, 55, 56], Steiner trees go by the name *support trees* in reference to the notion of the support number in Definition 17.

Obviously, if $\mathcal{H}_h$ contains all singletons $\{v\}$ with $v \in V$ the laminar Steiner tree is balanced. It should be noted that this is not required by the definition.

Considering only the edges of T_H between two levels i and $i+1$, that is the induced graph by the vertices of T_H at level i and $i+1$, we obtain a set of stars with roots being the vertices at level i of T_H. We call these stars *level i stars* of a laminar Steiner tree. For a level i star S with root r and leaves $l_1, \dots, l_k$ we have the corresponding sets $V_r \subseteq V$ and $V_{l_1}, \dots, V_{l_k} \subseteq V$ with $\mathcal{V} = \{V_{l_1}, \dots, V_{l_k}\}$ being a partition of V_r by definition. We call $A := V[V_r]/_{\mathcal{V}}$ the to S corresponding *quotient subgraph* on the vertex set $l_1, \dots, l_k$.

The well-known *Cheeger constant* [32] for a graph G is defined as

$$h(G) = \min_{x \subseteq V(G)} \frac{\mathrm{out}(X)}{\min\{\mathrm{vol}(X), \mathrm{vol}(V \setminus X)\}}$$

and bounds the smallest positive eigenvalue of G, i.e., $\lambda_2(G) > 0$ which is also known as *Fiedler value*, as

$$\lambda_2(G) \geq \frac{h(G)^2}{4}.$$

Since the Cheeger constant relates the outflow to the volume of subgraphs it is a measure for the existence of bottlenecks in a graph or, to put it the other way around, for its conductance. Therefore, it is also used for the construction of well-connected networks, i.e., expander graphs. Also note that the Cheeger constant equals the sparsity of the sparsest cut (4.1).

To capture the notion of conductance in a laminar Steiner tree we relate the capacity from a subset S into a cluster T in a partition to the outflow capacity of T as

$$\gamma(S,T) = \frac{\text{cap}(S-T,T)}{\text{out}(T)}.$$

We will use this ratio to guarantee *sufficient capacity* in more general Steiner graphs which are composed of Steiner trees. Assume we have a graph $G = (V,E,\omega)$ and $W \subset V$. Let $H = \{\mathcal{H}_0,\ldots,\mathcal{H}_h\}$ be a laminar decomposition of $G[W]$ and T_H the corresponding laminar Steiner tree. By definition, the edges of T_H have weight $\text{out}(W_t)$ with respect to $G[W]$. Now $H' := \{\{V\},\{W,V\setminus W\},\mathcal{H}_1,\ldots,\mathcal{H}_h\}$ is a laminar decomposition of G and $T_{H'}$ the corresponding laminar Steiner tree. By construction, $T_{H'}$ contains a subtree $\bar{T}_{H'}$ whose vertices t correspond to subsets $W_t \subseteq W$. An edge of $\bar{T}_{H'}$ has weight $\text{out}(W_t)$ with respect to G. The ratio of the weight of an edge in $T_{G[W]}$ and the corresponding edge in T_G is just $\gamma(W_t,W)$. Based on this insight, we define Γ to compensate for insufficient capacity:

Definition 28 (Sufficient Capacity)**.** Given a graph $G = (V,E,\omega)$ and $W \subset V$. If $H = \{\mathcal{H}_0,\ldots,\mathcal{H}_h\}$ is a laminar decomposition of $G[W]$ defining a Steiner tree $T_H = (V_T,E_T,\omega_T)$ where each vertex $t \in V_T$ corresponds to a vertex set $W_t \subset W$, we define

$$\Gamma_W(T_H) = \sum_{\substack{\{s,t\}\in E_T \\ \text{lev}(s)<\text{lev}(t)}} \gamma(W_t,W)^{-1} T_H[\{s,t\}]$$

as the augmented Steiner tree $\Gamma(T_H)$ with sufficient capacity. Also, we define the inverse function Γ_W^{-1} such that $\Gamma_W^{-1}(\Gamma_W(T_H)) = T_H$.

Based on this, we can now properly define general Steiner graphs.

Definition 29 (Quotient and Steiner Graph)**.** Let P be an edge cut that partitions the vertices V of the graph $G = (V,E,\omega)$ into disjoint sets $V_1,\ldots,V_k$. Furthermore, let H_i be a laminar decomposition of V_i defining a Steiner tree T_i with root r_i. We define the *quotient graph* Q_P on the root set $R = \{r_i \mid i = 1,\ldots,k\}$ with edge weights $\omega(r_i,r_j) = \text{cap}(V_i,V_j)$ and the *Steiner graph* as $S_P = Q_P + \sum_{i=1}^{k} \Gamma_{V_i}(T_i)$ with respect to P. If it is evident by the context, we will just write S (resp. Q) for a Steiner graph (resp. Quotient graph) and omit the index P for the sake of a simplified notation.

Following [55], to support a matrix $A \in \mathbb{R}^{n\times n}$ with its laminar Steiner graph $S \in \mathbb{R}^{(n+m)\times(n+m)}$, we extend A to $\tilde{A}$ by

$$\tilde{A} := \begin{pmatrix} A & 0 \\ 0 & 0 \end{pmatrix},$$

which allows us to compute $\sigma(\tilde{A}, S)$. Numbering the leaves of S the same way as the vertices in A and than the additional Steiner vertices, we have

$$S = \begin{pmatrix} S_{11} & S_{12} \\ S_{12}^T & S_{22} \end{pmatrix}.$$

Let B_s be the resulting graph after eliminating all Steiner vertices in S, i.e., the Schur complement $B_S = S_{11} - S_{12}S_{22}^{-1}S_{12}^T$. Gremban showed that preconditioning $\tilde{A}$ with S is effectively equivalent to preconditioning A with B_S [55, p. 44], formally $\sigma(\tilde{A}, S) = \sigma(A, B_S)$ as in Proposition 6.1 of [20].

The difficulty arises when computing $\sigma(B_s, A)$, because B_S is typically dense with no closed analytical expression and $\sigma(S, \tilde{A})$ cannot be computed since no path in $\tilde{A}$ can support an edge in S connecting two Steiner vertices. For simple model problems, i.e., an unweighted d-dimensional grid, Gremban proved bounds on $\sigma(B_s, A)$ but it was only until the monograph of Maggs et al. [90] that analytical ways were found to bound $\sigma(B_s, A)$ for general sdd matrices A, assuming S is a laminar Steiner tree. Koutis generalized these results for Steiner graphs in [79].

In [90] the support number $\sigma(B_S, A)$ was related to S.

Lemma 30. *Let $S \in \mathbb{R}^{(n+m)\times(n+m)}$ be a Steiner graph with m Steiner vertices of $A \in \mathbb{R}^{n\times n}$ and $B_S \in \mathbb{R}^{n\times n}$ its Schur complement as a result of eliminating all Steiner vertices, then*

$$\sigma(B_S, A) = \max_{x\in\mathbb{R}^n} \min_{y\in\mathbb{R}^m} \left(\begin{pmatrix} x \\ y \end{pmatrix}^T S \begin{pmatrix} x \\ y \end{pmatrix} \right) / x^T A x.$$

Due to this Lemma, we will denote $\sigma(B_S, A)$ by $\sigma(S, A)$ whenever S is defined as the Steiner graph of A.

Remark 31. The claim of Lemma 30 is even true for all sdd matrices $A \in \mathbb{R}^{n\times n}$ and

$$B = \begin{pmatrix} W & U \\ U^T & T \end{pmatrix} \in \mathbb{R}^{(n+m)\times(n+m)},$$

with a nonsingular $W \in \mathbb{R}^{n\times n}$ [90].

We will now use this characterization of $\sigma(B_S, A)$ to give a slight generalization of Lemma 5.0.4 in [81] which again is a generalization of Proposition 23 to Steiner graphs.

Lemma 32 (Steiner support triangle inequality)**.** *Let $S \in \mathbb{R}^{(n+m)\times(n+m)}$ and $S' \in \mathbb{R}^{(n+m+k)\times(n+m+k)}$ be Steiner graphs of $A \in \mathbb{R}^{n\times n}$ whereas S' has m Steiner vertices and S has additionally k more Steiner vertices than*

S'. Also, let B_S, B'_S be the Schur complements of S, S' with respect to the elimination of the Steiner vertices. The following triangle inequality holds:

$$\sigma(B_S, A) \leq \sigma(S, S')\sigma(B'_S, A).$$

Proof. From Lemma 30 we have that for all $x \in \mathbb{R}^n$ there exists a $y \in \mathbb{R}^m$ so that

$$\begin{pmatrix} x \\ y \end{pmatrix}^T S' \begin{pmatrix} x \\ y \end{pmatrix} \leq \sigma(B'_S, A)(x^T A x). \tag{4.4}$$

Considering the Schur complement of S with respect to the elimination of the k additional Steiner vertices compared to S', we also get that for every vector $(x^T, y^T)^T$ there exists a vector $z \in \mathbb{R}^k$ such that

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}^T S \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq \sigma(S, S') \begin{pmatrix} x \\ y \end{pmatrix}^T S' \begin{pmatrix} x \\ y \end{pmatrix}. \tag{4.5}$$

Substituting (4.4) into (4.5) and applying Lemma 30 directly yields the desired bound on $\sigma(B_s, A)$. □

4.4.2 *Bounds for Steiner Graphs*

The next theorem from [79, Theorem 5.2.2] provides us with a tool to bound $\sigma(S, A)$ of a Steiner graph S of A by calculating $\sigma(T_i, A_i)$ where T_i is a Steiner tree of a cluster A_i of A.

Theorem 33. *Let $S = Q + \sum_{i=1}^{k} \Gamma_{V_i}(T_i)$ be the Steiner graph of $G = (V, E, \omega)$ with respect to an edge cut resulting in a partitioning $\mathcal{V} = \{V_1, \ldots, V_k\}$ that induces the clusters $A_i = G[V_i]$ with $i = 1, \ldots, k$. If $h = \max_i \text{height}(T_i)$ we have $\sigma(A, S_P) \leq 2h + 1$ and*

$$\sigma(S, A) \leq (2h + 1)(1 + \max_{i=1,\ldots,k} (\sigma(\Gamma_{V_i}(T_i), A_i))) - 1.$$

Proof. Starting with the upper bound on $\sigma(A, S)$ we embed A into S and route each edge of A via the shortest path in S. The length of this path is shorter than two times the height of the highest tree plus one edge in the quotient graph connecting both trees so that we have a dilation of $2h + 1$. By the construction of each Steiner tree $\Gamma_{V_i}(T_i)$ and the sufficient capacity from Definition 28 the capacity of edge edge in S assures that the embedding has congestion of 1. By the Congestion-Dilation Theorem 24 we have $\sigma(A, S) \leq 2h + 1$.

To bound $\sigma(S, A)$ we first note that $\sigma(S, A) = \sigma(S + A, A) - 1$ by Proposition 21. By Lemma 32, we have that

$$\sigma(S + A, A) \leq \sigma(S + A, S + A - Q)\sigma(S + A - Q, A).$$

In order to bound the multiplicand $\sigma(S+A, S+A-Q)$, we route each edge $d = \{r_a, r_b\}$ in Q through multiple paths in $S+A$. Again, the sufficient capacity of the trees $\Gamma_{V_i}(T_i)$ ensures that the congestion is 1. To better see that, consider an edge e_1 in A that was supported by path(e_1) with $d \in$ path(e_1) when bounding $\sigma(A, S)$. We route a portion of $\omega(e)$ through the unique path from r_a through e_1 to r_b with congestion less or equal than 1. Analogously doing this for all other such edges $e_2, \ldots, e_l$ completes the necessary support for d, since we know by definition that $\omega(d) = \omega(r_a, r_b) = \text{cap}(V_a, V_b) = \sum_{i=1}^{l} \omega(e_i)$ which proves the congestion of 1. The maximal dilation obviously is $2h+1$.

Looking at the multiplier we see that $\sigma(S+A-Q, A) = \sigma(S-Q, A)+1$. Since $S-Q = \Gamma_{V_i}(T_i)$, i.e., disjoint Steiner trees, we have that $\sigma(S-Q, A) \leq \max_{i=1,\ldots,k}\{\sigma(\Gamma_{V_i}(T_i), A_i)\}$ by Proposition 20 . In total, we have that

$$\sigma(S+A, A) \leq (2h+1)(1+\max_{i=1,\ldots,k}\{\sigma(\Gamma_{V_i}(T_i), A_i)\})$$

which proves the statement. □

Using this theorem for Steiner graphs we can deduce a bound for $\sigma(T, A)$ with T being a Steiner tree by calculating the support of smaller partial Steiner trees. The following theorem and technique was invented and used by Iannis Koutis in [79, p. 59] to derive a more general upper bound. Based on this proof techniques and private communication with him, we derive a bound that can be easily computed.

Theorem 34. *Let $H = \{\mathcal{H}_0, \ldots, \mathcal{H}_h\}$ be a laminar decomposition with corresponding Steiner tree T of a graph $G = (V, E, \omega)$. Furthermore, for $i = 0, \ldots, h-1$ let S_j^i be the level i stars of T, A_j^i the corresponding quotient subgraphs and $\lambda_i = \max_j \sigma(S_j^i, A_j^i)$ with $j = 1, \ldots, |\mathcal{H}_i|$. We have*

$$\sigma(T, G) \leq \prod_{j=0}^{h-1} 3(1+\lambda_j).$$

Proof. We start by defining a sequence $G_i = G/_{\mathcal{H}_i}$ for $i = 0, \ldots, h$ so that each level i of T has a corresponding graph G_i. Furthermore, we define T_i as the subgraph of T induced by vertices which are on level $j \geq i$. The weight of an edge $\{u, v\}$ of T with $\text{lev}(u) = i$ and $\text{lev}(v) = i-1$ fulfills by construction $\omega(\{u, v\}) = \text{vol}(u')$ with the corresponding vertex u' in G_i. Therefore T_i has sufficient capacity and thus $T_i + G_i$ is a Steiner graph.

With the help of Lemma 30 the same proposition for Steiner trees as in Proposition 22 can be derived and we have

$$\sigma(T, G) \leq \sigma(T + \sum_{j=0}^{h-1} G_j, G).$$

The graph $T + \sum_{j=0}^{h-1} G_j$ is a hierarchy of quotient graphs G_j connected by the edges from level j to $j-1$ of T. We bound $\sigma(T + \sum_{j=0}^{h-1} G_j, G)$ by consecutively bounding each level in its hierarchy. We show by induction that for all $i \leq h-1$ we have

$$\sigma(T_i + \sum_{j=i}^{h-1} G_j, G) \leq \prod_{j=i}^{h-1} 3(1+\lambda_j). \tag{4.6}$$

Starting with the base case for level $i = h-1$, we have to bound $\sigma(T_{h-1} + G_{h-1}, G)$. The graph T_{h-1} is a forest consisting of level $h-1$ stars S_j^{h-1}, $j = 1, \ldots, |\mathcal{H}_{h-1}|$ with sufficient capacity whereas G_{h-1} acts as its quotient graph. Therefore, we can apply Theorem 33 and have

$$\begin{aligned}\sigma(T_{h-1} + G_{h-1}, G) &\leq 3(1 + \max_{j=1,\ldots,|\mathcal{H}_{h-1}|} \{\sigma(S_j^{h-1}, A_j^{h-1})\}) - 1 \\ &= 3(1+\lambda_{h+1}) - 1.\end{aligned}$$

Analogously, we get for level $i < h-1$

$$\begin{aligned}\sigma(T_i - T_{i+1} + G_i, G_{i+1}) &\leq 3(1 + \max_{j=1,\ldots,|\mathcal{H}_i|} \{\sigma(S_j^i, A_j^i)\}) - 1 \\ &= 3(1+\lambda_i) - 1.\end{aligned}$$

We can now use Lemma 32 to get

$$\begin{aligned}\sigma(T + \sum_{j=i}^{h-1} G_j, G) &\leq \sigma(T_i + \sum_{j=i}^{h-1} G_j, T_{i+1} + \sum_{j=i+1}^{h-1} G_j)\sigma(T_{i+1} + \sum_{j=i+1}^{h-1} G_j, G) \\ &= \sigma(T_i - T_{i+1} + G_i + T_{i+1} + \sum_{j=i+1}^{h-1} G_j, T_{i+1} + \sum_{j=i+1}^{h-1} G_j) \\ &\quad \cdot \sigma(T_{i+1} + \sum_{j=i+1}^{h-1} G_j, G) \\ &= (1 + \sigma(T_i - T_{i+1} + G_i, T_{i+1} + \sum_{j=i+1}^{h-1} G_j)) \\ &\quad \cdot \sigma(T_{i+1} + \sum_{j=i+1}^{h-1} G_j, G) \\ &\leq (1 + \sigma(T_i - T_{i+1} + G_i, G_{i+1}))\sigma(T_{i+1} + \sum_{j=i+1}^{h-1} G_j, G),\end{aligned}$$

where the last inequality follows from the generalization of Proposition 22 to Steiner graphs with the help of Lemma 30. This proves the induction hypothesis (4.6) whereof the claim follows. □

This theorem can be easily generalized in the way that not for every tree level i a corresponding G_i needs to be defined. In this case the stars become partial trees that need to be supported by their corresponding quotient subgraphs. Calculating these supports becomes in practice more expensive but it results in a tighter bound on $\sigma(T, G)$.

4.5 FLOWS IN NETWORKS

In this section we give a short overview of network flow problems. Thereafter, we introduce an algorithm that allows us to find a flow in a network with multiple sinks so that the flow-to-demand ratio of each sink is maximized. This will allow us to analyze a forest of Steiner trees as preconditioner, or in other words, a block-Jacobi application of a Steiner tree preconditioner.

4.5.1 *Introduction*

We will base this short introduction on the pioneering work of Ford and Fulkerson on flows in networks [50] and start with the following definition.

Definition 35 (Network flow)**.** Let $N = (V, E, \omega)$ be a directed graph, called *network*, with capacity function $c : V \times V \to \mathbb{R}_0^+$ and two distinguished nodes s and t, resp. named *source* and *sink*. A *flow* is a function $f : V \times V \to \mathbb{R}_0^+$ that satisfies the capacity constraints $f(x, y) \leq c(x, y)$ for all $(x, y) \in V \times V$ and for every $x \in V$ the conservation constraints

$$\sum_{(y,x)\in E} f(y, x) - \sum_{(x,y)\in E} f(x, y) = \begin{cases} -v, & x = s \\ 0, & x \neq s, t \\ v, & x = t \end{cases},$$

where $v \geq 0$ is the *flow value*. The set of all flows in a given network is denoted by $\mathcal{F}$.

Remark 36. In case of an undirected graph $N = (V, E, \omega)$ with $E \subseteq V \otimes V$ and capacity c, the capacity constraints are defined as

$$\begin{aligned} f(x, y) &\leq c(\{x, y\}), \\ f(y, x) &\leq c(\{x, y\}), \\ f(x, y) \cdot f(y, x) &= 0 \end{aligned}$$

for $x, y \in V$ [50, p. 23].

The subject of a network flow problem is to determine a flow f with maximal flow value v in a given network. In [49], Ford and Fulkerson proved that this problem is equivalent to finding a minimal cut.

Theorem 37 (Max-Flow Min-Cut)**.** *For any network the maximal flow value from s to t is equal to the minimal cut capacity of all cuts separating s and t.*

It should be noted that Definition 35 can also be canonically extended to a network $N = (V, E, \omega)$ with multiple sources $s \in S$ and sinks $t \in T \subseteq V \setminus S$. In this case, the problem of finding a flow with maximal flow value in N can be reduced to an equivalent problem with only one source s^* and one sink t^*. This is accomplished by creating a new network N^* consisting of $V(N)$ plus two adjoint vertices s^* and t^* as well as edges $E(N)$ plus edges (s^*, s) for all $s \in S$ and edges (t, t^*) for all $t \in T$ with capacity ∞ [50, p. 15].

In order to solve a network flow problem Ford and Fulkerson proposed the *Max-Flow Labeling* algorithm [50, p. 17] which is guaranteed to terminate if all capacities are rational numbers. An implementation of this algorithm is the *Edmonds-Karp* algorithm [42] with time complexity of $O(|E| \cdot |V|^2)$ for a network $N = (V, E, \omega)$.

4.5.2 *Network Flows with Multiple Sinks*

In [95] it was stated, that the maximization of the flow value in a network may not be the only objective in certain cases like economical applications with multiple sources and sinks. Here, an often encountered objective is to distribute the flow "fairly" among the sinks and sources of a network which could mean for instance the maximization of the minimum flow into a sink. In this case the Max-Flow Labeling algorithm can not directly be applied, because it finds a flow with maximum flow value without worrying about additional constraints. Furthermore, it is obvious that under these circumstances a network with multiple sources and sinks cannot just be transformed in a network with a single source and single sink, whereon most analytical results are based.

With these considerations in mind, Meggido showed in [95] how a flow f in $N = (V, E, \omega)$ with sources S and sinks T can be found that is fair in the sense that $\max_{s \in S}\{f(s, x) \mid (s, x) \in E\}$ is minimized and $\min_{t \in T}\{f(x, t) \mid (x, t) \in E\}$ is maximized. First, this flow problem is reduced to two equivalent problems, i.e., maximizing $\min_{t \in T}\{f(x, t) \mid (x, t) \in E\}$ on a network with a single source and multiple sinks and minimizing $\max_{s \in S}\{f(s, x) \mid (s, x) \in E\}$ on a network with multiple sources and a single sink which can both be treated analogously. Subsequently the theory for finding a sink-optimal flow is established.

We will generalize these results in the way that we introduce a
$d(t)$ *demand* $d(t) > 0$ for each sink $t \in T$. Our objective will be to maximize

the minimal ratio of a flow $f(x,t)$ into t to its demand $d(t)$. For this purpose, some necessary definitions and results from [95] will be restated for the sake of completeness in order to derive a generalization of Theorem 4.6.

For a network $N = (V, E, \omega)$ let $g : V \times V \to \mathbb{R}$ be a function, we define for $X, Y \subseteq V$, $g(X,Y)$

$$g(X,Y) := \sum_{(x,y)\in X\times Y} g(x,y)$$

and for every node $x \in V$ $\text{net}(g,x)$

$$\text{net}(g,x) := g(V,\{x\}) - g(\{x\},V).$$

Given a flow f, we will denote by $T(f)$ the $|T|$-tuple with components $\text{net}(f,t)$, $t \in T$ arranged in increasing order of magnitude and call it *sink flow*. Likewise, $S(f)$ denotes the same way arranged $|T|$-tuple with components $\text{net}(f,t)/d(t)$, $t \in T$ and call it *sink-demand satisfaction*. If g is a $|T|$-tuple, we define $g(A)$

$$g(A) = \sum_{t\in A} g(t)$$

for $A \subseteq T$.

Let u, v be two $|T|$-tuples arranged in increasing order of magnitude, we say u is *lexicographically greater* than v if there exists $i_0 \in \{1,\dots,n\}$ such that $u_{i_0} > v_{i_0}$ and $u_i = v_i$ for $i = 1,\dots,i_0 - 1$.

Definition 38. A flow f^* is called a *sink-demand-optimal flow* if for every $f \in \mathcal{F}$, $S(f)$ is not lexicographically greater than $S(f^*)$.

A multiple sinks network can be characterized by its outflows depending on active sinks.

Definition 39 (Characteristic function)**.** The *characteristic function* of a network $N = (V, E, \omega)$ with sinks T is a function $v : \mathcal{P}(T) \to \mathbb{R}_0^+$,

$$v(A) := \max\{f(V,A) - f(A,V) \mid f \in \mathcal{F}\}$$

for $A \subseteq T$.

With the help of the characteristic function the set $\{T(f) \mid f \in \mathcal{F}\}$ in a network can be expressed.

Lemma 40. *Given a network $N = (V, E, \omega)$ with sinks T and let $(g_t)_{t\in T}$ with $g_t \geq 0$ for $t \in T$. A necessary and sufficient condition for the existence of a flow f such that for each $t \in T$*

$$g_t = \text{net}(f,t)$$

is that for every $A \subseteq T$

$$g(A) \leq v(A).$$

Proof. The necessity follows directly from the definition of $v(A)$ and $\mathrm{net}(f,t)$. In order to prove sufficiency we extend the network N with a new sink t^* in the usual way and treating the other sinks $t \in T$ as normal vertices. Formally, we have $N^* = (V^*, E^*, \omega^*)$ with c^* where $V^* = V \cup \{t^*\}$, $E^* = E \cup \{(t,t^*) \mid t \in T\}$ and

$$c^*(x,y) = \begin{cases} g_t, & (x,y) = (t,t^*) \\ c(x,y), & (x,y) \in E \end{cases}.$$

We will prove that the set of edges forming the partition $\{V, \{t^*\}\}$ is a minimal cut. Let $\{X, V^* \setminus X\}$ be another partition with $X \subset V^*$, source $s \in X$ and $t^* \in V^* \setminus X$. We have

$$\begin{aligned} c^*(X, V^* \setminus X) &= c^*(X, V \setminus X) + c^*(X \cap T, \{t^*\}) \\ &= c(X, V \setminus X) + g(X \cap T) \geq v(T \setminus X) + g(X \cap T) \\ &\geq g(T) = c^*(V, \{t^*\}). \end{aligned}$$

From Theorem 37, we have that there exists a maximal flow f^* in N^* and necessarily $f^*(t,t^*) = g_t$ for $t \in T$. Therefore, if we consider the restriction f of f^* onto E, we have for every $t \in T$

$$\begin{aligned} \mathrm{net}(f,t) &= f^*(V,t) - f^*(t,V) = \mathrm{net}(f^*,t) + f^*(t,t^*) \\ &= f^*(t,t^*) = g_t \end{aligned}$$

and the claim follows. □

We will now state an obvious but lesser known result.

Lemma 41. *Let $b = (b_i)_{i=1}^n$ be a sequence with $b_i > 0$ and $A = \{a = (a_i)_{i=1}^n \mid a_i \geq 0, \sum_{i=1}^n a_i \leq k\}$ a set of sequences. For $a \in A$, we have*

$$f(a) := \min\{\frac{a_i}{b_i} \mid i = 1, \ldots, n\} \leq k / \sum_{i=1}^n b_i.$$

Proof. Let $l = \sum_{i=1}^n b_i$ and $c = (c)_{i=1}^n$ with $c_i = \frac{b_i}{l}k$. We have $c_i \geq 0$, $\sum_{i=1}^n c_i = k$ and $f(c) = k/l$. We suppose, per absurdum, that there is $a' \in A$ such that $f(a') > k/l$. We examine for that the sequence $(a'_i - a_i)_{i=1}^n$ and reorder it to be monotonically increasing with no loss of generality. If $a'_i - a_i \leq 0$ for all i we obviously have $f(a') \leq k/l$ in contrast to our assumption. Now, if $a'_i - a_i = 0$ for all $i \leq i_0$ and $a'_i - a_i > 0$ for $i > i_0$, we have that $\sum_{i=1}^n a'_i > \sum_{i=1}^n a_i = k$ which is a contradiction to $a' \in A$. In the last case, we assume $a'_i - a_i < 0$ for $i \leq i_0$, $a'_i - a_i = 0$ for $i_0 < i \leq i_1$ and $a'_i - a_i > 0$ for $i > i_1$. We have the contradiction that $f(a') \leq \frac{a'_1}{b_1} < \frac{a_1}{b_1} = k/l$ and the claim follows. □

We will repeatedly use Lemma 41 in the proof of the algorithm in Theorem 42 which allows us to construct a flow that maximizes the minimal fraction of flows into sinks to their demands.

We denote with

$$G := \{g = (g_t)_{t\in T} \mid \forall A \subseteq T, \sum_{t\in A} g_t \leq v(A), g_t \geq 0\} \tag{4.7}$$

the set of all sink flows due to Lemma 40. Based on this definition we have

$$H := \{h = (h_t)_{t\in T} \mid h_t = \frac{g_t}{d(t)}, g \in G\}. \tag{4.8}$$

Furthermore, we define the function $\Theta : \mathbb{R}^{|T|} \to \mathbb{R}^{|T|}$ that rearranges the components of a vector in order of increasing magnitude.

Theorem 42. *Let $T_0 = \emptyset$ and $w_0(A) = v(A)$ for every $A \subseteq T$. While $T_k \neq T$ and starting with $k = 0$, we define recursively*

$$\alpha_k = \min\{w_k(A)/d(A) \mid \emptyset \neq A \subseteq T \setminus T_k\}, \tag{4.9}$$

$$T_{k+1} = T_k \cup \bigcup\{A \mid \emptyset \neq A \subseteq T \setminus T_k, w_k(A) = \alpha_k d(A)\}, \tag{4.10}$$

$$h^*_t = \alpha_k, \quad (t \in T_{k+1} \setminus T_k), \tag{4.11}$$

$$g^*_t = d(t)h^*_t, \quad (t \in T_{k+1} \setminus T_k), \tag{4.12}$$

$$w_{k+1}(A) = \min\{v(A \cup B) - g^*(B) \mid B \subseteq T_{k+1}\}, \quad (A \subseteq T \setminus T_{k+1}). \tag{4.13}$$

Under these conditions there is k_0, $1 \leq k_0 \leq |T|$, such that $T_{k_0} = T$ and h^* is the lexicographical maximum of $\Theta(H)$ with corresponding sink flow $g^* \in G$.

Proof. From (4.10) we obviously have

$$\emptyset = T_0 \subsetneq T_1 \subsetneq \ldots \subsetneq T$$

and therefore a k_0 as specified exists. We first prove that g^* is a sink flow, i.e., $g^* \in G$. Let A be any nonempty subset of T and k the greatest index so that $A_k := A \cap (T_{k+1} \setminus T_k) \neq \emptyset$. Furthermore, let $B = A \setminus A_k$. We have

$$\begin{aligned} g^*(A) &= g^*(A_k) + g^*(B) = \alpha_k d(A_k) + g^*(B) \\ &\leq w_k(A_k) + g^*(B) \leq v(A_k \cup B) = v(A) \end{aligned}$$

and consequently $g^* \in G$. From (4.12), it obviously follows that $h^* \in H$.

Next, we show that (α_k) is an increasing sequence and therefore $\Theta(h^*) = h^*$. Let $\emptyset \neq A \subseteq T \setminus T_{k+1}$, $B \subseteq T_{k+1} \setminus T_k$ and $C \subseteq T_k$ such that

$$\alpha_{k+1} = \frac{w_{k+1}(A)}{d(A)} = \frac{v(A \cup B \cup C) - g^*(B \cup C)}{d(A)}$$

which must exists because of (4.10) and (4.13). Thereby, we have that

$$\begin{aligned}\alpha_{k+1} &= \frac{v(A\cup B\cup C)-g^*(B)-g^*(C)}{d(A)}\\ &\geq \frac{w_k(A\cup B)-g^*(B)}{d(A)} > \frac{\alpha_k d(A\cup B)-\alpha_k d(B)}{d(A)} = \alpha_k.\end{aligned}$$

In order to show h_t^* is the lexicographical maximum of $\Theta(H)$, we suppose, per absurdum, that there is an h so that $\Theta(h)$ is lexicographically greater than h^*. From Lemma 40, we have for every $A \subseteq T$, $A \neq \emptyset$ that

$$g(A) \leq v(A) \tag{4.14}$$

and therefore

$$\begin{aligned}\min\{h_t \mid t\in T\} &= \min_{A\subseteq T}\min\{\frac{g_t}{d(t)} \mid t\in A\}\\ &\leq \min\{\frac{v(A)}{d(A)} \mid \emptyset\neq A\subseteq T\} = \alpha_0,\end{aligned}$$

with the inequality following from Lemma 41. By our assumption, $\Theta(h)$ is lexicographically greater than h^* so that

$$\min\{h_t \mid t\in T\} \geq \min\{h_t^* \mid t\in T\} = \alpha_0.$$

By (4.10), we have

$$T_1 = \bigcup\{A \mid v(A) = \alpha_0 d(A)\}$$

and thus, together with (4.14), it follows for every $t \in T_1$ that $h_t = \alpha_0 = h_t^*$ and $g_t = g_t^*$.

We assume as induction hypothesis that for every $t \in T_k$, we have $h_t = h_t^*$ and $g_t = g_t^*$. It follows for every $A \subseteq T \setminus T_k$, $A \neq \emptyset$ and $B \subseteq T_k$

$$g(A) \leq v(A\cup B) - g(B) = v(A\cup B) - g^*(B). \tag{4.15}$$

Using this, we have

$$\begin{aligned}\min\{h_t \mid t\in T\setminus T_k\} &= \min_{A\subseteq T\setminus T_k}\min\{\frac{g(t)}{d(t)} \mid t\in A\}\\ &\leq \min\{\frac{g(A)}{d(A)} \mid A\subseteq T\setminus T_k\}\\ &\leq \min\{\frac{v(A\cup B)-g^*(B)}{d(A)} \mid A\subseteq T\setminus T_k, B\subseteq T_k\}\\ &= \alpha_k.\end{aligned}$$

On the other hand, we also have by our induction hypothesis and the fact that $\Theta(h)$ is lexicographically greater than h^* that

$$\min\{h_t \mid t\in T\setminus T_k\} \geq \min\{h_t^* \mid t\in T\setminus T_k\} = \alpha_k.$$

From (4.10) and (4.15) we have $h_t = \alpha_k = h_t^*$ and also $g_t = g_t^*$ for every $t \in T_{k+1} \setminus T_k$ and inductively it follows that $\Theta(h) = h^*$ which is a contradiction to our assumption. □

For the special case of equal demands, i.e., setting $d(t) = 1$ for all $t \in T$, the same result as in the original Theorem 4.6 from [95] follows.

4.6 HARDWARE AWARE PRECONDITIONERS

In High Performance Computing (HPC) a computer cluster's interconnect topology typically features a hierarchy. For instance, the bandwidth and speed of the node interconnect is almost always much slower than the interconnect between two CPUs on the same node. This gives reason for many preconditioners to follow a block-Jacobi strategy, i.e., dropping couplings in the preconditioner that would cause internode communication. In this section we analyze Steiner graph preconditioners as presented in Section 4.4 under this aspect with the help of the algorithm from the previous section. Therefore, we will split a Steiner tree into a set of smaller Steiner trees and show how these disconnected trees support the corresponding graph.

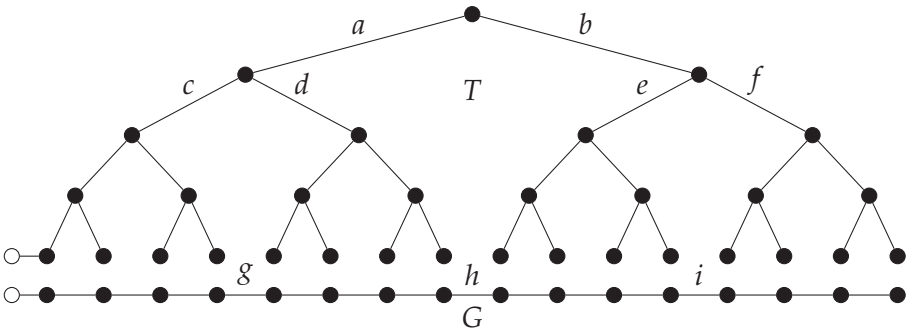
Based on this result, we will then show how a block-Jacobi Steiner tree preconditioner can be adapted to a given network topology in order to minimize the runtime of the solver. We will elaborate on the underlying hardware model and evaluate our method on a model problem.

4.6.1 *Block-Jacobi Steiner Tree*

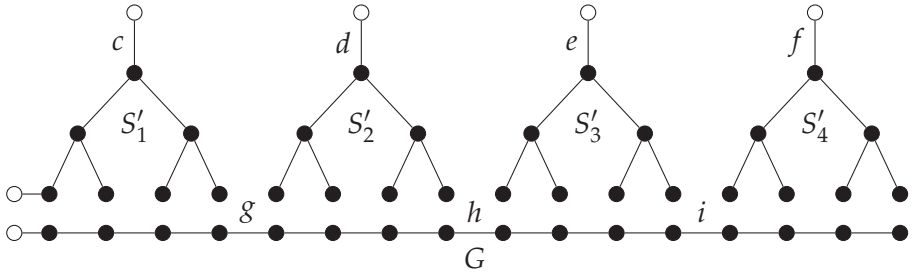
Assume a graph $G = (E, V, \omega)$ with at least one vertex with a loop or equivalently an edge to an implicit zero-valued boundary vertex (as defined in Remark 13) and a laminar decomposition $H = \{\mathcal{H}_0, \ldots, \mathcal{H}_h\}$ of G that defines a Steiner tree T_H. We will look at subgraphs of T_H consisting of edges and vertices at level $j \geq k$ which we denote by S and the quotient graph $Q = V/_{\mathcal{H}_k}$. For $\mathcal{H}_k = \{K_1, \ldots, K_n\}$, the level k vertices of T_H are denoted by v_i^k, $i = 1, \ldots, n$. We note that these vertices correspond to the vertices in Q and to the roots of the n disconnected trees in S which allows us to identify them.

We denote by D the graph on the same vertex set as S with only edges from vertices v_i^k to an implicit zero-valued boundary vertex b^0 with the same weight as the corresponding edge from level k to $k-1$ in T_H. We set $S' = S + D$ and enumerate with regard to v_i^k the pairwise not connected trees with S_i' (resp. S_i) in S' (resp. S) where $i = 1, \ldots, n$. We will use S' as a preconditioner for G. Figures 19a and 19b illustrate the construction of S' on a simple model problem.

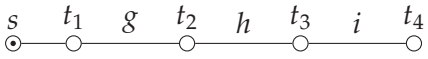
Bounding $\sigma(G, S')$ is straightforward: An edge $\{u, v\}$ with $u, v \in K_i$ can be supported by a path with maximal dilation $2d$ where $d = h - k$



(a) Laminar Steiner tree T above the corresponding graph G whereas both have an edge to an implicitly zero-valued boundary vertex. The weights $(a, \ldots, i)$ of some edges of T and G are given.



(b) Removing all edges with vertices below level 2 from T in (a) and adding edges to an implicit zero-valued boundary vertex b^0 with the same weight as the removed edges at level 2 results in four pairwise not connected trees $S'_1, \ldots, S'_4$. The zero-valued boundary vertex b^0 is represented by a set of white nodes for better illustration albeit it is effectively only one node.



(c) To support the edges c, d, e, f in (b), a network flow problem N with one source s and multiple sinks $t_1, \ldots, t_4$ with demand $d(t_1) = c, \ldots, d(t_4) = f$ is solved to determine the congestion. Traversing the edge with weight g means descending S'_1, moving over one edge in G and ascending S'_2 which is a path of length 5. The maximal dilation is therefore less then the maximal length of a path in N, i.e., 4, times 5.

Figure 19

and congestion 1 due to the sufficient capacity property of the laminar Steiner trees S_i. In case of an edge $e = \{u, v\}$ with $u \in K_i$ and $v \in K_j$, $i \neq j$, we support e by a path from u to the root of S_i over the implicit zero-valued boundary vertex in D to S_j and finally to v with maximal dilation $2(d+1)$ and congestion 1. Consequently, we have $\sigma(G, S') \leq 2(d+1)$.

Bounding $\sigma(S', G)$ is a more difficult task. By Lemma 30 and Remark 31, we have

$$\sigma(S', G) \leq \sigma(S', S+G)\sigma(S+G, G). \tag{4.16}$$

Since $\sigma(S+G, G) = \sigma(S, G) + 1 \leq \max_{i=1,\dots,n}\{\sigma(S_i, G_i)\} + 1$ where $G_i = G[K_i]$, we can apply Theorem 34 to bound the last factor. In order to bound $\sigma(S', S+G)$, we note that $\sigma(S', S+G) = \sigma(S+D, S+G) \leq \sigma(S, S+G) + \sigma(D, S+G) \leq 1 + \sigma(D, S+G)$.

To support the edges in D by paths in $S+G$ we construct a network flow problem with a source s and multiple sinks $t_1, \dots, t_n$, where each t_i corresponds to an edge $e_i = \{v_i^k, b^0\}$ in D with demand $d(t_i) = \omega(e_i)$. The network N itself is composed of a low-stretch spanning tree in Q where each v_i^k is set to be the sink t_i. Furthermore, attached to each sink t_i is an edge e_{s_i} to a source s_i if $S_i + G_i$ has an edge to b^0. We set the weight of e_{s_i} to the maximal flow value f of the network $\Gamma_{G_i}^{-1}(S_i)$ with source b^0 and sink e_i^k, where $\Gamma_{G_i}^{-1}(S_i)$ is the reduced tree from Definition 28. Figure 19c depicts this with the help of a model problem. We note that the maximal flow value in $\Gamma_{G_i}^{-1}(S_i)$ is easy to compute since $\Gamma_{G_i}^{-1}(S_i)$ is a tree. We assume the network N to have multiple sinks and one source without loss of generality as remarked in Section 4.5.

A flow from the source to a sink in N naturally corresponds to the routing of paths from sources in $S+G$ to the edges in D with maximal congestion 1 due to the sufficient capacity property of S as a subgraph of T_H. Let f^* be a sink-demand-optimal flow computed by the algorithm in Theorem 42 and let s be the minimal component of the sink-demand satisfaction $S(f^*)$. Clearly, supporting the edges of D has congestion $1/s$.

For dilation, we note that traversing one edge in N means a path in $S+G$ consisting of $d = h - k$ edges in S_i, one edge in G, and d edges in S_j , $i \neq j$. The dilation is then given by the length of the longest path in N times $2d+1$.

4.6.2 *Hardware Architecture Model*

Assume a typical HPC cluster that features an interconnect hierarchy as illustrated in Figure 20. For each pair of cores we can measure the cost of delivering a message of size 1 from one to another. Depending on the location of these two cores, a message will take a different path

from one to the other traversing some levels of the hierarchy. Let l be the minimum level that was visited by the message traveling between two cores. For each level l we can specify the communication time for a message of size 1 that has the lowest level l on his path. We define the communication time with g_l, starting from the slowest ($l = 0$) to the fastest level ($l = k$). The fastest level is commonly a cache that two or more cores can access coherently. In order to determine the values g_l, small benchmarks consisting of send and receive tests can be applied.

The performance of solving a system of linear equations with an iterative method on a cluster strongly depends on the communication cost. Let us consider for instance one iteration of the Chebyshev iteration. Some components of the updated approximate solution need to be communicated from one processing unit to its neighbors which results in communication over the whole interconnect hierarchy. Herewith, the slowest interconnect determines the cost of this exchange operation that is acting as a synchronization point between computation in each iteration step. Thus, the *Bulk-Synchronous Parallel (BSP)* Model [138] can be applied to model the parallel runtime of an iterative solver.

Let g_l be the communication cost for communication on level l as defined above. The BSP model assumes that transferring a message of size m takes time mg_l which is overly pessimistic on most architectures because the communication latency is accounted for m times. Since we will be considering only small messages, this simplification in the BSP model will not impair our analysis and we will assume that sending a message of size m is equal to sending m times a message of size 1. On a cluster with p processing units let w_i^S be the work time of one iteration step of a solver on processing unit i and h_{il}^S the number of messages that need to be sent or received in one iteration step over level l. Let $w = \max_{i=1,\dots,p} w_i^S$ be the maximum work time per iteration step and $e = \max_{i=1,\dots,p} \sum_{l=0}^{k} h_{il}^S g_l$ the maximum time a processing unit is occupied by data exchange. Thus, the time needed for the completion of one iteration step is $t = w + e$. If n is the number of iterations solver S needs to approximate a solution then the total compute time is $t_{\text{total}} = n(w + e)$.

Using a preconditioner in an iterative method allows the reduction of the number of iterations n by adding a work time overhead w_i^P per iteration step on processing unit i. The same applies for the communication time depending on the construction of the Steiner graph preconditioner. For instance, a block-Jacobi Steiner tree preconditioner as described in Section 4.6.1 with p trees where each tree is applied by a different processing unit results in no additional communication overhead h_{il}^P whereas a complete Steiner tree would result in at least $\max_{i=1,\dots,p} \sum_{l=0}^{k} h_{il}^P g_l$ additional communication time. Between these extremes, one can imagine block-Jacobi Steiner trees such that no com-

munication under a certain level is necessary therefore reducing the communication cost per iteration but increasing the necessary iteration steps. Given a hardware architecture with known g_l and an equation system, the goal is the construction of a block-Jacobi Steiner tree preconditioner that minimizes the overall execution time of the solver. With the tools developed so far at hand, it is possible to provide an estimated condition number for the preconditioned equation system allowing us to bound the necessary number of iterations. Combined with the BSP model the runtime of the solver can be estimated for different block-Jacobi sizes allowing us to choose the best possible variant.

Based on this preliminary considerations, we will define an algorithm to construct an optimized Steiner tree preconditioner. Therefore, we construct a laminar decomposition $H = \{\mathcal{H}_0, \ldots, \mathcal{H}_k, \ldots, \mathcal{H}_h\}$ of a graph G such that $H_T = \{\mathcal{H}_0, \ldots, \mathcal{H}_k\} \subset H$ corresponds to the interconnect hierarchy of a given compute cluster. Now let T_l be a block-Jacobi Steiner tree preconditioner based on $\{\mathcal{H}_l, \ldots, \mathcal{H}_k\}$ where $l = 0, \ldots, k$. For $l = 0$ this is just the complete Steiner tree, in case of $l = 1$ we have that T_1 is a set of $p = |\mathcal{H}_1|$ Steiner trees and so on. We now let w^S be the work time for one iteration step of the CG method and the application of the preconditioner T_l from level h up to level k. This is the static portion of our solver. For each level i, a preconditioner T_l above level k, i.e. $i = l, \ldots, k-1$, needs g_i communication time which depends on the number of senders $|\mathcal{H}_{i+1}|$ and receivers $|\mathcal{H}_i|$. The parallel working time for the application of the preconditioner at level i is neglectable compared to the communication cost. This is due to the fact that the number of operations per processing unit is only as large as the number of received messages. The total time for a solver with preconditioner T_i, $i = 0, \ldots, k$ is then just $t_{\text{total}}(T_i) = n(T_i) \cdot (w^S + \sum_{l=k+1}^{i} g_l)$ where $n(T_i)$ is the number of operations in dependence of the preconditioner T_i. From equation (3.3), we have that

$$n(T_i) \leq \frac{1}{2}\sqrt{\kappa(T_i^{-1}A)}\ln(\frac{2}{\epsilon}) + 1 \tag{4.17}$$

in order to reach an ϵ-approximate solution. To minimize the runtime we have to find $\min_{i=0,\ldots,k} t_{\text{total}}(T_i)$. This leads in total to Algorithm 4.1.

It should also be mentioned that architecture aware computing, i.e., aware regarding the network topology, is already an important field or research with respect to load balancing, partitioning and message passing [97, 133]. Applying these ideas to preconditioners as described above is novel to the best knowledge of the author.

Algorithmus 4.1 Construction of a hardware-aware preconditioner based on block-Jacobi Steiner trees. The cluster's interconnect hierarchy is denoted by H_T and its depth by k.

construct_preconditioner(H_T,G,ϵ):

H = construct_laminar_decomposition(H_T,G)
for $i = 0, \ldots, k-1$:
    g_i = calculate_comm_time_on_level(H_T,i)
T_k = construct_steiner_trees($\mathcal{H}_k$)
w^S = estimate_solver_worktime(T_k)
n_k = estimate_iterations(T_k,G,ϵ)
for $i = k-1, \ldots, 0$:
    T_i = extend_steiner_trees($\mathcal{H}_{i-1}$)
    n_i = estimate_iterations(T_i,G,ϵ)
for $i = 0, \ldots, k$:
    $t_i = n_i \cdot (w^S + \sum_{l=k+1}^{i} g_l)$
j = index_of_min($[t_0, \ldots, t_k]$)
return T_j

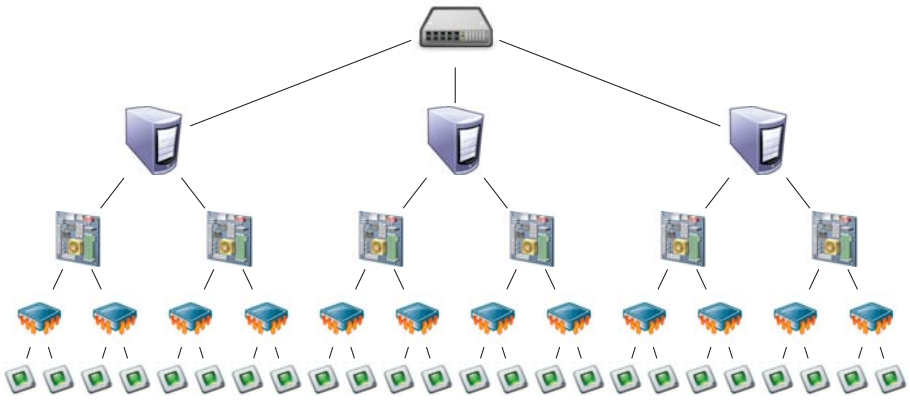


Figure 20: Interconnect hierarchy in a homogeneous compute cluster. The top of the tree represents the interconnect between compute nodes in a cluster. This interconnect is typically slower than the interconnect inside a compute node which again is superseded by the interconnect of CPUs on the same motherboard. At the bottom the fastest interconnect is given between two cores on the same CPU.

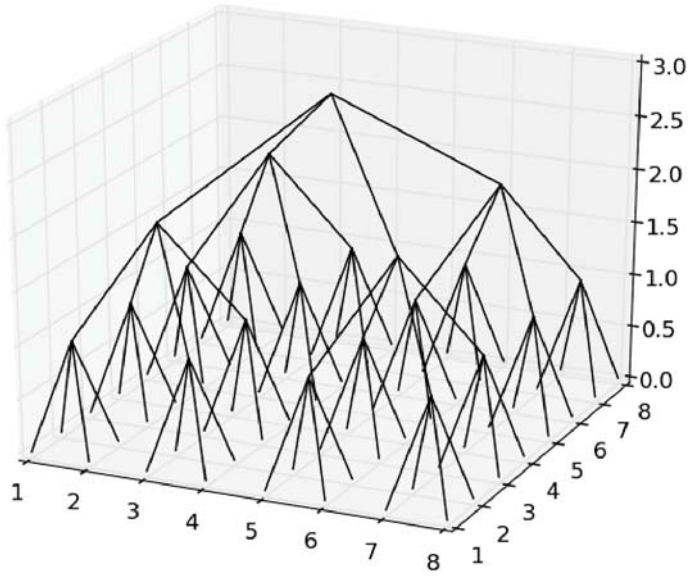


Figure 21: A laminar Steiner tree on an 8×8 mesh based on consecutively splitting one partition into four equally sized subpartitions.

4.6.3 *Evaluation of a Model Problem*

As a model problem to evaluate the techniques developed so far, we consider the matrix of a simple 2D Poisson Problem with homogeneous Dirichlet boundary on a unit square discretized by finite differences as detailed in Section 5.2. Letting h be the discretization length of a regular grid, we have after multiplication with h^2 a matrix A of structure

$$A = \begin{pmatrix} D & -I & & & \\ -I & D & -I & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & -I \\ & & & -I & D \end{pmatrix}, \quad D = \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix},$$

where I is the unity matrix. We will consider a mesh with 32×32 mesh points so that A has shape 1024×1024. Since the number of mesh points in each direction is based on the power of 2 we define a laminar decomposition by consecutively cutting each mesh partition into four equally sized parts. The laminar Steiner tree T resulting from this is illustrated in Figure 21 for an 8×8 mesh.

For larger irregular graphs the implementation of our preconditioner makes use of the Metis library [78] to generate a laminar decomposition by recursively calling the partitioner on a subpartition until it consists of only one vertex. The complexity when constructing the Steiner tree from this decomposition arises from the fact that Metis does not return which edges it cut. Therefore, determining the outflow

Preconditioner	Estimated $\kappa(A, S_*)$	Exact $\kappa(A, S_*)$	Ratio
T	109350.0	38.1	2870.1
T_1	307990.0	50.0	6159.8
T_2	42712.0	72.5	589.1

Table 10: Estimated and exact conditioner number of the preconditioned model problem. T is a laminar Steiner tree, T_1 a forest of 4 and T_2 a forest of 16 trees.

of a partition is costlier than it needs to be. After the tree is generated as a set of linked nodes and edge weights, it can be traversed in a breadth-first way to generate the corresponding matrix. A Cholesky decomposition with the obvious numbering, i.e., the inverse of a breadth-first numbering, can then be applied in order to generate a Cholesky factor with no fill-in. This Cholesky decomposition can then be applied in parallel to a vector which must be extended by zeros to the dimension of the Cholesky factors. This is necessary since the preconditioner is of larger dimension than the original matrix. More details regarding the construction and application of a Steiner tree preconditioner as wells as a thorough discussion about his parallel properties can be found in [55].

Using the Python* programming language with SciPy [75] which again uses NumPy [104], a code was developed to estimate the condition number of a Steiner tree preconditioned matrix (Theorem 34). For the case of Jacobi blocks as described in Subsection 4.6.1, with the help of maximal network flows (Theorem 42) an estimation of the condition number can be calculated.

Using SciPy's eigenvalue solver for symmetric matrices, the condition number of A was determined to be about 441. We first consider a complete Steiner tree and estimate the condition number by calculating the product of an estimation of $\sigma(T, A)$ and $\sigma(A, T)$ with the help of Theorem 34 where $\max_j \sigma(S_j^i, A_j^i)$ was calculated exactly with SciPy. For comparison, also the Schur complement S of T was determined in order to calculate the conditioner number of $S^{-1}A$ exactly. As a next step, the tree T was cut at its root resulting in a forest T_1 of four Steiner trees. Here, our block Jacobi techniques were applied to estimate an upper boundary for the conditioner number $\kappa(A, S_1)$. Again, the exact condition number $\kappa(A, S_1)$ was determined. As a last step, the roots of the four trees in T_1 were cut to obtain a forest T_2 of 16 Steiner trees and the former procedure was repeated. The results are shown in Table 10.

Unfortunately, the condition number estimations of the preconditioned systems are orders of magnitude too high, even compared to the condition number of the original matrix A. This is due to the

*http://www.python.org

exponential nature of the estimation in Theorem 34 where each λ_i of a level i star causes a factor larger than 3. A conclusion based on these vague estimations to set up a preconditioners for a specific hardware model as described in the previous subsection seems to be illusory.

Still, it is possible to use the exact calculated condition numbers to find the preconditioner, i.e. T, T_1 or T_2, which minimizes the total runtime. Using an iterative method to approximate the condition number like the power method requires us to calculate the Cholesky decomposition of T, T_1 and T_2 whereas our estimation was only based on their structure. In other words, we have to actually apply all preconditioners which comes with additional costs compared to an estimation based on their pure structure. When estimating the condition number for T_1 on the other hand, many intermediate results, i.e. $\lambda_i = \max_j \sigma(S_j^i, A_j^i)$ from Theorem 34, generated by the estimation for T_2 can be reused. Nevertheless, due to the complexity of calculating an estimation, it can be advantageous to directly calculate the condition numbers of the preconditioned system.

Assuming a supercomputer with 16 single-core CPUs where 4 CPUs reside in one compute node, we apply Algorithm 4.1 to the model problem and let $T_0 = T$. For simplification, we assume that the internode communication between compute nodes takes 0.1 seconds, i.e., $g_0 = 0.1$, communication inside one node 0.01 seconds, i.e., $g_1 = 0.01$, and applying the solver with T_2 takes $w^S = 1$ seconds. Using equation (4.17) and rounding to the previous largest integer, we have $n_0 = 89$, $n_1 = 102$ and $n_2 = 122$ needed CG iterations with preconditioners T_0, T_1 and T_2 to reach an ϵ-approximate solution where $\epsilon = 10^{-12}$. This allows us to estimate the corresponding runtimes $t_0 = 89(1\text{s} + 0.1\text{s} + 0.01\text{s}) = 107.69\text{s}$, $t_1 = 103.02\text{s}$ and $t_2 = 113.22\text{s}$. From these results we see that T_1 is the preconditioner which should be selected to minimize the overall runtime for this given problem and hardware topology. Although it was not possible to use our estimations for the conditioner number, we could still apply Steiner trees and thereby support theory to derive a flexible kind of preconditioner that can be used in real world applications.

Regarding the application of this technology in MPIOM, it turned out that the large code base of MPIOM and its inflexible parallel mechanisms prevent an implementation of a flexible Steiner tree preconditioner under a reasonable time and effort constraint. However, the gained experiences of our research could be transferred to existing libraries that provide support theory based preconditioners like TAUCS* and CMG† in order to extend them with hardware-awareness features. Before this is addressed, further development in the field of support theory should be undertaken.

*http://www.tau.ac.il/~stoledo/taucs/

†http://www.cs.cmu.edu/~jkoutis/cmg.html

While support theory provides many tools to asymptotically bound conditioner numbers in $O(f(n))$, there is still much need to investigate into better estimators that can be applied to a specific matrix and its preconditioner. The obvious parallel properties of and flexibility in designing Steiner graph based preconditioners make support theory an important field of research in light of the advent of exascale computing.

4.7 SUMMARY AND CONCLUSION

We saw that support theory provides us with a new valuable tool for the analysis of preconditioners like Steiner tree preconditioners. With the help of the theory of network flow problems, we were able to analyze block-Jacobi Steiner tree preconditioners with respect to the condition number of the preconditioned equation system. This is a new contribution to the field of support theory. Based on this result, we proposed a model for a hardware-aware preconditioner that can optimize itself depending on the interconnect topology to reduce the overall runtime of a solver. However, the implementation of this idea revealed that the estimation of the condition number is too high and therefore the optimization of a block-Jacobi Steiner tree preconditioner based on this estimation is not feasible. Still, our methodology bears many chances to obtain new preconditioners that are highly parallel and efficient.

With regard to our original problem, the barotropic subsystem, an approach as described here could be greatly beneficial. In Chapter 3, we used preconditioners in the way that each processing unit, i.e., one core, possesses one Jacobi block to allow for parallel treatment. As the number of cores rises, the quality of the preconditioner decreases and the number of iteration as well as the runtime increases. A Steiner tree on the other hand is parallel by definition and Jacobi blocks *can* but *do not need* to be used for possibly further optimization regarding the solver's runtime. The difficulty, as we have seen, is the possibly expensive estimation of $\sigma(S', G)$ to judge the quality of S'. Further research is necessary to address this problem.

In the following chapter we will address the challenges of utilizing new hardware technology for preconditioning. Besides advancements in the theory of preconditioning also the efficient implementation of a preconditioner on a target hardware as well as the hardware's capabilities are crucial for the execution time of a solver with a preconditioner.

5 FPGA BASED PRECONDITIONERS

In this chapter, we will approach the challenge of preconditioning from a more technical direction. It it obvious that beside the mathematical properties of an algorithm, its implementation on hardware as well as the hardware's capabilities determine in large parts the runtime of the algorithm. Therefore, we will now concentrate on a special technique for processing units that is different from an ordinary CPU and show how this technique can be utilized for preconditioning. Our main focus lies hereby on evincing ways of utilizing this technique from the view of a numerical mathematician which is novel in this field of research.

5.1 INTRODUCTION

One cannot emphasize too much the importance of numerical methods to solve socially relevant problems. Solving such increasingly complex problems requires much computational power in terms of speed and parallelism. Traditionally, these requirements are met with even larger clusters of commodity hardware based on x86 CPU design. Disadvantages include high energy consumption that are about to become the most important cost factor for computer centers, not even considering the environmental implications. Additionally, most scientific software does not scale linearly with the number of processors, resulting in a decrease of efficiency with an increase in count of CPUs.

Nowadays, scientists reconsider the multi-purpose approach of a CPU, meaning they become aware of the fact that a CPU is falling behind other technologies when it comes to a special niche of applications. This fact leads to special-purpose hardware, of which the most lately renowned are GPUs [100] and Cell processors [77]. GPUs are built to offer a high degree of parallelism and fixed function units that perform special tasks, often related to 3D calculations, with high efficiency. Today, the usage of GPUs as accelerators for certain parts of numerical programs is an overly accepted method to speed up execution, e.g., as preconditioners [6] or even entire solvers [62].

Another special-purpose hardware approach is to let the programmer build own parallel function units according to the special needs of an arbitrary application from any domain. Technology providing this is termed *reconfigurable hardware* and has gained more attention over the last years, but has not been considered the same breakthrough as GPU-based accelerators in scientific computing, yet. The properties of reconfigurable hardware like Field-Programmable Gate Arrays (FP-

GAs) are intriguing as they can be configured to adopt any arbitrary circuit design. Hence, FPGAs leave it up to the programmer to decide which and how many special-purpose function units are needed. A well-known example of the acceleration possibilities resulting from this is the Smith-Waterman algorithm that performs local sequence alignment of proteins in biotechnology. The speedup of Smith-Waterman on an FPGA [129] is up to 100x compared to a CPU implementation because of its special needs, e.g., highly parallel custom pipelines, that a generic CPU does not satisfy sufficiently.

The main reason for the hesitation of scientists to adapt to this technology is the challenge of implementing an algorithm. Commonly, an algorithm is implemented in a comparatively low-level description language like Verilog and VHDL. Synthesis tools further process this description of the algorithm to finally configure the FPGA. These languages follow a different programming paradigm than imperative languages like C, namely implicit parallelism and explicit sequentiality. Hence, programming hardware is error-prone, time-consuming and not feasible for most engineers and mathematicians. Recent advancements however allow implementing an algorithm by means of higher-level programming paradigms based on C, C++ or Java which are converted to hardware descriptions that can be synthesized by proprietary vendor tools afterwards.

We therefore study the applicability of this high-level language approach to FPGA programming and the interplay of CPU and FPGA for numerical applications from a mathematician's point of view. As exemplary application, we implement a preconditioner on an FPGA by using a high-level C based language. Section 5.2 provides the mathematical background for our model problem and also presents the rationale for a symmetric successive-over-relaxation preconditioner with red-black ordering. After a quick overview of the state of the art in C-based hardware development in Section 5.4, we present our implementation and benchmarking results of the hardware-assisted preconditioned CG algorithm. Section 5.5 summarizes our work and points out future perspectives.

Parts of this work arose in the context of the diploma thesis of Schmidtobreick [120].

5.2 NUMERICAL BACKGROUND

The maybe most well-known problem in numerical mathematics is to solve the Poisson equation that occurs in electrostatics and mechanical engineering. The Poisson equation also has many characteristics of the barotropic subsystem in Subsection 2.3.2 when solved for η and can therefore be seen as a simplified variant of it. This makes it a perfect first candidate to be approached with a new technology.

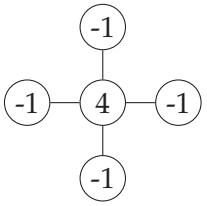


Figure 22: Five-point stencil of the Poisson problem (5.1) discretized by finite differences on an equidistant grid.

Let $\Omega \subset \mathbb{R}^2$ be an open and bounded domain and let $f : \Omega \to \mathbb{R}$, $f \in C(\Omega)$ be a given function. A function $u : \bar{\Omega} \to \mathbb{R}$, $u \in C^2(\Omega) \cap C(\bar{\Omega})$ is to be found that satisfies

$$-\Delta u = f \quad \text{in } \Omega, \tag{5.1}$$

where $\Delta := \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$. We further demand homogeneous Dirichlet boundary conditions $u = 0$ on $\partial\Omega$ and set $\Omega = (0,1)^2$ for simplicity. We discretize our domain Ω by an equidistant grid with parameter h

$$\Omega_h = \{(x,y) \in \Omega \mid x = k \cdot h, y = l \cdot h, (k,l) \in \mathbb{Z}^2\},$$

and approximate $-\Delta$ by means of finite differences

$$-\Delta u = \frac{-u_{j+e_1} - u_{j-e_1} + 4u_j - u_{j+e_2} - u_{j-e_2}}{h^2} + O(h^2), \tag{5.2}$$

where $u_{j+e_i} := u(x_j + he_i)$ and e_i denotes the i^{th} unit vector. We first number the grid points in a lexicographical way, i.e., starting at one corner of the grid and numbering the nodes consecutively. Then we multiply equation (5.2) with h^2 and obtain a matrix A_h with block structure

$$A_h = \begin{pmatrix} T & -I & & & \\ -I & T & -I & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & -I \\ & & & -I & T \end{pmatrix}, \quad T = \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix}, \tag{5.3}$$

where I is the unity matrix. The dimension of A_h depends on the number of grid points. The corresponding right-hand side that forms our equation system is $b_h(x_j) = h^2 \cdot f(x_j)$. As a result of the sparsity pattern in (5.3) we can express A_h as the well-known five-point stencil that is illustrated in Figure 22. Additionally, A_h has the advantageous properties that it is symmetric and positive definite. For solving this kind of linear system, the CG method, as introduced in Subsection

3.2.1, is the best known iterative technique [119]. To improve the convergence of the CG method we opted for a preconditioner.

5.2.1 *The Red-Black Symmetric Successive Over-Relaxation Preconditioner*

The preconditioner based on the Symmetric Successive Over-Relaxation (SSOR) method, as introduced in Subsection 3.3.1, is defined as

$$M^{-1} = \omega(2-\omega)(D+\omega L^T)^{-1}D(D+\omega L)^{-1}, \tag{5.4}$$

where D is the diagonal of A, L its strict lower part, $U = L^T$ its strict upper part and ω a relaxation parameter with $\omega \in (0,2)$. Given the sparsity pattern of our matrix A_h, this can be easily translated to a stencil formulation. The drawback of the scheme is that the left-hand side of (5.4) enforces the calculation of $x^{(k+1)}$ by a serial forward substitution. Using a red-black ordering of the unknowns remedies this drawback so that unknowns with the same color are decoupled from each other as illustrated in Figure 23. This ordering allows the parallel calculation of unknowns with the same color, thus making it a perfect candidate for execution on a highly parallel system like an FPGA. In the case of a red-black ordering, the best relaxation parameter ω is known to be 1, which renders the SSOR a symmetric Gauss-Seidel method [3]. Equation (5.4) can then be simplified to gain Algorithm 5.1.

5.3 RECONFIGURABLE COMPUTING

The first description of a reconfigurable processing unit for computation appeared as early as 1960 in the landmark paper of Estrin [45] wherein he sketches a "fixed plus variable structure computer". Based on the problem of computational polynomial evaluation, he demonstrated that depending on the context of this computation the

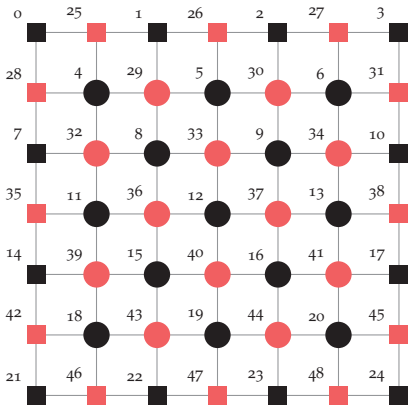


Figure 23: Example of a red-black ordering. The rectangles denote boundary values of 0.

Algorithmus 5.1 Red-black symmetric Gauss-Seidel (SSOR with $\omega = 1$) that is applied as preconditioner in the CG method.

1: **for all** z_i in red points **do**
2: $\quad z_i = (r_i + (r_{j+e_1} + r_{j-e_1} + r_{j+e_2} + r_{j-e_2})/4)/4$
3: **end for**
4: **for all** z_i in black points **do**
5: $\quad z_i = r_i + (z_{j+e_1} + z_{j-e_1} + z_{j+e_2} + z_{j-e_2})/4$
6: **end for**

suitability of a processing unit can vary. For instance, a processing unit that performs a polynomial evaluation in parallel can be advantageous over a processor that runs many sequential polynomial evaluations in parallel or vice versa. Subsequently Estrin proclaimed that only a computer with fixed elementary structures that can be variably connected would allow for the flexibility to adopt the most efficient configuration for a certain computational task and thereby the idea of *reconfigurable computing* was born.

Almost three decades later, in the late 80s, with the SPLASH 1 developed by IDA Supercomputing Research Center in 1989 the first reconfigurable system became available. Shortly after in 1991, the first available commercial system with FPGA technology was the ALGOTRONIX CHS2X4 [53].

An FPGA basically consists of three different programmable building blocks: logic blocks, routing and I/O blocks. The programmable logic blocks are used to define basic logical operations, e.g., a logical AND operation, given a number of one-digit binary operands, i.e., 0 or 1, to produce a one-digit binary output. The interconnection in between a set of logical building blocks with the help of the programmable routing yields more complex functional units which are then combined to lead eventually to the actual program, or more precisely, the design of the FPGA. The input and output of this program to the system outside of the FPGA is accomplished via the programmable I/O blocks. The actual realization in hardware of these three building blocks varies whereas *look-up tables* (LUTs), that produce the output value by taking the input as an index in a modifiable truth table, are commonly used for logical blocks. More background information regarding the FPGA hardware can be found in [53]. The most distinguishing properties of an FPGA besides the reconfigurability is its vast amount of parallelism compared to a CPU that has a certain number of cores and a fixed logic for fine-grained parallelism, e.g., instruction level parallelism, vector units. The clock frequency of a current FPGA is about one order of magnitude lower than that of a current CPU, reaching only a few hundred MHz instead of a few GHz. Also the FPGA's flexibility comes with the cost of a large overhead so that an FPGA with the same size and manufacturing density as a CPU leaves about 10-100 times less logic available to the user than the

CPU possesses. On the other hand, an FPGA is more energy efficient compared to CPUs [106] which is an important feature regarding the rise of green HPC.

Since programming an FPGA basically means defining an appropriate circuit it is by no means similar to programming a CPU. A CPU or more specific one core has a set of predefined operations which a programmer calls one after another, i.e., sequentially, albeit certain operations can be of parallel nature like vector operations. Therefore, a CPU exhibits explicit parallelism which is manifested in most programming languages targeted for CPUs. On the other hand, since an FPGA resembles an electrical circuit all operations are done in parallel per se so that sequentiality needs to be expressed explicitly. For this reason an FPGA is natively programmed with a hardware description language that directly reflects the interconnection of logical blocks which defines the FPGA design. Thus, programming in a hardware description language is not similar to programming in a conventional programming language and is considered rather complex and complicated. Therefore, a lot of effort has been made to find easier approaches to FPGA programming.

5.4 C-BASED FPGA PROGRAMMING

Due to the qualification of FPGAs as accelerators, already a multitude of floating-point algorithms [65] and numerical solvers [88] have been ported onto reconfigurable hardware. Since creating FPGA designs is a very tedious task as mentioned before, intensive research has gone into hiding the technical low-level details of implementation. A suitable approach for instance is to provide a toolbox of elementary operations on an FPGA as a library that can be accessed by a high-level language [30, 36]. Although using such a library is fairly easy and requires no deep knowledge about FPGA programming, the application is limited to the operations provided by the library. Another more flexible approach is to let the programmer design an algorithm in a high-level language and to convert it into a synthesizable Hardware Description Language (HDL) like *Very High Speed Integrated Circuit Hardware Description Language* (VHDL) or Verilog. Tools provided by FPGA vendors then process this HDL code to configure the hardware according to the design. Figure 24 sketches this process. In this work, we investigate how the latter approach, namely by using the toolchain of Impulse CoDeveloper Version 3.6, performs in solving our model problem with the help of an FPGA accelerator.

On the hardware side we use the Accelium Coprocessor System (AC2030) as illustrated in Figure 25. This is a product of DRC Computer Corporation consisting of a quad-core AMD Opteron processor 2350 and a Reconfigurable Processing Unit (RPU) attached via HyperTransport at 400 MHz. It is placed on a free Opteron socket and

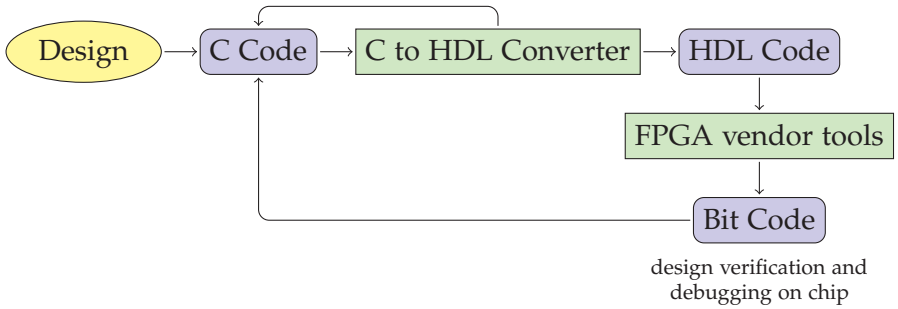


Figure 24: Hardware design workflow of an application design in C code that is converted to VHDL which is then used to configure an FPGA.

contains a Xilinx Virtex-5 LX 330 FPGA. The RPU holds its own Reduced Latency Dynamic RAM (RLDRAM) with a size of up to 512 MB. Detailed specifications can be found in [41].

5.4.1 *Impulse C*

As a high-level language, Impulse C uses the syntax of C, but instead of the C-typical procedural paradigm it employs the *communicating sequential processes paradigm* (CSP). This results in concurrently running software and hardware processes which talk to each other over streams or shared memory. While this paradigm only requires setting up a stream for input and output data on the host side, on the accelerator side it demands pipeline-based processing of the data such that one stage of the pipeline can only be executed after another. Accordingly, random data access is not possible. Streams can transport integer values, fixed-point data or floating-point numbers, each of arbitrary size. The FPGA-side memory is used by software processes as well as by the hardware processes on the FPGA to exchange data.

Source code in Impulse C needs to follow a strict scheme. In a source file for programming the hardware, the programmer declares functions that are then executed as *hardware processes*. The analogue applies to a software source file where one defines functions that become *software processes*. In a configuration function, inside the hardware source file, all processes are setup to use the formerly defined functions and to communicate by virtue of signals, streams and shared memory. The `main` function resides in the software file and is mostly intended to initialize the architecture with `co_initialize` that calls the configuration function and to start the software and hardware processes with `co_execute`. According to this scheme, the actual algorithm is defined by the interaction between hardware and software processes. Inside a hardware process certain restrictions regarding the available C language constructions apply. Thus, no recursions are allowed and function calls are only allowed to Impulse C API functions or special primitive functions that are indicated by a special pragma

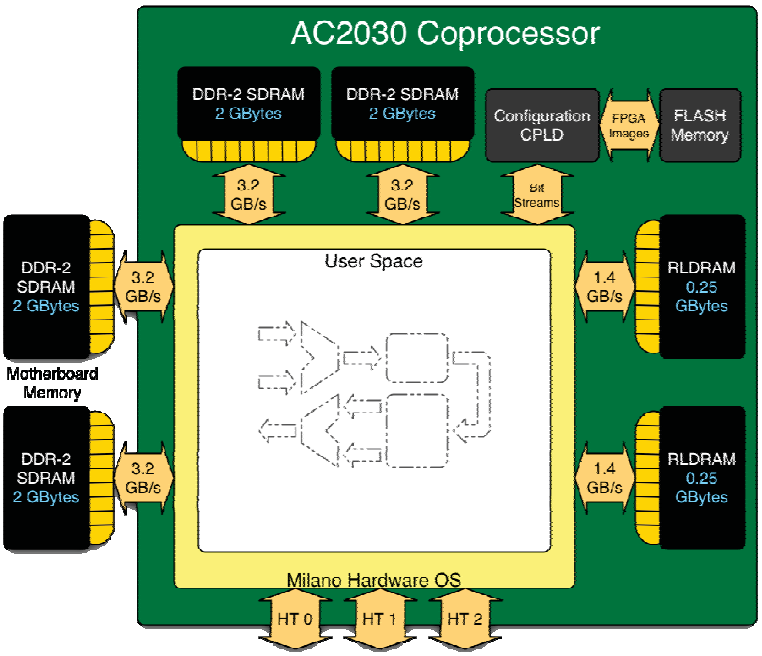


Figure 25: The FPGA AC2030 Coprocessor in the DRC system [41]. The configuration Complex Programmable Logic Device (CPLD) configures the FPGA with a user-defined FPGA design which is depicted as a flowchart.

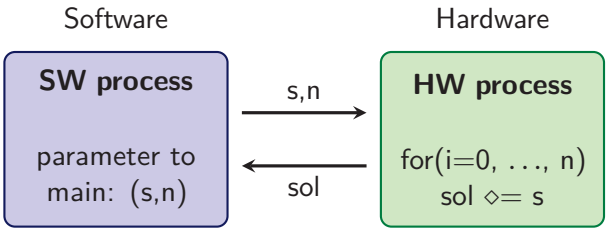


Figure 26: Repeatedly executing an elementary operation $\diamond \in \{+, *, /\}$.

and need to obey strong restrictions, e.g., only `void`, `int` and `float` return parameters. Additionally all pointers need to be resolvable at compile time, i.e., no dynamic memory allocation is possible and `struct` as well as some special conditional statements, e.g., `switch`, are only supported to some extent. For further details the reader is kindly referred to the Impulse C manual [131].

In order to find out about the potential of our FPGA, we implemented benchmarks of elementary mathematical operations. In the first benchmark, a software process sends a single-precision floating-point number s and an amount n over the stream interface to the hardware process, which in return applies n times a given operation $(+, \cdot, /)$ to s. The result is then communicated back to the software process over another stream as illustrated in Figure 26. We use this setup to find out the time a single mathematical operation on the FPGA needs. Since we can only measure the time between sending

operations	time per add	time per mult	time per div
100	1.543333	1.410000	1.693333
10,000	0.064400	0.063500	0.300633
1,000,000	0.049885	0.052089	0.288621
100,000,000	0.049742	0.049742	0.288497

Table 11: Timing results of an n times performed operation on a floating point number in microseconds.

s,n and receiving the solution, besides the runtime needed for the n operations, the measured runtime also includes communication time. Consequently, the hereof calculated time for a single operation includes $1/n$ the runtime of two communications. By increasing n we can asymptotically eliminate the communication time, as shown in Table 11. From our results, we can see that roughly after one million operations the portion of communication time vanishes.

Analyzing the result and considering that the FPGA is running at a clock rate of 100 MHz (10 ns clock period), we can conclude that a floating-point addition or multiplication in a `for` loop takes at least 4 clock cycles (0.05 µs = 50 ns) and in the case of a division at least 28 clock cycles. For the remaining worst-case estimations, we will therefore round to 5 cycles for an addition and to 29 cycles for a division. We performed the same tests for integers and even for an empty loop body. The very surprising result was that both an integer addition in a `for` loop and an empty `for` loop need 2 clock cycles. The reason for this is that the configured circuit for this algorithm on the FPGA concurrently executes the addition while also performing the counter increment and the evaluation of the conditional jump in the `for` loop. This kind of instruction level parallelism is uncommon to a standard CPU where the overhead of a loop operation would be clearly visible.

As our first benchmark executed sequentially one operation after another due to data dependencies, we did not exploit the possibility of a nearly arbitrary number of parallel processes in hardware, which is only restricted by the physical size of the FPGA. Hence, we implemented a second benchmark that executes a floating-point addition n times on 1, 4 and 8 hardware processes. Ideally, on k hardware processes the runtime should decrease to $1/k^{\text{th}}$ the time of a single process. Table 11 shows the results of these tests. Looking at the last row of this table, we can see that the ideal speedup is achieved for 4 and 8 hardware processes compared to 1 process. The parallelization was done manually by copying the function of the hardware process in order to get up to 8 hardware processes since there is no automatism provided for this kind of parallelization. Then we had to consistently

operations	1 HW proc	4 HW proc	Ratio	8 HW proc	Ratio
100	1.543333	2.056667	1.33	2.812500	1.82
10,000	0.064400	0.026533	2.43	0.026700	2.41
1,000,000	0.049885	0.012576	3.97	0.006407	7.79
100,000,000	0.049742	0.012437	4.00	0.006219	8.00

Table 12: Timing results of an n times performed add operation on a floating point number in microseconds executed simultaneously by 1, 4 and 8 hardware processes. The ratio compares the previous entry to the runtime of 1 hardware process.

wire one software process to many hardware processes via one input stream and one output stream per process which is error-prone.

5.4.2 *Implementation of Symmetric Gauss-Seidel Preconditioner*

We first implemented the preconditioned CG Algorithm 3.2 from Chapter 3 on the CPU in plain C. This CG implementation calls either a software preconditioner on the CPU or an FPGA-implemented Symmetric Gauss-Seidel (SGS) preconditioner as in Algorithm 5.1. We decided to not implement the entire CG method inside an Impulse C software CPU-side process because this covers the use case that an accelerated preconditioner needs to be integrated into existing software. Therefore, the Impulse C software process acts only as a proxy between the CG implementation and the Impulse C hardware process with the SGS preconditioner. Furthermore, as a result of the red-black scheme that exposes few data dependencies, the preconditioner is a suitable candidate for automatic parallel execution. Figure 27 illustrates the workflow of our program. Before the first iteration, the software process transfers the values of the five-point stencil to the hardware process where they are stored as coefficients in registers. This allows us to use the same code for other five-point stencil coefficients. In each iteration, CG passes the residual to a preconditioner function that invokes the entire architecture with `co_execute`. The software process then copies the residual to the FPGA memory and sends a signal to the hardware process after completion because there is no automatism until now to map the required arbitrary memory access to streams. Hereon, the FPGA performs the SGS operations on the residual which is extended by a boundary halo and therefore avoids unnecessary switch statements to distinguish between inner and boundary points. The treated elements are directly streamed back to the software process. After the last element has been transferred, the software process terminates and CG continues.

Employing the concept of the boundary halo can already be considered a minor aspect of hardware-awareness as it both saves hardware

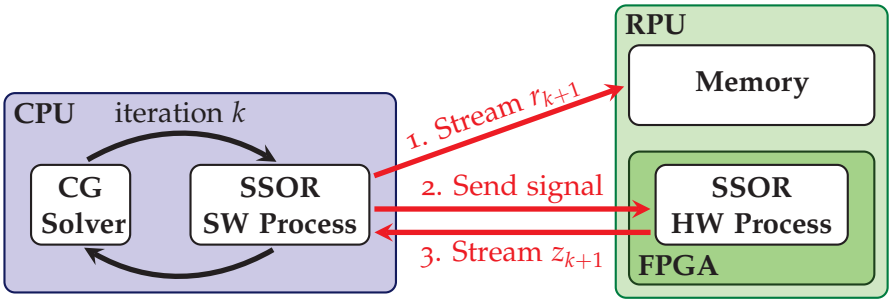


Figure 27: Flow chart of a CG solver which calls a preconditioner that is implemented on an FPGA. The current iteration number is denoted by k.

resources and keeps the pipeline structure simple. It should also be mentioned that we are using single-precision floating-point throughout the whole program, i.e., on the CPU and FPGA.

5.4.2.1 *A-priori Performance Estimation of the FPGA Implementation.*

We now estimate the theoretical time consumption of a straightforward implementation. The transport of one floating-point number is achieved at one clock cycle of the 400 MHz HyperTransport interconnect, i.e., $t_{\text{transport}} = 2.5\,\text{ns}$. The latency of the HyperTransport is negligible for these large amounts of data. Processing one stencil requires 5 random-access data fetches from RLDRAM, 4 additions, 1 or 2 divisions, and 1 write-back to host memory. This results in a pipeline length of $4\,\text{adds} \cdot 5\,\text{cycles/add} + 2\,\text{divs} \cdot 29\,\text{cycles/div} = 78\,\text{cycles}$ with an instruction issue rate of 1 instruction every 29 cycles due to the non-pipelined division. In return, this can hide the memory fetches, i.e., one stencil completing every $29 \cdot 10\,\text{ns} = 290\,\text{ns}$ when running at 100 MHz. The time $t_{\text{writeback}} = 2.5\,\text{ns}$ for writing back the results can also be hidden except for the very last datum for which we also have to account clearing the entire pipeline with $78 - 1$ cycles. For an $n \times n$ matrix, $t_{\text{overall}} = n^2 t_{\text{transport}} + n^2 \cdot 290\,\text{ns} + 77 \cdot 10\,\text{ns} + t_{\text{writeback}} = n^2 \cdot 2.5\,\text{ns} + n^2 \cdot 290\,\text{ns} + 770\,\text{ns} + 2.5\,\text{ns}$ approximates the execution time which is for our 4000×4000 case $t_{\text{overall}} = 4680\,\text{ms}$. Potential for optimization by Impulse C lies in pipelining the division, caching previously used data and exploiting data-level parallelism by instantiating several pipelines until the implementation becomes memory-bound.

5.4.2.2 *CPU and FPGA Performance Measurements.*

To give a fair comparison of the performance of the preconditioner on the FPGA, we implemented the same algorithm as a software function with red-black ordering and sequential processing on a single core. The application was compiled with gcc and flag `O2`. We then solely measured the runtime of the software and hardware preconditioner.

Refinement	SGS on FPGA	SGS on CPU	Ratio
500×500	0.974298	0.003518	276.95
1000×1000	3.880566	0.013678	283.71
2000×2000	15.60444	0.055825	279.52
4000×4000	61.91578	0.257733	240.23

Table 13: Runtime in seconds of a software and FPGA-based SGS preconditioner for different refinement levels. The FPGA-supported preconditioner performs 13x below our expectation.

Refinement	SGS on FPGA	Expected time	Ratio
500×500	0.974298	0.073	13.32
1000×1000	3.880566	0.292	13.26
2000×2000	15.60444	1.170	13.34
4000×4000	61.91578	4.680	13.23

Table 14: Runtime in seconds of a software and FPGA-based SGS preconditioner for different refinement levels. The FPGA-supported preconditioner performs 13 times below our expectation.

Table 13 shows the results of these benchmarks for different refinement levels h.

The maximum to-be-expected throughput of a naïve implementation is roughly 20 times less than the processing time on the CPU with a 20 times higher clock rate; though still without exploiting any additional parallelism apart from pipelining and without any caching.

5.4.2.3 *FPGA Performance Analysis.*

The poor real performance of the FPGA in comparison to the a-priori estimation and to the CPU has several causes. First, we transfer the residual r_{k+1} to the RPU's memory before the actual calculation starts. However, from the above formula we can see that this transfer only accounts for $2.5\,\text{ns}/290\,\text{ns} = 0.0086$ per element. Secondly, the FPGA is performing with $1/20$ of the CPU's clock rate. Thirdly, it needs to separately load each stencil operand from RPU memory into FPGA registers without help of a deep memory hierarchy in contrast to a CPU that employs caches. Fourth, the number of states in the state machine of the stencil is in the quite high range of 40 to 80, with each state lasting between 1 (loop increment) and 29 cycles (division). This high number of executed states potentially indicates that all the operations are only executed one after the other, leaving much room for optimization. So even without caching, the resulting hardware design is rather compute-bound than memory-bound because 29 cycles for the division would leave enough room to read 5 stencil data

Resource	Consumption	Ratio
DSP48Es	47/192	24.48%
RAMB36SDP_EXPs	54/288	18.75%
Slice Registers	41574/207360	20.05%
LUTs	43767/207360	21.11%
LUT-Flip Flop pairs	56843/207360	27.00%

Table 15: Resource consumption of the SGS method on Virtex-5 LX330.

and write the result. Fifth, computations are not arranged in a tree-based, pipeline-suitable order. Our efforts to allow easy parallelization with the help of the red-black ordering scheme did not automatically yield any notable parallelization because only one pipeline was created automatically as the resource consumption report of the place&route steps of the FPGA vendor toolchain indicates in Table 15. A simple test application revealed that the management infrastructure of the RPU system already accounts for more than 35K of the slice registers and more than 45K slice LUT-flip flop pairs. Seemingly, the Impulse Compiler did not exploit the data independence due to the red-black ordering implicitly, and explicit loop-unrolling via a pragma proved not to be possible with dynamic loop boundaries.

As manual parallelization like in Subsection 5.4.1 proved too error-prone, we did not further investigate in splitting the preconditioner into several hardware processes that concurrently work on distinct data sets. Adding to this is the fact that the DRC platform support package of our target platform does only provide access to two RL-DRAM interfaces, thus only allowing concurrent memory access of two hardware processes.

Table 13 also shows remarkably well, that our implementation on the FGPA scaled better with increasing refinement levels than the CPU implementation. This seems to be due to the streaming model and is of special interest with regard to the ongoing increase in FPGA bandwidth and increasing FPGA frequencies.

5.5 SUMMARY AND CONCLUSION

We showed that with the help of Impulse CoDeveloper it is possible to implement a preconditioner on an FPGA as part of a CG solver on a CPU for a Laplace model problem. This did not require any deeper knowledge of reconfigurable hardware and an HDL. The implementation of our model problem was accomplished in a reasonable amount of time, incomparably shorter than it would have taken us using an HDL. The actual performance results (approximately 13.4 times below what could be expected) are not good enough to consider

it yet a valid approach to design an accelerator for numerical applications. Until now, hardware-awareness is crucial when targeting FPGA technology, such as exploiting bandwidth and the available FPGA resources while also creating efficient pipeline structures. Hence, only hardware designers rather than high-level programmers can access the full potential of FPGAs. Nevertheless, the high-level language to HDL converter technology shows great potential because hardware designers can start with C-like descriptions of their algorithms and use the generated hardware description to improve on that. This can significantly reduce time to market for high-performance FPGA designs. However, it should be kept in mind that this technology is still an ongoing field of research, with numerous investigations towards streaming and memory access optimizations currently being undertaken and already today it allows non-hardware developers to easily develop applications for FPGAs. Moreover, the development of the FPGA technology itself is gathering pace and we are looking forward to higher clock rates, faster interconnects and other improvements yet to come, especially since frequency in general-purpose processors has stopped to rise for the sake of more cores on a single die. We are convinced that reconfigurable computing, made accessible to scientists in the field of HPC by high-level languages, has a bright future as an accelerator technology or even processing technology.

6

SUMMARY AND OUTLOOK

This thesis started with the motivation of this work by emphasizing the importance of reliable climate simulations which are based on highly complex systems of coupled numerical models running on high-performance computers. The ScalES project addressed the problem of porting legacy components of an earth-system-model like the ocean/sea-ice model MPIOM to modern, highly-parallel supercomputers. In this work we presented amongst other subjects the improvements achieved by the author in the ScalES project regarding parallel solvers and preconditioners for the barotropic subsystem of MPIOM.

We begun by deriving the ocean primitive equations which are used as fundamental model in MPIOM. This was followed by the decomposition of the velocity into baroclinic and barotropic velocities in order to obtain the barotropic subsystem. The discretization of the barotropic subsystem by finite differences resulted in a system of linear equations. With the determined barotropic and baroclinic velocities, the actual velocity field can be reconstructed. Figure 28 visualizes the lateral velocity fields in MPIOM at a layer of 17 m depth.

To efficiently solve the barotropic equation system we analyzed the scalability and performance of a set of newly implemented solvers and preconditioners and compared them to the traditionally used SOR method. This analysis encompassed the CG method and the Chebyshev which can be combined with a Jacobi, SSOR, ILU, ICC(p) or MICC(p) block-Jacobi based preconditioners as well as an additive Schwarz method and a multi-precision iterative refinement approach. We saw that the CG and Chebyshev method together with an ICC(p) or MICC(p) preconditioner highly outperforms SOR on 64×32 cores due to the reduced number of iterations resulting in less communication overhead. Presumably, the speed-up will even be higher on larger setups due to this fact. This assumption should be examined in further benchmarks. Regarding the ScalES-Lib development, an implementation of the Steiner graph preconditioners could be promising albeit tedious given the various restrictions of the Fortran 95 language.

With the objective to better understand the impact of a preconditioner on the condition number and consequently on the number of iterations of the CG and Chebyshev method, we provided an introduction the support theory. Using this theory we analyzed $\sigma(S, A)$, where S is a Steiner graph preconditioner and A is a system of linear equations, in order to obtain an estimation for $\kappa(S, A)$. With the help

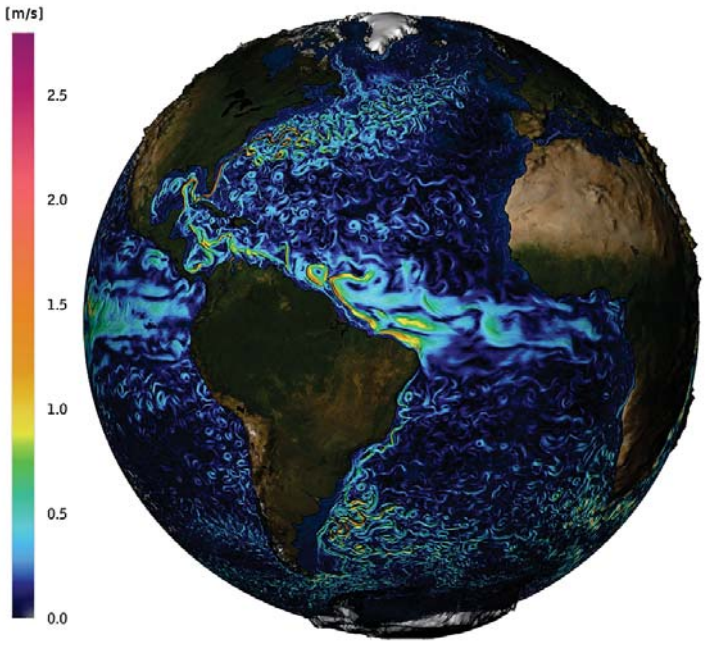


Figure 28: The velocity in m/s at 17 m depth simulated with MPIOM. This picture is courtesy of DKRZ.

of the theory of network flow problems we could extend our analysis to the case where S is a block-Jacobi Steiner graph preconditioner which is a new addition to this field. The feasibility of our approach to estimate $\kappa(S, A)$ was shown on a model problem. The obtained estimations highly overestimated the actual condition number but the applicability was shown and therefore the ground was laid for further extensions to support theory that may result in better estimations.

Based on the idea of estimating the condition number of a preconditioned system we showed how Steiner tree preconditioners could be used as hardware-aware preconditioners. We proposed a way to exploit the similarity between a Steiner tree structure and the network topology of a homogeneous compute cluster with the help of a BSP model and support theory. The main idea was to determine the optimal sizes of the Jacobi blocks for the Steiner tree preconditioner to reduce the overall runtime of the solver. In order to do this, we applied our new techniques to estimate the condition number of the preconditioned system. These estimations proved to be too inexact to be used in our model for a hardware-aware preconditioner. Still, with the help of exact condition numbers the application of our method could be shown on a model problem. With the advent of increasingly larger, highly parallel clusters that exhibit a deep network hierarchy, the need arises for hardware-aware preconditioners that exploit fast interconnections while shunning the slow ones. This is an important field of research that should definitely be pushed further.

With regard to hardware acceleration, we also examined the utilization of reconfigurable computing for preconditioners. The focus was laid on an approach to FPGA programming that allows scientists, that are not familiar with hardware languages, to use a high-level language which they are accustomed to, like C. The Impulse CoDeveloper converter technology was then used to translate C to the hardware language VHDL which then configured an FPGA using Xilinx. We showed the feasibility of this approach by implementing a CG solver that is accelerated by a red/black SSOR preconditioner on an FPGA. Our benchmarks comparing this implementation with a SSOR preconditioner on a CPU and a performance estimation based on the FPGA's hardware capabilities revealed some shortcomings of the C to VHDL converter approach. The parallelism of our algorithm was not exploited and a large overhead compared to the potential performance of our FPGA was observed. Still, the possibility to use reconfigurable technology without actual hardware knowledge promises to be highly important for numerical applications in the future given that this converter technology as well as the performance of FPGAs advances rapidly.

With high resolution numerical simulations based on increasingly complex models, parallel solvers and preconditioners will play a more and more important role considering the fact that future exascale computers will presumably not be reached by increasing processor speed but by massively increased parallelism only.

APPENDIX

A

SCALES PROJECT

Besides the solver component as detailed in Chapter 3, two other valuable tools for climate models were developed within the scope of the ScalES project by our partners. Firstly, a library for universal data transposition called UniTrans which is introduced in the following section. Secondly, an efficient software component for data partitioning in climate models to allow dynamic load balancing was developed. A short introduction hereto is given in Section A.2. The solver and data partitioning components together form the ScalES-Lib which can be used together with UniTrans.

A.1 TRANSPOSITION AND EXCHANGE OF DATA WITH UNITRANS

In the atmosphere model ECHAM, global data are continuously restructured and repartitioned among processing units with regard to the respective sub-problem. This is carried out in order to improve cache utilization and parallelism by data structures adjusted to the sub-problem's solving algorithm. Another reason is making required data, for solving the sub-problem, locally available if possible or reducing global communication to a minimum in order to speed up computation. This is illustrated in Figure 29 showing two different decompositions. There also exist sub-problems, e.g., horizontal diffusion, that are not even solved in grid-point space but in spectral space requiring another special decomposition. For the ocean model, where iterative methods are applied in certain sub-problems, fast exchange of boundary data is highly relevant.

In all these cases the performance of the model software depends on the ability to efficiently redistribute data on a global scale. Imple-

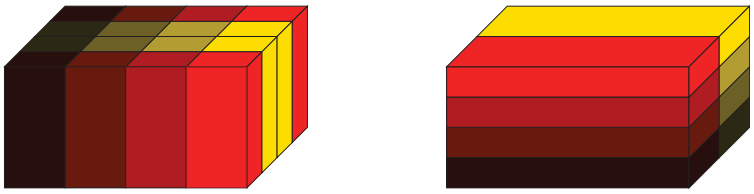


Figure 29: Two ECHAM decompositions: Left for vertically-only coupled physical effects like precipitation and right for horizontal transport. In the left decomposition each column is completely decoupled from others whereas in the right decomposition complete decoupling has been given up in favor of higher parallelism, i.e., processing units need to communicate boundary points to their horizontal neighbors.

menting such functionality is a complex task for many reasons. The implementation of an efficient redistribution scheme requires a deep understanding of the Message Passing Interface (MPI) and hardware interconnect. Additionally, the repetitive task of writing new and maintaining existing code for redistributing data between different sub-problems can be quite error-prone and time-consuming.

There are high level libraries that provide means to repartition and redistribute data such as the Model Coupling Toolkit (MCT) [84] or to some extent also the PETSC library [11]. We found that none of these were flexible enough to fill our needs, especially when it comes to local data representation. There, we need high flexibility to adapt the data layout to the algorithm of the sub-model *and* the hardware capabilities. For this reason we designed and implemented a new library, named UniTrans, that provides a high productivity and flexible interface to setup fast *uni*versal data *trans*positions.

The main concept behind UniTrans is a clear separation between the logical decomposition of data and the physical data layout. The former is described by a global index field enumerating all data elements. Each decomposition is now defined by local subsets of this global index field, i.e., a local index field for each processing unit. In order to create a transposition from a source to a target decomposition UniTrans calculates a *transposition template*, including information about communication partners and the indices that are to be sent/received.

To describe the physical data layout, one needs to pass a data representation containing the types of the elements as well as a mapping between the indices and memory addresses to UniTrans. This representation is then used by UniTrans to create a *transposition plan* with the help of the transposition template. In this step UniTrans allocates and initializes auxiliary communication buffers, even creates MPI derived datatypes and sets up everything else needed for the communication process to be executed with minimal overhead.

A transposition plan can then be applied, together with the location of the source and target data, to execute the actual transposition. The internal processing of UniTrans, completely hidden from the user, is sketched in Figure 30. This encapsulation allows UniTrans to transparently adapt to the fastest communication method available on a given Computer System such as MPI optimized collectives instead of point-to-point messaging. Perhaps the most important performance enhancing feature of UniTrans is its capability to aggregate multiple transposition plans for different data sets into a single larger execution plan. Data which is exchanged over the network with the same target will be coalesced to form larger messages which greatly reduces the latency of the communication. In a typical production situation with noticeable latency the transposition from the left to the right

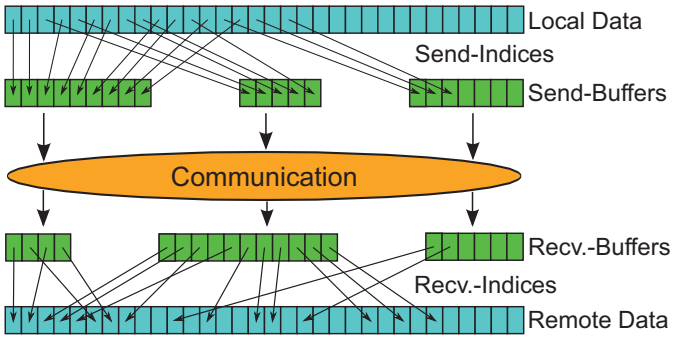


Figure 30: The picture illustrates the machinery of indexes and buffers managed by the UniTrans library to describe a transposition between to different data decompositions and representations.

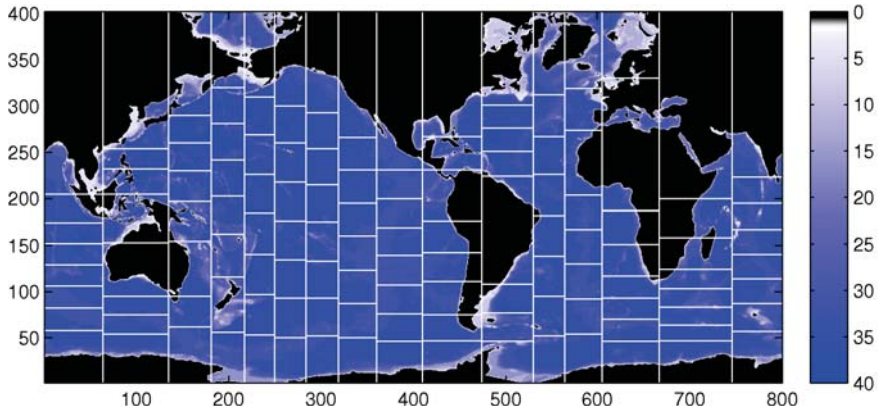


Figure 31: Hierarchical 16×8 decomposition of the land/sea-mask of MPIOM based on the number of wet points in each partition.

decomposition in Figure 29 is about 2 to 3 times faster using UniTrans than the original model implementation.

A.2 HIERARCHICAL PARTITIONER

One of the surprises one might encounter when trying to scale an established application towards higher parallelism for a given problem size is the appearance of load imbalance that was not noticeable before. This effect occurs due to the shrinking size of partitions which, in the simplest case, are blocks of a regular decomposition. Imagine for instance a decomposition into equally sized partitions of the land/sea-mask shown in Figure 31 with 16×8 or more partitions. In this case, at least one partition contains only land points, meaning that the corresponding processing unit will idle during the calculation of the ocean dynamics. Another cause for load imbalance are temporary effects on certain partitions like increased atmospheric biogeochemical reactions during dusk and dawn.

There exist advanced load balance solutions to overcome this problem like METIS [78] but for complex models the central issue is about

how to integrate these solutions into the model without imposing a radical change in data and code structures. The amount of flexibility available to a load balance scheme is further limited by the performance impact of a possibly arbitrary partitioning. In the case of MPIOM the stencil-based operations as described in Chapter 3 require access to nearby points which leads to an increased surface as one moves away from a rectilinear shape. This would increase the communication and decrease the efficiency of memory access due to overly shortened loops.

As a block decomposition with a favorable low surface to volume ratio and neighbor count we decided to use a hierarchical approach in the form of a multi-level one-dimensional decomposition [111] as shown in Figure 31. In this example the two-dimensional workload is first added along the y-axis giving a coarse one-dimensional load on the x-axes only. This is then decomposed into 16 equally loaded coarse partitions with a simple and efficient algorithm. In a second step each coarse partition is then split into 8 partitions leading to the final decomposition as illustrated.

This hierarchical decomposition restricts the number of possible neighborhood candidates and therefore allows for a simplified intersection analysis of different decompositions. Given this information, UniTrans can speed up the calculation of intersections of indices in order to create the transposition template. This is especially relevant for dynamic load balancing.

A.3 CONCLUSION AND PERSPECTIVES

UniTrans and ScalES-Lib serve mainly two purposes in the field of climate research. Firstly, by identifying reoccurring tasks in legacy climate models and providing a solution through a library we help to encapsulate technical auxiliary functions from the primary functions of the model itself. Thus, the model's code becomes clearer and better maintainable. It is our hope that a tailored library is much more likely to be accepted and used by climate scientists than any of the available generic libraries which are often cumbersome to integrate due to the special needs of legacy climate models like COSMOS.

Secondly, resolving the presented tasks by means of modern techniques and methods greatly improved the scalability and speed of COSMOS. Some of these performance gains were shown in Chapter 3. This gives space for more complex models with finer resolutions on high performance computers which leads to more accurate and reliable climate projections. We are convinced that the complexity of parallel programming will even increase on future hardware. Our libraries hide some of this complexity from climate scientists and let them concentrate on their problems at hand.

We see that COSMOS already benefits from the integration of Uni-Trans and ScalES-Lib and hope to encourage developers of other climate models to use and further extend them as well.

B

BLIZZARD CLUSTER

On the 10th of December in 2009, the new supercomputer Blizzard at the DKRZ in Hamburg was officially put into service. Its main purpose are high-resolution climate simulations for the IPCC Assessment Reports as well as the annually Conferences of the Parties (COP) where representatives of industrial nations meet to assess progress and discuss actions in dealing with climate change.

The Blizzard is a tool of paramount importance for climate researchers in order to further extend the self-proclaimed, world-wide largest climate data archive at the DKRZ. In this chapter we outline the major features of the Blizzard supercomputer [108].

B.1 COMPONENTS OF BLIZZARD

The Blizzard supercomputer was built by IBM. It consists of the following components:

- 249 IBM p575 general compute nodes
  - 32 POWER6 CPU cores at 4.7 GHz, each core with simultaneous multithreading (SMT) of 2 threads
  - 80 nodes with 128 GB memory and 169 nodes with 64 GB memory adding up to a total of 21 TB memory
- 2 IBM p575 interactive login nodes
  - 256 GB memory, connected to the outside via two 10 GB Ethernet interfaces
- 1 IBM p575 serial compute node
  - 128 GB memory, connected to the outside via two 10 GB Ethernet interfaces
  - can act as login node if a login node fails
- 12 p575 IO nodes
  - GPFS NSD servers
  - 2.5 GB/s IO bandwidth each, 30 GB/s aggregated bandwidth
- 4 p575 HPSS* mover nodes

*High Performance Storage System™

Figure 32: The high-performance supercomputer Blizzard at the DKRZ. The computes nodes are highlighted in orange, the Infiniband switches in red and the disk system in green [2]. This photo is courtesy of DKRZ.

  - provide access to HPSS via pftp
  - 2.5 GB/s IO bandwidth each

- 24 IBM DS5300 and 3 IBM DS4700 disk systems for a total of 3 petabyte of data

The compute nodes are interconnected with an 8 plane Infiniband 4x DDR Fat CLOS Tree interconnect with an aggregated bandwidth of 7.6 Terabyte/s. The storage system with the GPFS global file system is realized with a 2 plane 4 GB Fibre Channel interconnection allowing 30 GB/s aggregated bandwidth in and out over all file systems with up to 1.2 GB/s single read/write stream performance. Figure 32 shows the physical setup of Blizzard's components at the DKRZ.

B.2 IBM POWER6 COMPUTE NODES

One p575 compute node is formed by a densely packed and directly water cooled chassis of 2U height holding 16 dual core CPUs as shown in Figure 33. This form factor allows for 14 nodes per rack which leads to a total of 448 cores per rack. One p575 compute node is a 32 way cache coherent Non-Uniform Memory Architecture (ccNUMA) with a total of 64 hardware threads due to SMT technology. The POWER6 CPUs feature a high clock frequency of 4.7 GHz which was the highest in industry in 2009 [108]. Each core has a large private 64 kB L1 cache as well as a 4 MB L2 cache. Two cores share one 32 MB L3 cache with on-chip directory on a single CPU. The memory bandwidth is 128-140 GB/s as measured by STREAM Triad [94]. One core has a peak performance of 18.8 Gigaflop/s giving a total system peak performance of 158 Teraflop/s. The Vector Multimedia eXtension (VMX) of the POWER6 allows for short-vector SIMD operations with integers and 32 bit floating point numbers. With the help of the VMX units, a peak performance of 32-40 Gigaflop/s per core can be obtained for 32 bit calculations.

Figure 33: An IBM p575 general compute node. This photo is courtesy of IBM.

B.3 SOFTWARE STACK

The Blizzard supercomputer adheres to a typical IBM software stack. An AIX 6.1.5.2 kernel is building the basis of this stack. It includes the Infiniband driver, an Infiniband Userspace driver for low latency communication as well as the TCP/IP driver for GPFS 3.4.0.4. The Low-Level Application Programming Interface (LAPI) is used in version 3.1.6.1 and the Parallel Operating Environment (POE) which implements the MPI 2.0 standard in version 5.2.2.2. As compilers, the IBM XL C/C++ compiler 11.1.0.7 and the IBM XL Fortran compiler 13.1.0.7 are provided. Additionally, several parallel libraries for scientific computing are available, the IBM Parallel Engineering and Scientific Software Library (PESSL) in version 3.3.0.2 which utilizes IBM's Engineering and Scientific Subroutine Library (ESSL) in version 4.4.0.1 as well as the Mathematical Acceleration Subsystem (MASS) libraries version 4.4. As parallel job scheduling system IBM's LoadLeveler 4.1.1.3 is in use.

Adiabatic process: A process in which the change of internal energy (e.g. temperature) of a material volume is not influenced by the internal energy of its surroundings, only by its inner working (e.g. change of pressure).

Advection: The transport of a physical property or particle concentration (denoted by a scalar function ϕ) solely by the motion of a fluid (denoted by a velocity field $\mathbf{v}$). The advection term has the general form $\mathbf{v} \cdot \nabla\phi$.

Baroclinic velocity: The non-uniform part of velocity that is dependent on depth and the associated variations of density and pressure.
The baroclinic velocitiy is the velocity minus the *barotropic* velocity.

Barotropic velocity: The part of velocity that is uniform with depth and the associated pressure and density. This requirement concludes that *isobaric* and *isopycnic surfaces* are parallel.

Convection: The sum of all motions resulting in transport and mixing of physical properties of a fluid or of the particle concentration in a fluid. For a scalar property of a fluid, convection can be understood as the sum of *advection* and *diffusion*.

Diagnostic equation: Any equation governing a system with no time derivatives. Thus, such an equation defines a balance of variables in space at a moment of time.

Diagnostic variable: Any variable which is solely determined by a *diagnostic equation*.

Diffusion: With diffusion all transport processes of scalar physical properties or quantities are denoted wich cannot be exactly resolved within the model's scale. This includes amongst others molecular motions (*Brownian motion*), general unresolved mixing and turbulent motions of a fluid. The general form of the diffusion term is $\nabla \cdot (D\nabla\phi)$ with a scalar-valued function ϕ and a tensor D as parameterization of the diffusion.

In situ: The actual measure of a physical property, e.g. temperature, density, of a fluid at the measuring point.

Isobaric surface: A surface of constant pressure.

Isopycnic surface: A surface of constant density.

Meridional: Direction along a meridian or longitudinal circle, i.e., north-south direction.

Potential temperature: The temperature that a parcel of a fluid would attain if brought adiabatically (in an *adiabatic process*) and reversibly (no friction) from its initial condition to a reference pressure p_0.

Prognostic equation: Any (partial) differential equation governing a system with a time derivative of a time-dependent variable $u(\star, t)$. If the other variables in the equation and $u(\star, t_i)$ are known, the value of $u(\star, t_{i+1})$ can be determined.

Prognostic variable: Any variable with a time derivative in a *prognostic equation*.

Viscosity: Generally, description of a fluid's behaviour under the influence of physical stress. It can also be perceived as *inner friction*. On larger scale other unresolved processed like eddies can be treated as a form of viscosity.

Zonal: Direction along a latitude circle, i.e., west-east direction.

BIBLIOGRAPHY

[1] Klimaprojektionen für das 21. Jahrhundert. URL `http://www.mpimet.mpg.de/fileadmin/grafik/presse/Klimaprojektionen2006.pdf`.

[2] Supercomputing and Data at DKRZ, 2010. URL `http://www.dkrz.de/pdfs/poster/ISC10_HardwareDKRZ.pdf`.

[3] G. Alefeld. On the Convergence of the Symmetric SOR Method for Marices with Red-Black Ordering. *Numerische Mathematik*, 39:113–117, 1982. ISSN 0029-599X. doi: 10.1007/BF01399315.

[4] G. Alefeld, I. Lenhardt, and H. Obermaier. *Parallele numerische Verfahren*. Springer, Berlin, 2002. ISBN 3-540-42519-5.

[5] N. Alon, R. M. Karp, D. Peleg, and D. West. A Graph-Theoretic Game and its Application to the *k*-Server Problem. *SIAM J. Comput.*, 24:78–100, February 1995. ISSN 0097-5397. doi: 10.1137/S0097539792224474.

[6] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser. A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform. In *PDP '10: Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 583–592, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-3939-3. doi: 10.1109/PDP.2010.51.

[7] A. Arakawa and V. Lamb. Computational Design of the Basic Dynamical Processes of the UCLA General Circulation Model. *Methods in Computational Physics*, 17:173–265, 1977.

[8] R. Aris. *Vectors, Tensors, and the Basic Equations of Fluid Mechanics*. Dover books on mathematics. Dover, New York, NY, 1. publ. edition, 1989. ISBN 0-486-66110-5. Unabridged and corr. republ. of the work 1. publ. by Prentice Hall, Englewood Cliffs, NJ, in 1962.

[9] O. Axelsson and V. A. Barker. *Finite Element Solution of Boundary Value Problems : Theory and Computation*. Computer science and applied mathematics. Acad. Pr., Orlando [u.a.], 1984. ISBN 0-12-068780-1.

[10] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Accelerating Scientific Computations with Mixed Precision Algorithms. *Computer Physics*

Communications, 180(12):2526–2533, Dec 2009. ISSN 00104655. doi: 10.1016/j.cpc.2008.11.005.

[11] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. *Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries*, pages 163–202. Birkhauser Boston Inc., Cambridge, MA, USA, 1997. ISBN 0-8176-3974-8.

[12] S. Balay, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. C. Mcinnes, B. Smith, and H. Zhang. PETSc Users Manual, Sept. 2011. URL `http://www.mcs.anl.gov/petsc/petsc-as/snapshots/petsc-current/docs/manual.pdf`.

[13] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[14] G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge mathematical library. Cambridge Univ. Press, Cambridge [u.a.], 9. print. edition, 2007. ISBN 978-0-521-66396-0.

[15] M. I. Beare and D. P. Stevens. Optimisation of a Parallel Ocean General Circulation Model. *Annales Geophysicae*, 15:1369–1377, 1997. ISSN 0992-7689. doi: 10.1007/s00585-997-1369-3.

[16] B. Beckermann and A. B. J. Kuijlaars. Superlinear Convergence of Conjugate Gradients. *SIAM J. Numer. Anal*, 39:300–329, 1999.

[17] M.l Bern, J. R. Gilbert, B. Hendrickson, N. Jguyen, and S. Toledo. Support-Graph Preconditioners. *SIAM J. Matrix Anal. Appl.*, 27 (4):930–951, 2006.

[18] M. Bienkowski, M. Korzeniowski, and H. Räcke. A Practical Algorithm for Constructing Oblivious Routing Schemes. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '03, pages 24–33, New York, NY, USA, 2003. ACM. ISBN 1-58113-661-7. doi: 10.1145/777412.777418.

[19] OpenMP Architecture Review Board. OpenMP Specifications. OpenMP Specifications, July 2011.

[20] E. G. Boman and B. Hendrickson. Support Theory For Preconditioning. *SIAM J. Matrix Anal. Appl.*, 25:694–717, March 2003. ISSN 0895-4798. doi: 10.1137/S0895479801390637.

[21] E. G. Boman, B. Hendrickson, and S. A. Vavasis. Solving Elliptic Finite Element Systems in Near-Linear Time with Support Preconditioners. *CoRR*, cs.NA/0407022:–, 2004.

[22] J. A. Bondy and U. S. R. Murty. *Graph Theory*, volume 244 of *Graduate texts in mathematics*. Springer, New York [u.a.], 2008. ISBN 978-1-84628-969-9.

[23] J. Boussinesq. Théorie analytique de la chaleur: mise en harmonie avec la thermodynamique et avec la theorie mécanique de la lumière, 1903. Paris.

[24] H. Bowdler, R. Martin, G. Peters, and J. Wilkinson. Solution of Real and Complex Systems of Linear Equations. *Numerische Mathematik*, 8:217–234, 1966. ISSN 0029-599X. doi: 10.1007/BF02162559.

[25] D. Braess. *Finite Elemente : Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie*. Springer, Berlin, 2007. ISBN 978-3-540-72449-0. 4., überarb. und erw. Aufl.

[26] R. Budich. COSMOS Network Plan, February 2008. URL `http://comsos.enes.org`.

[27] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak. Mixed Precision Iterative Refinement Techniques for the Solution of Dense Linear Systems. *Int. J. High Perform. Comput. Appl.*, 21:457–466, November 2007. ISSN 1094-3420. doi: 10.1177/1094342007084026.

[28] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov. Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy. *ACM Trans. Math. Softw.*, 34:17:1–17:22, July 2008. ISSN 0098-3500. doi: 10.1145/1377596.1377597.

[29] C. Cao and E. S. Titi. Global Well-Posedness of the Three-Dimensional Viscous Primitive Equations of Large Scale Ocean and Atmosphere Dynamics. *Annals of Mathematics*, 166:245–267, 2007.

[30] João M. P. Cardoso, Pedro C. Diniz, and Markus Weinhardt. Compiling for Reconfigurable Computing: A Survey. *ACM Comput. Surv.*, 42(4):1–65, 2010. ISSN 0360-0300. doi: 10.1145/1749603.1749604.

[31] T. F. Chan and H. A. van der Vorst. Approximate And Incomplete Factorizations. In *ICASE LaRC Interdisciplinary Series in Science and Engineering*, pages 167–202, 1994.

[32] J. Cheeger. A lower Bound for the Smallest Eigenvalue of the Laplacian. In *Problems in analysis (Papers dedicated to Salomon Bochner, 1969)*, pages 195–199. Princeton Univ. Press, Princeton, N. J., 1970.

[33] D. Chen and S. Toledo. Vaidya's Preconditioners: Implementation and Experimental Study. *Electronic Transactions on Numerical Analysis*, 16:30–49, 2003.

[34] K. W. Chong, Y. Han, and T. W. Lam. Concurrent Threads and Optimal Parallel Minimum Spanning Trees Algorithm. *J. ACM*, 48:297–323, March 2001. ISSN 0004-5411. doi: 10.1145/375827.375847.

[35] M. D. Cox. A Primitive Equation Three-Dimensional Model of the Ocean. Technical Report 1, GFDL Ocean Group, Princeton University, 1984.

[36] J. Curreri, S. Koehler, B. Holland, and A. D. George. Performance Analysis with High-Level Languages for High-Performance Reconfigurable Computing. In *16th International Symposium on Field-Programmable Custom Computing Machines*, pages 23–30, 2008.

[37] É. J. M. Delhez and É. Deleersnijder. Overshootings and Spurious Oscillations Caused by Biharmonic Mixing. *Ocean Modelling*, 17(3):183–198, 2007. ISSN 1463-5003. doi: 10.1016/j.ocemod.2007.01.002.

[38] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, 1997. ISBN 0-89871-389-797808987138 93. URL `http://www.ulb.tu-darmstadt.de/tocs/59648104.pdf`.

[39] R. Diestel. *Graph Theory*. Graduate texts in mathematics ; 173. Springer, Berlin, 4. ed. edition, 2010. ISBN 978-3-642-14278-9.

[40] P. G. Doyle and J. L. Snell. *Random Walks and Electric Networks*. The Carus mathematical monographs ; 22. Mathematical Association of America, Washington, DC, 1984. ISBN 0-88385-024-9.

[41] DRC Computer Corporation. DRC Coprocessor System User's Guide. online, April 2009. v3.1.

[42] J. Edmonds and R. M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM*, 19:248–264, April 1972. ISSN 0004-5411. doi: 10.1145/321694.321699.

[43] L. Eisen, J. W. Ward, III, H.-W. Tast, N. Mäding, J. Leenstra, S. M. Mueller, C. Jacobi, J. Preiss, E. M. Schwarz, and S. R. Carlough. IBM POWER6 accelerators: VMX and DFU. *IBM J. Res. Dev.*, 51:663–683, November 2007. ISSN 0018-8646.

[44] M.l Elkin, Y. Emek, D. A. Spielman, and S.-H. Teng. Lower-stretch Spanning Trees. In *Proceedings of the thirty-seventh annual*

ACM symposium on Theory of computing, STOC '05, pages 494–503, New York, NY, USA, 2005. ACM. ISBN 1-58113-960-8. doi: 10.1145/1060590.1060665.

[45] G. Estrin. Organization of Computer Systems: The Fixed Plus Variable Structure Computer. In *Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference*, IRE-AIEE-ACM '60 (Western), pages 33–40, New York, NY, USA, 1960. ACM. doi: 10.1145/1460361.1460365.

[46] M. Feistauer. *Mathematical Methods in Fluid Dynamics*. Pitman monographs and surveys in pure and applied mathematics; 67. Longman [u.a.], Harlow, 1. publ. edition, 1993. ISBN 0-582-20988-9.

[47] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, sept. 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009071.

[48] P. Fofonoff and R. C. Millard Jr. Algorithms for Computation of Fundamental Properties of Seawater. UNESCO Technical Papers in Marine Science 4, Unesco, 1983.

[49] L. R. Ford and D. R. Fulkerson. Maximal Flow through a Network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[50] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Pr., Princeton, NJ, 1962. ISBN 0-691-07962-5.

[51] MPI Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

[52] A. E. Gill. *Atmosphere-Ocean Dynamics*, volume 30 of *International Geophysics Series*. Academic Press, 111 Fith Avenue, New York, New York, 1982. ISBN 0-12-283520-4 ; 0-12-283522-0.

[53] M. Gokhale and P. S. Graham. *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer, Berlin [u.a.], 2005. ISBN 0-387-26105-2 ; 978-0-387-26105-8.

[54] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins series in the mathematical sciences. Johns Hopkins Univ. Pr., Baltimore, Md. [u.a.], 3. ed., [nachdr.] edition, 1997. ISBN 0-8018-5413-X ; 0-8018-5414-8.

[55] K. Gremban. *Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, October 1996. CMU CS Tech Report CMU-CS-96-123.

[56] K. D. Gremban, G. L. Miller, and M. Zagha. Performance Evaluation of a New Parallel Preconditioner. Technical report, In Proceedings of the Ninth International Parallel Processing Symposium, 1995.

[57] S. M. Griffies. *Fundamentals of Ocean Climate Models*. Princeton, N.J. : Princeton University Press, 2004.

[58] S. M. Griffies, C. Böning, F. O. Bryan, E. P. Chassignet, R. Gerdes, H. Hasumi, A. Hirst, Treguier A.-M., and Webb D. Developments in Ocean Climate Modelling. *Ocean Modelling*, 2:123–192, 2000.

[59] S. Guattery. Graph Embedding Techniques for Bounding Condition Numbers of Incomplete Factor Preconditioners. Technical report, ICASE, NASA Langley Research, 1997.

[60] M. H. Gutknecht and S. Röllin. The Chebyshev Iteration Revisited. *Parallel Comput.*, 28(2):263–283, 2002. ISSN 0167-8191. doi: 10.1016/S0167-8191(01)00139-9.

[61] D. Göddeke, R. Strzodka, and S. Turek. Performance and Accuracy of Hardware-Oriented Native-, Emulated-and Mixed-Precision Solvers in FEM Simulations. *Int. J. Parallel Emerg. Distrib. Syst.*, 22:221–256, January 2007. ISSN 1744-5760. doi: 10.1080/17445760601122076.

[62] D. Göddeke, H. Wobker, R. Strzodka, J. Mohd-Yusof, P. McCormick, and S. Turek. Co-processor Acceleration of an Unmodified Parallel Solid Mechanics Code with FEASTGPU. *Int. J. Comput. Sci. Eng.*, 4(4):254–269, 2009. ISSN 1742-7185. doi: 10.1504/IJCSE.2009.029162.

[63] W. Hackbusch. *Iterative Solution of Large Sparse Systems of Equations*. Applied mathematical sciences ; 95. Springer, New York, 1994. ISBN 0-387-94064-2 ; 3-540-94064-2.

[64] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. Chapman & Hall/CRC computational science series. CRC, Boca Raton, Fla., 2011. ISBN 978-1-4398-1192-4 ; 1-439-81192-X.

[65] M. Herbordt, B. Sukhwani, M. Chiu, and Md. A. Khan. Production Floating Point Applications on FPGAs, July 2009. Symposium on Application Accelerators in High Performance Computing (SAAHPC'09).

[66] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro,

J. M. Willenbring, A. Williams, and K. S. Stanley. An Overview of the Trilinos Project. *ACM Trans. Math. Softw.*, 31(3):397–423, Sept. 2005. ISSN 0098-3500. doi: 10.1145/1089014.1089021.

[67] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, Dec 1952.

[68] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, 2. ed. edition, 2002. ISBN 0-89871-521-0.

[69] W. R. Holland. The Role of Mesoscale Eddies in the General Circulation of the Ocean—Numerical Experiments Using a Wind-Driven Quasi-Geostrophic Model. *Journal of Physical Oceanography*, 8(3):363–392, 1978. doi: 10.1175/1520-0485(1978)008<0363:TROMEI>2.0.CO;2.

[70] J. R. Holton. *An Introduction to Dynamic Meteorology*. International geophysics series ; 88. Elsevier Acad. Press, Amsterdam, 4. ed. edition, 2004. ISBN 0-12-354015-1.

[71] F. K. Hwang and D. S. Richards. Steiner Tree Problems. *Networks*, 22(1):55–89, 1992. ISSN 1097-0037. doi: 10.1002/net.3230220105.

[72] Intergovernmental. *Climate Change 2007 - The Physical Science Basis: Working Group I Contribution to the Fourth Assessment Report of the IPCC*. Cambridge University Press, Cambridge, UK and New York, NY, USA, 2007. ISBN 0521880092.

[73] Intergovernmental Panel on Climate Change. Climate Change 2007: Synthesis Report. Summary for Policymakers. Technical report, Intergovernmental, 2007.

[74] V. John, G. Matthies, and J. Rang. A Comparison of Time-Discretization/Linearization Approaches for the Incompressible Navier-Stokes Equation. *Computer Methods in Applied Mechanics and Engineering*, 195(44–47):5995–6010, 2006. doi: 10.1016/j.cma.2005.10.00.

[75] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open Source Scientific Tools for Python, 2001. URL `http://www.scipy.org/`.

[76] W. Kahan. *Gauss-Seidel Methods of Solving Large Systems of Linear Equations*. PhD thesis, University of Toronto, Canada, 1958.

[77] J. Kahle. The Cell Processor Architecture. In *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, page 3, 2005. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1540943`.

[78] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[79] I. Koutis. *Combinatorial and Algebraic Tools for Optimal Multilevel Algorithms*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 2007.

[80] I. Koutis. Personal Communication, 2011.

[81] I. Koutis and G. L. Miller. Graph Partitioning into Isolated, High Conductance Clusters: Theory, Computation and Applications to Preconditioning. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 137–145, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-973-9. doi: 10.1145/1378533.1378559.

[82] I. Koutis, G. L. Miller, and R. Peng. Approaching Optimality for Solving SDD Linear Systems, April 2010.

[83] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra. Exploiting the Performance of 32 bit Floating Point Arithmetic in Obtaining 64 bit Accuracy (Revisiting Iterative Refinement for Linear Systems). In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi: 10.1145/1188455.1188573.

[84] J. Larson, R. Jacob, and E. Ong. The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models. *Inter. J. High Perform. Comput. Appl.*, 19(3):pp. 277 – 292, 2004.

[85] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 Microarchitecture. *IBM J. Res. Dev.*, 51(6):639–662, 2007. ISSN 0018-8646. doi: 10.1147/rd.516.0639.

[86] G. G. F. Lemieux. Hardware Performance Monitoring in Multiprocessors. Technical report, Computer Engineering, University of Toronto, 1996.

[87] C.-J. Lin and J. J. Moré. Incomplete Cholesky Factorizations With Limited Memory. *SIAM J. SCI. COMPUT*, 21(1):24–45, 1999.

[88] A. R. Lopes, G. A. Constantinides, and E. C. Kerrigan. A Floating-Point Solver for Band Structured Linear Equations. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 353–356, October 2008. doi: 10.1109/FPT.2008.4762416.

[89] P. Lynch. The Origins of Computer Weather Prediction and Climate Modeling. *J. Comput. Phys.*, 227:3431–3444, March 2008. ISSN 0021-9991. doi: 10.1016/j.jcp.2007.02.034.

[90] B. M. Maggs, G. L. Miller, O. Parekh, R. Ravi, and S. L. M. Woo. Finding Effective Support-Tree Preconditioners. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 176–185, New York, NY, USA, 2005. ACM. ISBN 1-58113-986-1. doi: 10.1145/1073970.1073996.

[91] S. J. Marsland, H. Haak, J. H. Jungclaus, M. Latif, and F. Röske. The Max-Planck-Institute Global Ocean/Sea Ice Model with Orthogonal Curvilinear Coordinates. *Ocean Modelling*, 5:91–127, 2003.

[92] R. Martin, G. Peters, and J. Wilkinson. Iterative Refinement of the Solution of a Positive Definite System of Equations. *Numerische Mathematik*, 8:203–216, 1966. ISSN 0029-599X. doi: 10.1007/BF02162558. 10.1007/BF02162558.

[93] P. C. Matthews. *Vector Calculus*. Springer Undergraduate Mathematics Series. Springer, London, 1998. ISBN 3-540-76180-2.

[94] J. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. URL `http://www.cs.virginia.edu/stream/`.

[95] N. Megiddo. Optimal Flows in Networks with Multiple Sources and Sinks. *Mathematical Programming*, 7:97–107, 1974. ISSN 0025-5610. doi: 10.1007/BF01585506.

[96] H. Moritz. Geodetic Reference System 1980 (GRS80). *Bulletin Géodésique*, 54:128–162, 1980.

[97] I. Moulitsas and G. Karypis. Architecture Aware Partitioning Algorithms. In *Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP '08, pages 42–53, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-69500-4. doi: 10.1007/978-3-540-69501-1_6.

[98] R. J. Murray. Explicit Generation of Orthogonal Grids for Ocean Models. *J. Comput. Phys.*, 126(2):251–273, 1996. ISSN 0021-9991. doi: 10.1006/jcph.1996.0136.

[99] F. Nebeker. *Calculating the Weather: Meteorology in the 20th Century*, volume 60 of *International Geophysics Series*. Academic Press, 525 B Street, Suite 1900, San Diego, California 92101-4495, 1995. ISBN 0-12-515175-6.

[100] J. Nickolls and W. J. Dally. The GPU Computing Era. *Micro, IEEE*, 30(2):56–69, March 2010. ISSN 0272-1732. doi: 10.1109/MM.2010.41. URL http://dx.doi.org/10.1109/MM.2010.41.

[101] H. Oertel, M. Böhle, and U. Dohrmann. *Strömungsmechanik: Grundlagen - Grundgleichungen - Lösungsmethoden - Softwarebeispiele*. Studium. Vieweg + Teubner, Wiesbaden, 5., überarb. und erw. aufl. edition, 2009. ISBN 978-3-8348-0483-9.

[102] D. P. O'Leary. *A Century of Excellence in Measurements, Standards, and Technology; A Chronicle of Selected NBS/NIST Publications, 1901-2000*, volume 958, chapter Methods of Conjugate Gradients for Solving Linear Systems, pages 81–85. NIST Special Publication, 2001. URL http://nvl.nist.gov/pub/nistpubs/sp958-lide/cntsp958.htm.

[103] L. Oliker, A. Canning, J. Carter, J. Shalf, D. Skinner, S. Ethier, R. Biswas, J. Djomehri, and R. Van Der Wijngaart. Performance Evaluation of the SX6 Vector Architecture for Scientific Computations, 2005.

[104] T. E. Oliphant. Python for Scientific Computing. *Computing in Science and Engg.*, 9:10–20, May 2007. ISSN 1521-9615. doi: 10.1109/MCSE.2007.58.

[105] R. C. Pacanowski and S. M. Griffies. *MOM 3.0 Manual*, 2000.

[106] V. K. Prasanna. Energy-Efficient Computations on FPGAs. *The Journal of Supercomputing*, 32:139–162, 2005. ISSN 0920-8542. doi: 10.1007/s11227-005-0289-9.

[107] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing, Second Edition*. Cambridge University Press, 2 edition, Oct 1992. ISBN 0521431085.

[108] M. Pütz. BLIZZARD System Overview. Presentation at the DKRZ POWER6 Programming Workshop, July 2009.

[109] A. Quarteroni and A. Valli. *Domain Decomposition Methods for Partial Differential Equations*. Numerical mathematics and scientific computation. Clarendon Press, Oxford [u.a.], repr. edition, 2005. ISBN 0-19-850178-1.

[110] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical Mathematics*. Springer, 2000.

[111] D. Quinlan and M. Berndt. MLB: Multilevel Load Balancing for Structured Grid Applications. In *Proceedings of the SIAM Parallel Conference*, Minneapolis, MN, 1997.

[112] S. K. Raman, V. Pentkovski, and J. Keshava. Implementing Streaming SIMD Extensions on the Pentium III Processor. *IEEE Micro*, 20(4):47–57, July 2000. ISSN 02721732. doi: 10.1109/40.865866. URL http://dx.doi.org/10.1109/40.865866.

[113] R. Rannacher. Numerische Mathematik 3 (Numerik von Problemen der Kontinuumsmechanik), May 2008. URL http://numerik.iwr.uni-heidelberg.de/~lehre/notes/.

[114] R. Redler, S. Valcke, and H. Ritzdorf. OASIS4 – a Coupling Software for Next Generation Earth System Modelling. *Geoscientific Model Development*, 3(1):87–104, 2010. doi: 10.5194/gmd-3-87-2010.

[115] J. K. Reid. *Large Sparse Sets of Linear Equations*, chapter On the Method of Conjugate Gradients for the Solution of Large Sparse Systems of Linear Equations, pages 231–254. Academic Press, New York, 1971.

[116] L. F. Richardson. *Weather Prediction by Numerical Process*. Cambridge University Press, 1922.

[117] B. H. K. Rocker. *Hardware-Aware Solvers for Large, Sparse Linear Systems : Multi-Precision and Parallel Approaches*. PhD thesis, Karlsruhe Institute of Technology, 2011. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023260.

[118] E. Roeckner, G. Bäuml, L. Bonaventura, R. Brokopf, M. Esch, M. Giorgetta, S. Hagemann, I. Kirchner, L. Kornblueh, E. Manzini, A. Rhodin, U. Schlese, U. Schulzweida, and A. Tompkins. The Atmospheric General Circulation Model ECHAM5 Part I. MPI Report 349, Max-Planck-Insitut für Meteorologie, November 2003.

[119] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2. ed. edition, 2003. ISBN 0-89871-534-2.

[120] M. Schmidtobreick. Numerical Methods on Reconfigurable Hardware using High Level Programming Paradigms. Master's thesis, Karlsruhe Institute of Technology, 2010.

[121] H. A. Schwarz. Ueber einen Grenzübergang durch alternirendes Verfahren. *Vierteljahresschrift der Naturforschenden Gesellschaft in Zürich*, 15:272–286, 1870.

[122] A. J. Semtner and Y. Mintz. Numerical Simulation of the Gulf Stream and Mid-Ocean Eddies. *Journal of Physical Oceanography*, 7(2):208–230, 1977. doi: 10.1175/1520-0485(1977)007<0208:NSOTGS>2.0.CO;2.

[123] A. Sergyienko and O. Maslennikov. Implementation of Givens QR-Decomposition in FPGA. In *PPAM '01: Proceedings of the 4th International Conference on Parallel Processing and Applied Mathematics-Revised Papers*, pages 458–465, London, UK, 2002. Springer-Verlag. ISBN 3-540-43792-4.

[124] E. Simonnet, Tachim T. Medjo, and R. Temam. Barotropic-Baroclinic Formulation of the Primitive Equations of the Ocean. *Applicable Analysis*, 82(5):439 – 456, 2003. doi: 10.1080/0003681031000094591.

[125] B. F. Smith, P. E. Bjørstad, and W. D. Gropp. *Domain decomposition : Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge Univ. Press, Cambridge [u.a.], 1996. ISBN 0-521-49589-X.

[126] D. A. Spielman and S.-H. Teng. Solving Sparse, Symmetric, Diagonally-Dominant Linear Systems in Time O($m^{1.31}$). *Foundations of Computer Science, Annual IEEE Symposium on*, 0:416, 2003. ISSN 0272-5428. doi: 10.1109/SFCS.2003.1238215.

[127] D. A. Spielman and J. Woo. A Note on Preconditioning by Low-Stretch Spanning Trees. *ArXiv e-prints*, 2009. In Proceedings of CoRR.

[128] P. Stier, J. Feichter, S. Kinne, Kloster S., Vignati E., Wilson J., L. Ganzeveld, I. Tegen, M. Werner, Y. Balkanski, M. Schulz, O. Boucher, A. Minikin, and A. Petzold. The Aerosol-Climate Model ECHAM5-HAM. *Atmos. Chem. Phys.*, 5:1125—1156, 2005.

[129] O. Storaasli, W. Yu, D. Strenski, and J. Maltby. Performance Evaluation of FPGA-Based Biological Applications. Online, May 2007. Proc Cray Users Group'07.

[130] E. Süli and D. F. Mayers. *An Introduction to Numerical Analysis*. Cambridge University Press, Cambridge [u.a.], 1. publ. edition, 2003. ISBN 0-521-81026-4; 0-521-00794-1.

[131] Impulse Accelerated Technologies. CoDeveloper User Guide, 2009. Version 3.60.

[132] R. Temam and A. Miranville. *Mathematical Modeling in Continuum Mechanics*. Cambridge University Press, Cambridge [u.a.], 2. ed. edition, 2005. ISBN 0-521-64362-7.

[133] J. D. Teresco, J. E. Flaherty, S. B. Baden, J. Faik, S. Lacour, M. Parashar, V. E. Taylor, and C. A. Verela. Approaches to Architecture-Aware Parallel Scientific Computation. Technical report, Williams College Department of Computer Science, 2005.

[134] A. Toselli and O. Widlund. *Domain Decomposition Methods - Algorithms and Theory*. Number 34 in Springer series in computational mathematics. Springer, Berlin, 2005. ISBN 3-540-20696-5; 978-3-642-05848-6.

[135] K. E. Trenberth, editor. *Climate System Modeling*. Cambridge Univ. Press, Cambridge [u.a.], 1. publ. edition, 1992. ISBN 0-521-43231-6.

[136] *Challenges in Climate Change Science and the Role of Computing at the Extreme Scale*, Scientific Grand Challenges, 2008. U.S. Department of Energy. Report from the Workshop Held November 6-7, 2008.

[137] P. M. Vaidya. Solving Linear Equations with Symmetric Diagonally Dominant Matrices by Constructing Good Preconditioners. An unpublished manuscript. A talk based on the manuscript was presented at the IMA Workshop on Graph Theory and Sparse Matrix Computations in Minneapolis, October 1991.

[138] L. G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33:103–111, August 1990. ISSN 0001-0782. doi: 10.1145/79173.79181.

[139] W. S. VonArx. *Introduction to Physical Oceanography*. Addison-wesley Series in the earth sciences. Addison-Wesley Publ. Co., Reading, Mass., 1962.

[140] P. Wesseling. *Principles of Computational Fluid Dynamics*. Springer series in computational mathematics; 29. Springer, Berlin, 1. softcover printing edition, 2009. ISBN 978-3-642-05145-6.

[141] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Dover Publications, Incorporated, 1994. ISBN 0486679993. Originally published: Englewood Cliffs, N.J. : Prentice-Hall, 1963.

[142] J. O. Wolff, E. Maier-Reimer, and S. Legutke. The Hamburg Ocean Primitive Equation Model HOPE. Technical report, DKRZ Hamburg, 1997.

[143] D. M. Young. *Iterative Solution of Large Linear Systems*. Computer science and applied mathematics. Acad. Press, New York [u.a.], 1971. ISBN 0-12-773050-8.