Designing Scientific Software for Heterogeneous Computing

With application to large-scale water wave simulations

Glimberg, Stefan Lemvig; Engsig-Karup, Allan Peter; Dammann, Bernd

Publication date:
2013

Document Version
Publisher's PDF, also known as Version of record

Designing Scientific Software for Heterogeneous Computing

With application to large-scale water wave simulations

Stefan Lemvig Glimberg

Kongens Lyngby 2013
IMM-PhD-2013-317

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk IMM-PhD-2013-317

Preface

This thesis was prepared at the Technical University of Denmark in fulfillment of the requirements for acquiring a PhD degree. The work has been carried out during the period of May 2010 to November 2013 at the Department of Applied Mathematics and Computer Science at the Scientific Computing section.

Part of the work has been carried out during my research visit to the Scientific Computing section at University of Illinois at Urbana-Champaign, USA, autumn 2011. The visit was hosted by Prof. Luke Olson and partially sponsored by Otto Mønsteds Fond for which I am grateful.

Lyngby, October 30^{th}, 2013.

Stefan Lemvig Glimberg

Summary (English)

The main objective with the present study has been to investigate parallel numerical algorithms with the purpose of running efficiently and scalably on modern many-core heterogeneous hardware. In order to obtain good efficiency and scalability on modern multi- and many- core architectures, algorithms and data structures must be designed to utilize the underlying parallel architecture. The architectural changes in hardware design within the last decade, from single to multi and many-core architectures, require software developers to identify and properly implement methods that both exploit concurrency and maintain numerical efficiency.

Graphical Processing Units (GPUs) have proven to be very effective units for computing the solution of scientific problems described by partial differential equations (PDEs). GPUs have today become standard devices in portable, desktop, and supercomputers, which makes parallel software design applicable, but also a challenge for scientific software developers at all levels. We have developed a generic C++ library for fast prototyping of large-scale PDEs solvers based on flexible-order finite difference approximations on structured regular grids. The library is designed with a high abstraction interface to improve developer productivity. The library is based on modern template-based design concepts as described in Glimberg, Engsig-Karup, Nielsen & Dammann (2013). The library utilizes heterogeneous CPU/GPU environments in order to maximize computational throughput by favoring data locality and low-storage algorithms, which are becoming more and more important as the number of concurrent cores per processor increases.

We demonstrate in a proof-of-concept the advantages of the library by assem-

bling a generic nonlinear free surface water wave solver based on unified potential flow theory, for fast simulation of large-scale phenomena, such as long distance wave propagation over varying depths or within large coastal regions. Simulations that are valuable within maritime engineering because of the adjustable properties that follow from the flexible-order implementation. We extend the novel work on an efficient and robust iterative parallel solution strategy proposed by Engsig-Karup, Madsen & Glimberg (2011), for the bottleneck problem of solving a σ-transformed Laplace problem in three dimensions at every time integration step. A geometric multigrid preconditioned defect correction scheme is used to attain high-order accurate solutions with fast convergence and scalable work effort. To minimize data storage and enhance performance, the numerical method is based on matrix-free finite difference approximations, implemented to run efficiently on many-core GPUs. Also, single-precision calculations are found to be attractive for reducing transfers and enhancing performance for both pure single and mixed-precision calculations without compromising robustness.

A structured multi-block approach is presented that decomposes the problem into several subdomains, supporting flexible block structures to match the physical domain. For data communication across processor nodes, messages are sent using MPI to repeatedly update boundary information between adjacent coupled subdomains. The impact on convergence and performance scalability using the proposed hybrid CUDA-MPI strategy will be presented. A survey of the convergence and performance properties of the preconditioned defect correction method is carried out with special focus on large-scale multi-GPU simulations. Results indicate that a limited number of multigrid restrictions are required, and that it is strongly coupled to the wave resolutions. These results are encouraging for the heterogeneous multi-GPU systems as they reduce the communication overhead significantly and prevent both global coarse grid corrections and inefficient processor utilization at the coarsest levels.

We find that spatial domain decomposition scales well for large problems sizes, but for problems of limited sizes, the maximum attainable speedup is reached for a low number of processors, as it leads to an unfavorable communication to compute ratio. To circumvent this, we have considered a recently proposed parallel-in-time algorithm referred to as Parareal, in an attempt to introduce algorithmic concurrency in the time discretization. Parareal may be perceived as a two level multigrid method in time, where the numerical solution is first sequentially advanced via course integration and then updated simultaneously on multiple GPUs in a predictor-corrector fashion. A parameter study is performed to establish proper choices for maximizing speedup and parallel efficiency. The Parareal algorithm is found to be sensitive to a number of numerical and physical parameters, making practical speedup a matter of parameter tuning. Results are presented to confirm that it is possible to attain reasonable speedups, independently of the spatial problem size.

To improve application range, curvilinear grid transformations are introduced to allow representation of complex boundary geometries. The curvilinear transformations increase the complexity of the implementation of the model equations. A number of free surface water wave cases have been demonstrated with boundary-fitted geometries, where the combination of a flexible geometry representation and a fast numerical solver can be a valuable engineering tool for large-scale simulation of real maritime scenarios.

The present study touches some of the many possibilities that modern heterogeneous computing can bring if careful and parallel-aware design decisions are made. Though several free surface examples are outlined, we are yet to demonstrate results from a real large-scale engineering case.

Summary (Danish)

Hovedformålet med dette studie har været, at undersøge parallele numeriske algoritmer der kan eksekveres effektivt og skalerbart på moderne mange-kerne heterogen hardware. For at opnå effektivitet og skalerbarhed på moderne multi- og mange-kerne arkitekuterer må algoritmer og datastrukturer designes til at udnytte den underliggende parallelle arkitektur. De seneste års skift indenfor hardware design, fra enkelt- til multi-kerne arkitekturer, kræver at software-udviklere identificerer og implementerer metoder der udnytter parallelitet og bevarer numerisk effektivitet.

Graphical Processing Units (GPU'er) har vist sig at være særdeles gode beregningsenheder til løsning af videnskabelige problemer beskrevet ved partielle differential ligninger (PDE'er). GPU'er er i dag standard i både bærbare, desktop og supercomputere, hvilket gør parallel software design aktuelt, men også udfordrende, for videnskabelige softwareudviklere på alle niveauer. Vi har udiklet et generisk C++ bibliotek til hurtig proto-typing af stor-skala løsere, baseret på fleksibel-ordens finite difference approximationer på strukturerede og regulære net. Biblioteket er designet med at abstract interface for at forbedre udviklerens produktivitet. Biblioteket er baseret på moderne template-baserede designkoncepter som beskrevet i Glimberg, Engsig-Karup, Nielsen & Dammann (2013). Biblioteket udnytter heterogene CPU/GPU systemer for at maximere beregningseffektiviteten ved at udnytte datalokalitet og hukommelsesbesbarende algoritmer, hvilket kun bliver vigtigere og vigtigere i takt med at der kommer flere kerner per processor.

Vi demonstrerer, i et proof-of-concept, fordelene ved biblioteket ved at sammensætte en ikke-linær vandbølgeløser baseret på potential flow teori, til effektiv

simulering af stor-skala fænomener, såsom langdistance bølgetransformationer over varierende vanddybder eller indenfor større kystområder. Sådanne simuleringer har stor værdi indenfor maritime analyser på grund af de justerbare egenskaber der følger med den fleksibel-ordens implementering. Vi udvider arbejdet af en effektiv og robust iterativ parallel strategi, foreslået af Engsig-Karup, Madsen & Glimberg (2011), til løsning af et σ-transformeret Laplace problem i tre dimensioner. En geometrisk multigrid pre-konditioneret defect correction metode er benyttet til at opnå høj-ordens nøjagtige løsninger med hurtig konvergens og skalerbar beregningsarbejde. For at minimere hukommelsesforbruget og forbedre performance er den numeriske metode baseret på matrix-frie finite difference approximationer, implementeret til effektivt eksekvering på mangekerne GPU'er. Derudover er det vist at single-præcisions beregninger kan være attraktive til at reducere hukommelsesoverførsler og forbedre performance, uden at kompromitere nøjagtigheden af resultaterne.

En struktureret multi-blok teknik er præsenteret der inddeler problemet i flere delproblemer, der kan tilpasses det fysiske domæne. Beskeder sendes via MPI for at opdatere randinformationer mellem nabo-blokke. Indvirkningen på konvergens og performanceskalering med den foreslåede CUDA-MPI hybridmetode er undersøgt og præsenteres. En undersøgelse af konvergens og performance af defect correction metoden er lavet, med særlig fokus på stor-skala multi-GPU simuleringer. Resultaterne indikerer at et begrænset antal af multigrid restriktioner er nødvendigt og at antallet er stærkt koblet til bølgeopløsningen. Disse resultater tilskyndes heterogene multi-GPU systemer, fordi de reducerer kommunikations-overhead signifikant og forhindrer både global coarse grid korrektioner og ineffektiv udnyttelse af processorerne på de grove grid niveauer.

Vi demonstrerer at spatial domæne dekompositionering skalerer godt for store problemstørrelser, men at for mindre problemer opnås den maksimale hastighedsforøgelse for et lavt antal processorer, da det fører til et ugunstigt kommunikations-til-beregnings forhold. For at imødekomme dette, har vi undersøgt en algoritme til parallelisering i den tidslige dimension, kaldet Parareal. Parareal kan betragtes som en to-niveau multigrid metode i tid, hvor den numeriske løsning først propageres med store tidsskridt. Disse mellemliggende tidsskridt kan så benyttes som begyndelsesbetingelser for nøjagtigere beregninger der kan udføres parallelt vha. flere GPU'er. Et parameterstudie er udført for at demonstrere valg der optimerer speedup og parallel effektivitet. Parareal algoritmen har vist sig at være sensitiv overfor er række af numeriske og fysiske parametre, hvilket gør effektiv speedup til et spørgsmål om parametertuning. Resultater præsenteres der bekræfter at der er muligt at opnå fornuftige hastighedsforøgelser, uafhængigt at den rumlige diskretisering.

For at forbedre anvendelsesmulighederne indenfor mere komplekse modeller introduceres kurvilinære koordinater. Brug af kurvilinære koordinater er demon-

streret på en række testeksempler for bølgemodellen, hvor kombinationen af fleksible geometrier og en hurtig numerisk løser kan være et værdifuldt inginørværktøj til stor-skala simulering af virkelige maritime scenarier.

Dette studie berører mange af de muligheder moderne heterogene beregninger kan bringe hvis omhyggelige og parallel-bevidste beslutninger tages. Selvom flere eksempler på bølgesimuleringer er præsenteret, mangler vi endnu at vise en stor-skala test baseret på en virkelig inginøropgave.

Acknowledgements

First, I would like to give great thanks to my main supervisor Assoc. Prof. Allan P. Engsig-Karup for his strong commitment to the project and for all of his knowledge-sharing. I truly appreciate all of the guidance and the strong effort he has put into this project.

I would also like to thank all the people involved in the GPUlab project at the Technical University of Denmark, in particular my co-supervisor Assoc. Prof. Bernd Dammann and fellow students Nicolai Gade-Nielsen and Hans Henrik B. Sørensen, for all of the inspiring and interesting conversations. I also acknowledge the guidance and advice I have received from all those involved in the OceanWave3D developer group meetings, and I thank Ole Lindberg for the collaboration we have had.

I would also like to thank Prof. Jan S. Hesthaven, Wouter Boomsma, and Andy R. Terrel for granting access to the latest hardware and large-scale compute facilities. Scalability and performance tests were carried out at the GPUlab at DTU Compute, the Oscar GPU cluster at the Center for Computing and Visualization, Brown University, and the Stampede HPC cluster at the Texas Advanced Computing Center, University of Texas. The NVIDIA Corporation is also acknowledged for generous hardware donations to GPUlab.

This work was supported by grant no. 09-070032 from the Danish Research Council for Technology and Production Sciences (FTP). I am very grateful for the financial support that I have received from FTP, and from Otto Mønsteds Fond and Oticon Fonden to cover parts of my travel expenses.

Last, I would like to thank my beloved wife and our wonderful children for their endless and caring support.

Declarations

The work presented in this dissertation is a compilation of all the work that has been carried out during the project's three year period, some of which has been described in the following published references:

- Engsig-Karup, A. P., **Glimberg, S. L.**, Nielsen, A. S., Lindberg, O.. Designing Scientific Applications on GPUs, *Chapter:* Fast hydrodynamics on heterogenous many-core hardware. *Chapman & Hall/CRC. Numerical Analysis and Scientific Computing Series*, 2013.
- Engsig-Karup, A. P., Madsen, M. G., **Glimberg, L. S.**. A massively parallel gpu-accelerated model for analysis of fully nonlinear free surface waves. In: *International Journal for Numerical Methods in Fluids*, vol. 70, pp. 20–36, 2011.
- **Glimberg, S. L.**, Engsig-Karup, A. P., Nielsen, A. S., Dammann, B.. Designing Scientific Applications on GPUs, *Chapter:* Development of software components for heterogeneous many-core architectures. *Chapman & Hall/CRC Numerical Analysis and Scientific Computing Series*, 2013.
- **Glimberg, S. L.**, Engsig-Karup, A. P., Madsen, M. G.. A Fast GPU-accelerated Mixed-precision Strategy for Fully Nonlinear Water Wave Computations, In: *Proceedings of European Numerical Mathematics and Advanced Applications (ENUMATH)*, 2011.
- Lindberg, O., **Glimberg, S. L.**, Bingham, H. B., Engsig-Karup, A. P., Schjeldahl, P. J.. Real-Time Simulation of Ship-Structure and Ship-Ship Interaction. In *3rd International Conference on Ship Manoeuvring in Shallow and Confined Water*, 2013.

- Lindberg, O., **Glimberg, S. L.**, Bingham, H. B., Engsig-Karup, A. P., Schjeldahl, P. J.. Towards real time simulation of ship-ship interaction - Part II: double body flow linearization and GPU implementation. In: *Proceedings of The 28th International Workshop on Water Waves and Floating Bodies (IWWWFB)*, 2012.

Parts of these references have provided a basis for the work presented in this dissertation as follows:

- Most of Chapter 2 has previously been published in Glimberg et al. (2013)
- Parts of Chapter 4 have previously been published in Glimberg et al. (2013) and Engsig-Karup et. al. (2013)
- Parts of Chapter 5 have previously been published in Glimberg et al. (2013) and Engsig-Karup et. al. (2013)

Contents

Chapter 1

Introduction to heterogeneous computing

Based on few years of observations, Gordon E. Moore predicted in 1965, that the number of processor on an integrated circuit would double approximately every two years [Moo65]. Though this prediction is almost fifty years old, it has been remarkably accurate, though today it has become more of a prophecy or trend setter for the industry to follow in order to keep up with their competitors.

For many years, chip manufactures were able to produce single-core processors with increased clock frequencies following Moore's law, enabling faster execution of any software application, with no modifications to the underlying code required. However, within the last ten years there has been a remarkable change in the architectural design of microprocessors. Issues with power constraints and uncontrollable heat dissipation have forced manufactures to favor multi-core chip design in order to keep up

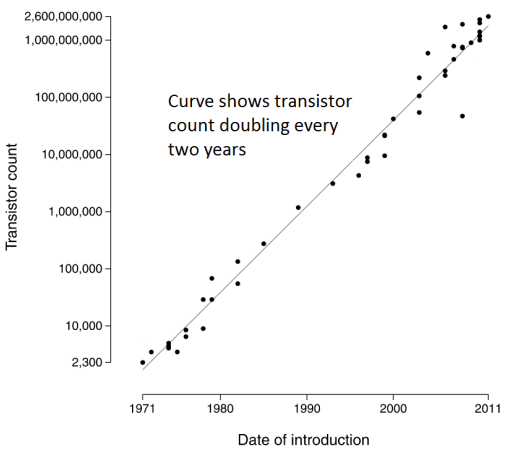


Figure 1.1: Moore's law until 2011.

with Moore's law [ITRon]. These architectural design changes have caused a paradigm shift, and as a consequence software developers can no longer rely on increased performance as a result of new and faster hardware. Sequential legacy codes will have to be redesigned and re-implemented to fit the emerging parallel platforms. Unfortunately, these parallel paradigms tend to introduce additional overhead, causing less than linear performance improvements as the number of processors increases. Well-designed algorithms with little or no sequential dependency and communication overhead are essential for good performance and scalability on parallel computers. Though the first parallel computers dates back to the 1950s and high-performance computing (HPC) topics have been researched for decades, parallel computing has been limited to few developers and mostly focused on utilizing distributed clusters for advanced applications. With recent trends, parallel computing is now more accessible for the masses than ever before, and therefore it is now–more than ever–fundamentally important that basic principles of efficient, portable, and scalable parallel algorithms and design patterns are investigated and developed.

1.1 HPC on affordable emerging architectures

As a consequence of these emerging multi-core processors, there has been a rapidly growing market for low-cost, low energy, and easy accessible HPC resources, with a broad target group of software developers and engineers from different research areas. Optimal utilization of all processor cores is becoming a desirable feature for software developers and a necessity in almost any commercial application. Today, multi-core processors have become the standard in any personal desktop or laptop computer, many of them are also accompanied with a many-core co-processing Graphical Processing Unit (GPU). This combined setup constitutes a heterogeneous setup, where the GPU can be used as a specialized compute accelerator for given applications. The intense promotion of programmable GPUs, has been a key contributor to the breakthrough in HPC on mass-produced commodity hardware and they have opened up new opportunities within scientific computing and mathematical modeling. Pioneered by Nvidia and AMD, graphics hardware has developed into easily programmable high-performance platforms, suitable for many kinds of general purpose applications with no connection to computer graphics.

1.1.1 Programmable Graphical Processing Units

Graphical processors became popular as part of the growing gaming industry during the 1990's. Back then, GPUs only supported specialized fixed-function pipelines. During the early 2000s, the first work on General Purpose GPU (GPGPU) computing was initiated, but required profound understanding of graphical programming interfaces and shader languages, such as the Open Graphics Library (OpenGL), Direct3D, OpenGL Shading Language (GLSL), C for Graphics (Cg), or the High-level Shader Language (HLSL). These early and promising results, presented by the rising GPGPU community, led to the development of several high-level languages for graphics hardware, to help developers run applications on the GPUs. Though several languages were proposed, e.g., BrookGPU[BFH+04, JS05] developed at Stanford University or Close-to-The-Metal by ATI (now AMD), only two programming models remain as the main competitors today; CUDA and OpenCL. CUDA (Compute Unified Device Architecture) is developed and maintained by the Nvidia Corporation and therefore runs exclusively on Nvidia GPUs. The CUDA project was initiated in 2006, and has been subject to an aggressive promotion campaign by Nvidia, in order to ensure a solid market share in both industrial, academic, and personal computing. Therefore the proportion of CUDA documentation, articles, code examples, user guides, etc. are still dominating, though the interest seems to have peaked compared to OpenCL. OpenCL was initially developed by Apple in 2008 with support from AMD, and was released later in 2008 and is now maintained by the Khronos Group. OpenCL is a more versatile model as it is designed for execution on any multiprocessor platform and not limited to GPUs. Throughout this work we are using CUDA as our programming model, both because it has proven to be the most mature and because it directly supports generic programming via C++ templates. We note that this dissertation is not an introduction to CUDA or MPI programming. We expect the reader to be familiar with the basic concepts of HPC and GPU programming, such as the CUDA thread hierarchy, shared memory, kernels, ranks, etc. For a thorough introduction to GPU architecture and CUDA programming we refer the reader to the books [KmWH10, Coo12, Nvi13] or [SK10, Far11, Hwu11] for more application oriented introductions. Introduction to MPI can be found in [GLS99, GLT99].

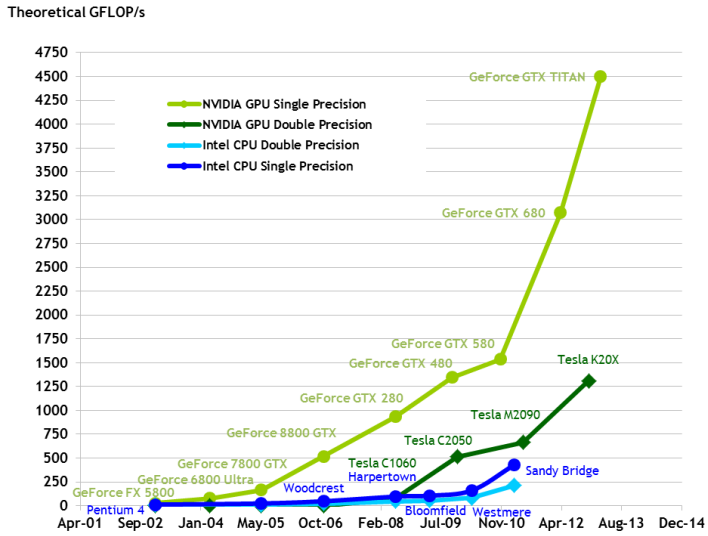
The GPU obtained its popularity from its massively parallel design architecture, based on the Single Instruction Multiple Thread (SIMT) model, meaning that the multiprocessors execute the same instructions with multiple threads, yet allowing conditional operations. The promotion of parallel GPU programming has been carried by the manufacturers (Nvidia in particular), who have used their noticeable peak performance numbers in comparisons to the traditional CPU alternatives to emphasize their eligibility on the HPC market, cf. Figure

1.2.

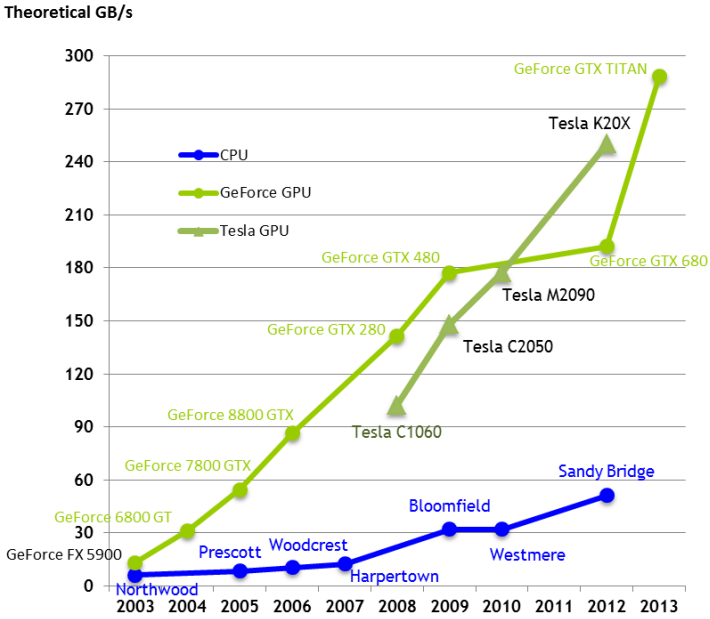
Though the majority of GPUs are still produced and sold to the gaming industry, they have also established themselves as important components in high-performance accelerated computing, by entering several of the top rankings on the Top500 list of the most powerful computers in the world[Gmb]. In November 2012, the Titan supercomputer at Oak Ridge National Laboratory topped the list with more than 27PFlops peak performance, at only 8209kW. Titan has 18.688 compute nodes, each equipped with an Nvidia Tesla K20 GPU, providing more than 80% of its total peak performance. Titan was assembled as an upgrade to the previous supercomputer Jaguar, achieving almost 10 times performance speedup at approximately the same power consumption.

As of 2013, several of the fastest supercomputers in the world rely on co-processing accelerators for high-performance, with Nvidia GPUs being the most prominent. More than 50 of the Top500 supercomputers use accelerators to speed up their computational performances. This number has increased from below five in just six years [Gmb]. In addition, the increasing need for energy efficient supercomputers, as we approach the exascale era[DBM+11, BMK+10], have forced supercomputer vendors to pay more and more attention to heterogeneous computers, because co-processors, such as the GPU, offer an favorable performance to watt ratio. Today, the top of the Green500 list (measured in Flops/watt) is dominated by heterogeneous systems [FC]. The most energy-efficient supercomputer as of June 2013, breaks through the three billion floating-point operations per second per watt barrier for the first time. GPUs are energy efficient compared to CPUs, because the cores are running at approximately one-forth of the speed of a CPU core. The GPU achieves its superior performance because of the high number of cores running in parallel.

Traditionally supercomputers have been profiled against the Linpack benchmark. However, this tradition seems to be changing, as Linpack is no longer a good representative for supercomputing performance profiling. The number of applications that rely on sparse matrix-vector products, stencil operations, and irregular memory access patterns, such as those based on differential equations, is increasing. In practice this means that these applications will be limited by the memory bandwidth wall and cannot obtain the optimistic performance measures in Figure 1.2a but rather those in 1.2b. Therefore, a new benchmark, the High Performance Conjugate Gradient (HPCG), has been proposed to meet the requirements of modern applications. Future Top500 lists will therefore be available based on this performance scale as well.



(a) Floating point performance.



(b) Memory bandwidth

Figure 1.2: Peak performance and memory bandwidth for recent generations of CPUs and GPUs. From [Nvi13].

1.2 Scope and main contributions

In this thesis we will try to address some of the challenges that face software developers when introduced to new programming paradigms for massively parallel executions on GPUs. We will in particular discuss and present the challenges that are related to computing the solution of partial differential equation (PDE) problems and the related important components, as discussed by [ABC+06]. We present results based on a generic software library, designed to handle simple mathematical operations to more advanced and distributed computations, using a high abstraction level. The library is referred to as the *GPUlab library*. Our strategy has been to implement a proof-of-concept framework that utilize modern GPUs for parallel computations, in a heterogeneous CPU-GPU hardware setup. Such a hardware setup constitutes what can be considered an affordable standard consumer desktop environment. Therefore these new HPC programming paradigms potentially have a much broader target group than previous HPC software packages.

We present a generic strategy to build and implement software components for the solution of PDEs based on regular and curvilinear structured grids and matrix-free stencil operations. The library is implemented with generic C++ templates to allow a flexible, extensible, and efficient framework to assemble custom PDE solvers. An overview of basic components, essential for the solution of different types of PDEs, is presented, with special emphasis on components that can be used and modified with little or no GPU programming experience. During the duration of this project, several GPU-supporting software libraries and applications with different objectives have emerged. Some libraries integrated support for heterogeneous computing, e.g., PETSc[MSK10, BBB+13, BBB+11] or Matlab. Other libraries emerged as purely GPU-accelerated frameworks; Thrust[BH11], CUSP[BG09], Magma, ArrayFire, pyCUDA, ViennaCL, OpenCurrent, etc.. The GPUlab library falls in the latter category and has similarities to some of these libraries. A similar generic template based approach is also used in the sparse matrix library CUSP, and to some extent in the vector based Thrust library. The GPUlab library derives from Thrust, for easy and portable vector manipulations. Though some of the aforementioned software libraries offer functionalities that matched some of our requirements, we decided to implement our own library, in order to ensure that we would have full control and a deep understanding of the implementations on all levels. Secondly, a high-level library for PDE solvers utilizing some of the generic features available in C++, was not fully developed at the beginning of our research.

As a recurring case study throughout this thesis, we have adopted and continued the development of a fully nonlinear free surface water wave implementation: OceanWave3D. Our work can be seen as a continuation to the work first initiated

by Li & Fleming in 1997 [LF97], as they proposed an efficient geometric multigrid strategy for solving the computationally expensive σ-transformed Laplace equation. A strategy for accurate high-order finite difference discretization and a fourth-order Runge-Kutta scheme was later proposed by Bingham and Zhang in two spatial dimensions [BZ07] and later extended to three dimensions by Engsig-Karup et al. [EKBL09]. The latter demonstrates an alternative use of ghost points to satisfy boundary conditions and proposes to employ multigrid as a precondioner to GMRES to allow efficient higher order discritizations. Further development and changes to the algorithmic strategy with respect to improved parallel feasibility was carried out in [EKMG11], where an efficient single-GPU parallel implementation of a multigrid method for high-order discretizations was proposed, which can be seen as a generalization of the original work due to Li & Fleming[LF97]. As an outcome to the promising efficiency and scalability results presented in [EKMG11], we decided to continue development of the unified free surface model and to port the dedicated solver into the generic GPUlab library to establish a performance portable application with better development productivity, maintenance, and to enable large-scale simulations on heterogeneous hardware systems ranging from desktops to large supercomputers. The redesigned free surface water wave solver has been the basis for much of the research in this thesis, leading up to an industrial collaboration on ship-wave interaction[LGB+12, LGB+13], two book chapters on scientific GPU programming[GEKND13, EKGNL13] two articles [GEKM11, EKMG11], and several conference contributions. The OceanWave3D model has also served as a platform and benchmark application for almost all the software, including library components, that has been developed throughout this PhD project. The mathematical and the numerical model contain several properties and components that are present in many PDE problems in various important engineering applications and they are well-suited for efficient parallel implementations. The template-based GPUlab library has provided us with a basis for improving and extending the free surface water wave implementation with library components that can be reused for the solution of other PDE problems and explore new paradigms for scientific computing and engineering applications.

As an extension to previous work, we present the addition of both spatial and temporal decomposition techniques for fast simulation of large-scale phenomena. We present the extension from a single-block into a multi-block strategy, with automatic memory distribution across multiple GPUs, based on an extensible and generic grid topology. In order to allow local low-storage grid operations in parallel, artificial overlapping boundary layers are introduced and updated via message passing (MPI [GLS99, GLT99]). The challenges of efficient and scalable data distribution and domain decomposition techniques on heterogeneous systems are discussed, where strong scaling is often challenging for PDE problems, as the ratio between surface and volume increases while work per core decreases. We have demonstrated that good weak scaling for large-scale modeling, based

on unified potential flow theory for engineering computations is indeed possible with the recent advancement of high-performance accelerators. With these results we address the limitation observed by P. Lin in 2008 [Lin08], stating that no unified model exists for practical large-scale engineering applications, until future advancement in computer power is made. Thus, present work clearly demonstrates how modern heterogeneous high-performance computing can be utilized to the advancement of numerical modeling of water waves, for accurate wave propagation over varying depths from deep to shallow water. In Chapter 4 we demonstrate examples of such large-scale simulations that can have a great value in a wide range of engineering applications, such as long distance wave propagation over varying depths or within large coastal regions. In the same chapter we present and discuss performance measurements and scalability aspects of using multiple compute devices to speed up the time-to-solution. Numerical experiments based on distributed multi-block computations show that very large-scale simulations are possible on present computer systems, for system sizes in the order of billions of grid points in spatial resolution.

Multiple GPUs have also been utilized to extract parallelism in the time domain for initial value problems, using the parareal algorithm[LMT01, BBM+02, Nie12]. Parareal is implemented as a regular time integrator component into the library, with flexibility to define and configure the integrators for fine and coarse time stepping. Reasonable performance speedups are reported for both a heat conduction problem and for the free surface water wave problem. It has been the first time that a heterogeneous multi-GPU setup has been utilized to solve a free surface problem with temporal parallelization techniques.

A final extension to the library includes routines for curvilinear grid transformations, that allow representation of boundary-fitted geometries. These routines are profiled and demonstrated on a number of cases for the free surface water wave model, i.e., water flow in a circular channel and around offshore monopiles in open water. Such an extension has value in marine engineering as it allows for much more realistic settings and can in combination with the domain decomposition techniques be utilized to reconstruct large maritime areas, such as harbors and shore lines.

Besides the completion of this PhD project and the results presented in this thesis, perhaps the most important contribution that has come out of the project and the development of the GPUlab library, are the possibilities that it has opened for advanced engineering applications. With a thoroughly tested and benchmarked library, researchers and industrial collaborators are now able to benefit from our research, relevant to their own projects or engineering applications. An industrial collaboration with FORCE Technology on real-time ship-wave interaction in full mission marine simulators has already been established and is ongoing. Such a project on real-time simulation with engineering

accuracy shows how HPC on heterogeneous hardware can be used to make a difference in the industry today, and is also an example of how the industry can benefit from the expertise and research that are carried out at the universities. In addition to this collaboration, two student research projects have been able to benefit from the GPUlab library.

1.2.1 Setting the stage 2010 – 2013

Within the last three years, during the time period of this project, there have been some significant changes to the architectural design and the programming guidelines for optimal utilization of the GPU. When GPGPU first became popular, there was a significant difference between single- and double-precision arithmetic performance. Some of the early CUDA-enabled devices did not even support double precision arithmetic. As a consequence, researchers used mixed precision iterative refinement techniques to obtain accurate double precision solutions, with partial use of single-precision operations [GS10, GÏ0, BBD+09]. We also made a contribution using templates to setup a mixed precision version of our solver [GEKM11]. Recent generations of the Nvidia Tesla series for scientific computing have a more balanced ratio between single and double precision performance, so today mixed techniques receive less attention. The first generations of CUDA enabled GPUs had only a small user controllable shared memory cache. Optimal performance for near-neighbor-type operations, e.g., stencil or matrix-like operations, is only achievable if the shared cache is fully utilized[DMV+08, Mic09]. Today there are both an automatic L1 and L2 cache available, significantly reducing the programming effort of implementing device kernels with good performance. Interestingly, the GPU core clock frequency has been almost unchanged during the three years. What has changed is the total number of cores, from a few hundreds to several thousands, e.g. 240 cores in the Tesla C1060 to 2,688 cores in the Tesla K20x. Maintaining core frequency, but increasing the core count has been a deliberate choice from the manufacturers to limit the power consumption. Though the memory bandwidth between the chip and device memory has increased, it has not increased at the same rate as the number of cores. This is a potential bottleneck problem, not only present in GPU computing, but is appointed to be one of the major difficulties in future HPC and a problem that will have to be addressed before we can reach the exascale era[Key11].

During the last few years, improvements have been introduced to address the bottleneck problem of data transfers on multi-GPU systems. Memory access and message passing have been improved via new hardware features, such as increased cache sizes, ECC support, Remote Direct Memory Access (RDMA), and Unified Virtual Address (UVA). In [WPL+11b, WPL+11a] the authors demon-

strate more than 60% latency reduction for exchanging small messages using GPU-Direct with RDMA. In addition, two MPI distributions (MVAPICH2 and OpenMPI) have pushed the lead towards more intuitive device memory transfers, enabling support for direct device memory pointers. This is definitely the road for future heterogeneous multi-device systems, because it enables transparent implementations with high productivity *and* high performance. However, these features are at present limited to specific GPU generations and Infiniband interconnect drivers, that are not standard in many systems.

As a reaction to the emerging HPC market in commodity hardware, Intel proposed an alternative to the GPU, when they launched their Many Integrated Core Architecture (Intel MIC) in 2012. Though the MIC in many aspects is similar to the GPU, it offers some interesting alternatives. Most noticeable is that the MIC processor runs its own operating system, allowing more flexible and direct interaction. The present work was initiated as a research project on GPU programming, and therefore we will only consider this throughout the thesis.

1.3 Hardware resources and GPUlab

Several computer systems have been used for software development and performance measuring during this PhD project. Whenever relevant, we will refer to three of these systems. The first two computers are desktop computers, located at the Technical University of Denmark. They are both equipped with Nvidia GPUs, one with two Tesla K20c GPUs, kindly donated by Nvidia, and one with two GeForce GTX590 GPUs. The third test setup is a GPU cluster, located at Brown University. Each compute node has an Infiniband interconnection equipped with two Tesla M2050 GPUs. Technical details are summarized in Table 1.1.

Name	*G4*	*G6*	*Oscar*
No. Nodes	1	1	44
CPU	Intel Xeon E5620	Intel Core i7-3820	Intel Xeon E5630
Cores	4	4	4
Clock rate	2.40 GHz	3.60 GHz	2.53 GHz
Total memory	12 GB	32 GB	24 GB
GPU†	2 x GeForce GTX590	2 x K20c	2 x Tesla M2050
CUDA driver	5.0	5.0	5.0
CUDA capability	2.0	3.5	2.0
CUDA cores	1024	2496	448
Clock rate	1215 MHz	706 MHz	1150 MHz
Peak performance‡	2488.3 Gflops	3520 Gflops	1030.46 Gflops
Total memory	1.5 GB	5 GB	3 GB
Mem. bandwidth⋆	328 GB/s	208 GB/s	148 GB/s
L2 chache	768 KB	1280 KB	768 KB
Shared mem/block	48 KB	48 KB	48 KB
Registers/block	32768	65536	32768

Table 1.1: Hardware configurations used throughout the thesis. Stats are per GPU. †GTX590 consists of two GTX580 GPUs, e.i. a total of 4 GPUs in *G4*. ‡Single precision arithmetics. ⋆With ECC off.

Chapter 2

Software development for heterogeneous architectures

Massively parallel processors designed for high throughput, such as graphical processing units (GPUs), have in recent years proven to be effective for a vast number of scientific applications. Today, most desktop computers are equipped with one or more powerful GPUs, offering heterogeneous high-performance computing to a broad range of scientific researchers and software developers. Though GPUs are now programmable and can be highly effective computing units, they still pose challenges for software developers to fully utilize their efficiency. Sequential legacy codes are not always easily parallelized, and the time spent on conversion might not pay off. This is particularly true for heterogeneous computers, where the architectural differences between the main and co-processor can be so significant that they require completely different optimization strategies. The cache hierarchy management of CPUs and GPUs is an evident example of this. In the past, industrial companies were able to boost application performance solely by upgrading their hardware systems, with an overt balance between investment and performance speedup. Today, the picture is different; not only do they have to invest in new hardware, but they must also account for the adaption and training of their software developers. What traditionally used to be a hardware problem, addressed by the chip manufacturers, has now become a software problem for application developers.

Software libraries can be a tremendous help for developers as they make it easier to implement an application, without requiring special knowledge of the underlying computer architecture and hardware. A library may be referred to as *opaque* when it automatically utilizes the available resources, without requiring specific details from the developer[ABC+06]. The ultimate goal for a successful library is to simplify the process of writing new software and thus to increase developer productivity. Since programmable heterogeneous CPU/GPU systems are a rather new phenomena, there are only a limited number of established software libraries that take full advantage of such heterogeneous high performance systems, and there are no de facto design standards for such systems either. Some existing libraries for conventional homogeneous systems have already added support for offloading computationally intense operations onto co-processing GPUs. However, this approach comes at the cost of frequent memory transfers across the low bandwidth PCIe bus.

In this chapter, we focus on the use of a software library to help application developers achieve their goals without spending an immense amount of time on optimization details, while still offering close-to-optimal performance. A good library provides performance-portable implementations with intuitive interfaces, that hide the complexity of underlying hardware optimizations. Unfortunately, opaqueness sometimes comes at a price, as one does not necessarily get the best performance when the architectural details are not *visible* to the programmer [ABC+06]. If, however, the library is flexible enough and permits developers to supply their own low-level implementations as well, this does not need to be an issue. These are some of the considerations library developers should take into account, and what we will try to address in this chapter.

For demonstrative purposes we present details from a generic CUDA-based C++ library for fast assembling of partial differential equation (PDE) solvers, utilizing the computational resources of GPUs. This library has been developed as part of research activities associated with the GPUlab, at the Technical University of Denmark and, therefore, is referred to as the *GPUlab library*. It falls into the category of computational libraries, as categorized by Hoefler and Snir [HS11]. Memory allocation and basic algebraic operations are supported via object-oriented components, without the user having to write CUDA specific kernels. As a back-end vector class, the parallel CUDA Thrust template-based library is used, enabling easy memory allocation and a high-level interface for vector manipulation [BH11]. Inspirations for *good library design*, some of which we will present in this chapter, originate from guidelines proposed throughout the literature [HS11, GHJV95, SDB94]. An identification of desirable properties, which any library should strive to achieve, is pointed out by Korson and McGregor [KM92]. In particular we mention being easy-to-use, extensible, and intuitive.

The library is designed to be effective and scalable for fast prototyping of PDE solvers, (primarily) based on matrix-free implementations of finite difference (stencil) approximations on logically structured grids. It offers functionalities that will help assemble PDE solvers that automatically exploit heterogeneous architectures much faster than manually having to manage GPU memory allocation, memory transfers, kernel launching, etc.

In the following sections we demonstrate how software components that play important roles in scientific applications can be designed to fit a simple framework that will run efficiently on heterogeneous systems. One example is finite difference approximations, commonly used to find numerical solutions to differential equations. Matrix-free implementations minimize both memory consumption and memory access, two important features for efficient GPU utilization and for enabling the solution of large-scale problems. The bottleneck problem for many PDE applications is to solve large sparse linear systems, arising from the discretization. In order to help solve these systems, the library includes a set of iterative solvers. All iterative solvers are template-based, such that vector and matrix classes, along with their underlying implementations, can be freely interchanged. New solvers can also be implemented without much coding effort. The generic nature of the library, along with a predefined set of interface rules, allows assembling components into PDE solvers. The use of parameterized-type binding allows the user to assemble PDE solvers at a high abstraction level, without having to change the remaining implementation.

Since this chapter is mostly dedicated to the discussion of software development for high performance heterogeneous systems, the focus will be more on the development and usage of the GPUlab library, than on specific scientific applications. We demonstrate how to use the library on two elementary model problems and refer the reader to Chapter 3 for a detailed description of an advanced application tool for free surface water wave simulations. These examples are assembled using library components similar to those presented in this chapter.

2.1 Heterogeneous library design for PDE solvers

In the following, we present an overview of the library and the supported features, introduce the concepts of the library components, and give short code examples to ease understanding. The library is a starting point for fast assembling of GPU-based PDE solvers, developed mainly to support finite difference operations on regular grids. However, this is not a limitation, since existing vector objects could be used as base classes for extending to other discretization methods or grid types as well.

2.1.1 Component and concept design

The library is grouped into component classes. Each component should fulfill a set of simple interface and template rules, called concepts, in order to guarantee compatibility with the rest of the library. In the context of PDE solving, we present five component classes: vectors, matrices, iterative solvers for linear system of equations, preconditioners for the iterative solvers, and time integrators. Figure 2.1 lists the five components along with a subset of the type definitions they should provide and the methods they should implement. It is possible to extend the implementation of these components with more functionality that relate to specific problems, but this is the minimum requirement for compatibility with the remaining library. With these concept rules fulfilled, components can rely on other components to have their respective functions implemented.

A component is implemented as a generic C++ class, and normally takes as a template arguments of the same types that it offers through type definitions: a matrix takes a vector as template argument, and a vector takes the working precision type. The matrix can then access the working precision through the vector class. Components that rely on multiple template arguments can combine these arguments via type binders to reduce the number of arguments and maintain code simplicity. We will demonstrate use of such type binders in the model problem examples. A thorough introduction to template-based programming in C++ can be found in [VJ02].

The generic configuration allows the developer to define and assemble solver parts at the very beginning of the program using type definitions. Changing PDE parts at a later time is then only a matter of changing type definitions. We will give two model examples of how to assemble PDE solvers in Section 2.2.

2.1.2 A matrix-free finite difference component

Common vector operations, such as memory allocation, element-wise assignments, and basic algebraic transformations, require many lines of codes for a purely CUDA-based implementation. These CUDA-specific operations and kernels are hidden from the user behind library implementations, to ensure a high abstraction level. The vector class inherits from the CUDA-based Thrust library and therefore offers the same level of abstraction that enhances developer productivity and enables performance portability. Creating and allocating device (GPU) memory for two vectors can be done in a simple and intuitive way using the GPUlab library, as shown in Listing 2.1 where two vectors are added together.

Vector

```
typedef value_type;
typedef size_type;

            Vector(size_type);
            Vector(Vector);
void        axpy(value_type,Vector);
void        axpby(value_type,Vector)
    ;
void        copy(Vector);
value_type  dot(Vector);
Vector*     duplicate();
void        fill(value_type);
value_type  nrmi();
value_type  nrm2();
void        scal(vale_type);
size_type   size();
```

Matrix

```
typedef vector_type;

void mult(vector_type,vector_type);
```

Preconditioner

```
typedef vector_type;
typedef matrix_type;
typedef monitor_type;

     Preconditioner(matrix_type
                    ,monitor_type);
void operator()(vector_type
                ,vector_type)
```

EqSolver

```
typedef vector_type;
typedef matrix_type;
typedef monitor_type;
typedef preconditioner_type;

     EqSolver(matrix_type
              ,monitor_type);
void solve(vector_type,vector_type)
    ;
void set_preconditioner(
    preconditioner_type);
```

TimeIntegrator

```
template <typename rhs_type
        , typename vector_type
        , typename value_type>
void operator()(rhs_type
                ,vector_type
                ,value_type
                ,value_type
                ,value_type);
```

Figure 2.1: Schematic representation of the five main components, their type definitions, and member functions. Because components are template based, the argument types cannot be known beforehand. The concepts ensure compliance among components.

```
1  #include <gpulab/vector.h>
2
3  __global__ void add(double* a, double const* b, int N)
4  {
5      int i = blockDim.x*blockIdx.x + threadIdx.x;
6      if(i<N)
7          a[i] += b[i];
8  }
9
10 int main(int argc, char *argv[])
11 {
12     int N = 1000;
13
14     // Basic CUDA example
```

```
15      double *a1, *b1;
16      cudaMalloc((void**)&a1, N*sizeof(double));
17      cudaMalloc((void**)&b1, N*sizeof(double));
18      cudaMemset(a1, 2.0, N);
19      cudaMemset(b1, 3.0, N);
20      int blocksize = 128;
21      add<<<(N+blocksize-1)/blocksize,blocksize>>>(a1, b1, N);
22
23      // gpulab example
24      gpulab::vector<double,gpulab::device_memory> a2(N, 2.0);
25      gpulab::vector<double,gpulab::device_memory> b2(N, 3.0);
26      a2.axpy(1.0, b2); // BLAS1: a2 = 1*b2 + a2
27
28      return 0;
29  }
```

Listing 2.1: Allocating, initializing, and adding together two vectors on the GPU: first example uses pure CUDA C; second example uses the built-in library template-based vector class

The vector class (and derived classes hereof) is compliant with the rest of the library components. Matrix-vector multiplications are usually what makes PDE-based applications different from each other, and the need to write a user specific implementation of the matrix-vector product is essential when solving specific PDE problems. The PDE and the choice of discretization method determine the structure and sparsity of the resulting matrix. Spatial discretization is supported by the library with finite difference approximations, and it offers an efficient, low-storage (matrix-free), flexible order implementation to help developers tailor their custom codes. These matrix-free operators are feasible for problems where the matrix structure is known in advance and can be exploited, such that the matrix values can be either precomputed or computed on the fly. Furthermore, the low constant memory requirement makes them perfect in the context of solving large-scale problems, whereas traditional sparse matrix formats require increasingly more memory, see e.g., [BG09] for details on GPU sparse matrix formats.

Finite differences approximate the derivative of some function $u(x)$ as a weighted sum of neighboring elements. In compact notation we write

$$\frac{\partial^q u(x_i)}{\partial x^q} \approx \sum_{n=-\alpha}^{\beta} c_n u(x_{i+n}), \tag{2.1}$$

where q is the order of the derivative, c_n is a set of finite difference coefficients, and α plus β define the number of coefficients that are used for the approximation. The total set of contributing elements is called the stencil, and the size of the stencil is called the rank, given as $\alpha + \beta + 1$. The stencil coefficients c_n can be derived from a Taylor expansion based on the values of α and β, and q, using the method of undetermined coefficients [LeV07]. An example of a three-point

finite difference matrix that approximates the first ($q = 1$) or second ($q = 2$) derivative of a one-dimensional uniformly distributed vector u of length 8 is given here:

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} & 0 & 0 & 0 & 0 & 0 \\ c_{10} & c_{11} & c_{12} & 0 & 0 & 0 & 0 & 0 \\ 0 & c_{10} & c_{11} & c_{12} & 0 & 0 & 0 & 0 \\ 0 & 0 & c_{10} & c_{11} & c_{12} & 0 & 0 & 0 \\ 0 & 0 & 0 & c_{10} & c_{11} & c_{12} & 0 & 0 \\ 0 & 0 & 0 & 0 & c_{10} & c_{11} & c_{12} & 0 \\ 0 & 0 & 0 & 0 & 0 & c_{10} & c_{11} & c_{12} \\ 0 & 0 & 0 & 0 & 0 & c_{20} & c_{21} & c_{22} \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{bmatrix} \approx \begin{bmatrix} u_0^{(q)} \\ u_1^{(q)} \\ u_2^{(q)} \\ u_3^{(q)} \\ u_4^{(q)} \\ u_5^{(q)} \\ u_6^{(q)} \\ u_7^{(q)} \end{bmatrix}. \quad (2.2)$$

It is clear from this example that the matrix is sparse and that the same coefficients are repeated for all centered rows. The coefficients differ only near the boundaries, where off-centered stencils are used. It is natural to pack this information into a stencil operator that stores only the unique set of coefficients:

$$\mathbf{c} = \begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix}. \quad (2.3)$$

Matrix components precompute these compact stencil coefficients and provide member functions that compute the finite difference approximation of input vectors. Unit scaled coefficients (assuming grid spacing is one) are computed and stored to be accessible via both CPU and GPU memory. On the GPU, the constant memory space is used for faster memory access [Nvi13]. In order to apply a stencil on a non unit-spaced grid, with grid space Δx, the scale factor $1/(\Delta x)^q$ will have to be multiplied by the finite difference sum, i.e., $(c_{00}u_0 + c_{01}u_1 + c_{02}u_2)/(\Delta x)^q \approx u_0^{(q)}$, as in the first row of (2.2).

Setting up a two-dimensional grid of size $N_x \times N_y$ in the unit square and computing the first derivative thereof is illustrated in Listing 2.2. The grid is a vector component, derived from the vector class. It is by default treated as a device object, and memory is automatically allocated on the device to fit the grid size. The finite difference approximation as in (2.1), is performed via a CUDA kernel behind the scenes during the calls to `mult` and `diff_x`, utilizing the memory hierarchy as the CUDA guidelines prescribe [Nvi13, Nvi12b]. To increase developer productivity, kernel launch configurations have default settings, based on CUDA guidelines, principles, and experiences from performance testings, such that the user does not have to explicitly specify them. For problem-specific finite difference approximations, where the built-in stencil operators are insufficient, a pointer to the coefficient matrix (2.3) can be accessed as demonstrated in Listing 2.2 and passed to customized kernels.

```
#include <gpulab/grid.h>
#include <gpulab/FD/stencil.h>

int main(int argc, char *argv[])
{
    // Initialize grid dimensions
    unsigned int Nx = 10,  Ny = 10;
    gpulab::grid_dim<unsigned int> dim(Nx,Ny);
    gpulab::grid<double> u(dim); // 2D function u
    gpulab::grid<double> ux(u);  // 1st order derivative in x
    gpulab::grid<double> uxy(u); // Mixed derivative in x/y

    // Put meaningful values into u here ...

    // Stencil size, alpha=beta=2, 9pt 2D stencil
    int alpha = 2;
    // 1st order derivative
    gpulab::FD::stencil_2d<double> stencil(1, alpha);
    // Calculate uxy = du/dx + du/dy
    stencil.mult(u,uxy);
    // Calculate ux  = du/dx
    stencil.diff_x(u,ux);
    // Host and device pointers to stencil coeffs
    double const* hc = stencil.coeffs_host();
    double const* dc = stencil.coeffs_device();

    return 0;
}
```

Listing 2.2: Two-dimensional finite difference stencil example: computing the first derivative using five points ($\alpha = \beta = 2$) per dimension, a total nine-point stencil

In the following sections we demonstrate how to go from an initial value problem (IVP) or a boundary value problem (BVP) to a working application solver by combining existing library components along with new custom-tailored components. We also demonstrate how to apply spatial and temporal domain decomposition strategies that can make existing solvers take advantage of systems equipped with multiple GPUs. The next section demonstrates how to rapidly assemble a PDE solver using library components. Appendix A contains additional examples and guidelines on how to use the GPUlab library.

2.2 Model problems

We present two elementary PDE model problems, to demonstrate how to assemble PDE solvers, using library components that follow the guidelines described above. The first model problem is the unsteady parabolic heat conduction equation; the second model problem is the elliptic Poisson equation. The two model problems consist of elements that play important roles in solving a broad range of more advanced PDE problems.

We refer the reader to Chapter 3 for an example of a scientific application relevant for coastal and maritime engineering analysis that has been assembled using customized library components similar to those presented in the following.

2.2.1 Heat conduction equation

Firstly, we consider a two-dimensional heat conduction problem defined on a unit square. The heat conduction equation is a parabolic partial differential diffusion equation, including both spatial and temporal derivatives. It describes how the diffusion of heat in a medium changes with time. Diffusion equations are of great importance in many fields of sciences, e.g., fluid dynamics, where the fluid motion is uniquely described by the Navier-Stokes equations, which include a diffusive viscous term [CM93, FP96].

The heat problem is an IVP, it describes how the heat distribution evolves from a specified initial state. Together with homogeneous Dirichlet boundary conditions, the heat problem in the unit square is given as

$$\frac{\partial u}{\partial t} - \kappa \nabla^2 u = 0, \qquad (x, y) \in \Omega([0,1] \times [0,1]), \quad t \geq 0, \tag{2.4a}$$

$$u = 0, \qquad (x, y) \in \partial\Omega, \tag{2.4b}$$

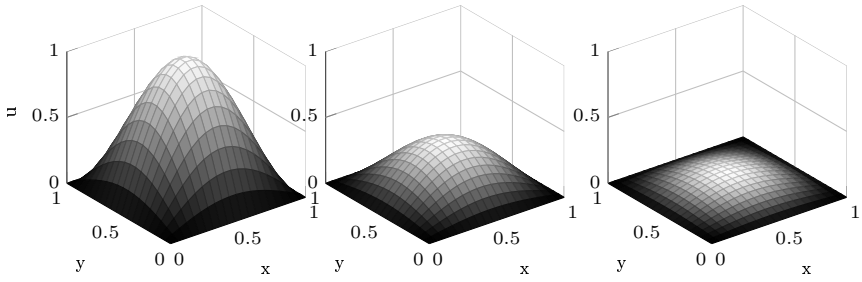
where $u(x, y, t)$ is the unknown heat distribution defined within the domain Ω, t is the time, κ is a heat conductivity constant (let $\kappa = 1$), and ∇^2 is the two-dimensional Laplace differential operator $(\partial_{xx} + \partial_{yy})$. We use the following initial condition:

$$u(x, y, t_0) = \sin(\pi x)\, \sin(\pi y), \qquad (x, y) \in \Omega, \tag{2.5}$$

because it has a known analytic solution over the entire time span, and it satisfies the homogeneous boundary condition given by (2.4b). An illustrative example of the numerical solution to the heat problem, using (2.5) as the initial condition, is given in Figure 2.2.

We use a Method of Lines (MoL) approach to solve (2.4). Thus, the spatial derivatives are replaced with finite difference approximations, leaving only the temporal derivative as unknown. The spatial derivatives are approximated from $\mathbf{u}^n$, where $\mathbf{u}^n$ represents the approximate solution to $u(t_n)$ at a given time t_n with time step size δt such that $t_n = n\delta t$ for $n = 0, 1, \ldots$. The finite difference approximation can be interpreted as a matrix-vector product as sketched in (2.2), and so the semi-discrete heat conduction problem becomes

$$\frac{\partial u}{\partial t} = \mathcal{A}\mathbf{u}, \qquad \mathcal{A} \in \mathbb{R}^{N \times N}, \quad \mathbf{u} \in \mathbb{R}^N, \tag{2.6}$$



(a) $t = 0.00s$ **(b)** $t = 0.05s$ **(c)** $t = 0.10s$

Figure 2.2: Discrete solution, at times $t = 0s$ and $t = 0.05s$, using (2.5) as the initial condition and a small 20×20 numerical grid.

where $\mathcal{A}$ is the sparse finite difference matrix and N is the number of unknowns in the discrete system. The temporal derivative is now free to be approximated by any suitable choice of a time-integration method. The most simple integration scheme would be the first-order accurate explicit forward Euler method,

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \delta t \, \mathcal{A}\mathbf{u}^n, \tag{2.7}$$

where $n + 1$ refers to the solution at the next time step. The forward Euler method can be exchanged with alternative high-order accurate time integration methods, such as Runge-Kutta methods or linear multistep methods, if numerical instability becomes an issue, see, e.g., [LeV07] for details on numerical stability analysis. For demonstrative purposes, we simply use conservative time step sizes to avoid stability issues. However, the component-based library design provides exactly the flexibility for the application developer to select or change PDE solver parts, such as the time integrator, with little coding effort. A generic implementation of the forward Euler method that satisfies the library concept rules is illustrated in Listing 2.3. According to the component guidelines in Figure 2.1, a time integrator is basically a functor, which means that it implements the parenthesis operator, taking five template arguments: a right hand side operator, the state vector, integration start time, integration end time, and a time step size. The method takes as many time steps as necessary to integrate from the start to the end, continuously updating the state vector according to (2.7). Notice, that nothing in Listing 2.3 indicates whether GPUs are used or not. However, it is likely that the underlying implementation of the right hand side functor and the `axpy` vector function, do rely on fast GPU kernels. However, it is not something that the developer of the component has to account for. For this reason, the template-based approach, along with simple interface concepts, make it easy to create new components that will fit well into a generic library.

```
struct forward_euler
{
    template <typename F, typename T, typename V>
    void operator()(F fun, V& x, T t, T tend, T dt)
    {
        V rhs(x);              // Initialize RHS vector
        while(t < tend)
        {
            if(tend-t < dt)
                dt = tend-t;       // Adjust dt for last time step

            (*fun)(t, x, rhs);  // Apply rhs function
            x.axpy(dt,rhs);     // Update stage
            t += dt;            // Next time step
        }
    }
}
```

Listing 2.3: Generic implementation of explicit first-order forward Euler integration

The basic numerical approach to solve the heat conduction problem has now been outlined, and we are ready to assemble the PDE solver.

2.2.1.1 Assembling the heat conduction solver

Before we are able to numerically solve the discrete heat conduction problem (2.4), we need implementations to handle the the following items:

Grid. A discrete numerical grid to represent the two-dimensional heat distribution domain and the arithmetical working precision (32-bit single precision or 64-bit double precision).

RHS. A right-hand side operator for (2.6) that approximates the second-order spatial derivatives (matrix-vector product).

Boundary conditions. A strategy that ensures that the Dirichlet conditions are satisfied on the boundary.

Time integrator. A time integration scheme, that approximates the time derivative from (2.6).

All items are either directly available in the library, or can be designed from components therein. The built-in stencil operator may assist in implementing the matrix-vector product, but we need to explicitly ensure that the Dirichlet boundary conditions are satisfied. We demonstrated in Listing 2.2 how to approximate the derivative using flexible-order finite difference stencils. However,

from (2.4b) we know that boundary values are zero. Therefore, we extend the stencil operator with a simple kernel call that assigns zero to the entire boundary. Listing 2.4 shows the code for the two-dimensional Laplace right-hand side operator. The constructor takes as an argument the stencil half size α and assumes $\alpha = \beta$. Thus, the total two-dimensional stencil rank will be $4\alpha + 1$. For simplicity we also assume that the grid is uniformly distributed, $N_x = N_y$. Performance optimizations for the stencil kernel, such as shared memory utilization, are handled in the underlying implementation, accordingly to CUDA guidelines [Nvi13, Nvi12b]. The macros, `BLOCK1D` and `GRID1D`, are used to help set up kernel configurations based on grid sizes, and `RAW_PTR` is used to cast the vector object to a valid device memory pointer.

```
template <typename T>
__global__ void set_dirichlet_bc(T* u, int Nx)
{
    int i = blockDim.x*blockIdx.x+threadIdx.x;
    if(i<Nx)
    {
        u[i]           = 0.0;
        u[(Nx-1)*Nx+i] = 0.0;
        u[i*Nx]        = 0.0;
        u[i*Nx+Nx-1]   = 0.0;
    }
};

template <typename T>
struct laplacian
{
    gpulab::FD::stencil_2d<T>   m_stencil;

    laplacian(int alpha) : m_stencil(2,alpha) {}

    template <typename V>
    void operator()(T t, V const& u, V & rhs) const
    {
        m_stencil.mult(u,rhs); // rhs = du/dxx + du/dyy

        // Make sure bc is correct
        dim3 block = BLOCK1D(rhs.Nx());
        dim3 grid  = GRID1D(rhs.Nx());
        set_dirichlet_bc<<<grid,block>>>(RAW_PTR(rhs),rhs.Nx());
    }
};
```

Listing 2.4: The right-hand side Laplace operator: the built-in stencil approximates the two dimensional spatial derivatives, while the custom set_dirichlet_bc kernel takes care of satisfying the boundary conditions

With the right-hand side operator in place, we are ready to implement the solver. For this simple PDE problem we compute all necessary initial data in the body of the main function and use the forward Euler time integrator to compute the solution until $t = t_{end}$. For more advanced solvers, a built-in `ode_solver` class is defined that helps take care of initialization and storage of multiple state

variables. Declaring type definitions for all components at the beginning of the main file gives a good overview of the solver composition. In this way, it will be easy to control or change solver components at later times. Listing 2.5 lists the type definitions that are used to assemble the heat conduction solver.

```
typedef double                                value_type;
typedef laplacian<value_type>                 rhs_type;
typedef gpulab::grid<value_type>              vector_type;
typedef vector_type::property_type            property_type;
typedef gpulab::integration::forward_euler    time_integrator_type;
```

Listing 2.5: Type definitions for all the heat conduction solver components used throughout the remaining code

The grid is by default treated as a device object, and memory is allocated on the GPU upon initialization of the grid. Setting up the grid can be done via the property type class. The property class holds information about the discrete and physical dimensions, along with fictitious ghost (halo) layers and periodicity conditions. For the heat conduction problem we use a non periodic domain of size $N \times N$ within the unit square with no ghost layers. Listing 2.6 illustrates the grid assembly.

```
// Setup discrete and physical dimensions
gpulab::grid_dim<int>          dim(N,N,1);
gpulab::grid_dim<value_type>   p0(0,0);
gpulab::grid_dim<value_type>   p1(1,1);
property_type                  props(dim,p0,p1);

// Initialize vector
vector_type                    u(props);
```

Listing 2.6: Creating a two-dimensional grid of size `N` times `N` and physical dimension 0 to 1

Hereafter the vector `u` can be initialized accordingly to (2.5). Finally we need to instantiate the right-hand side Laplacian operator from Listing 2.4 and the forward Euler time integrator in order to integrate from t_0 until t_{end}.

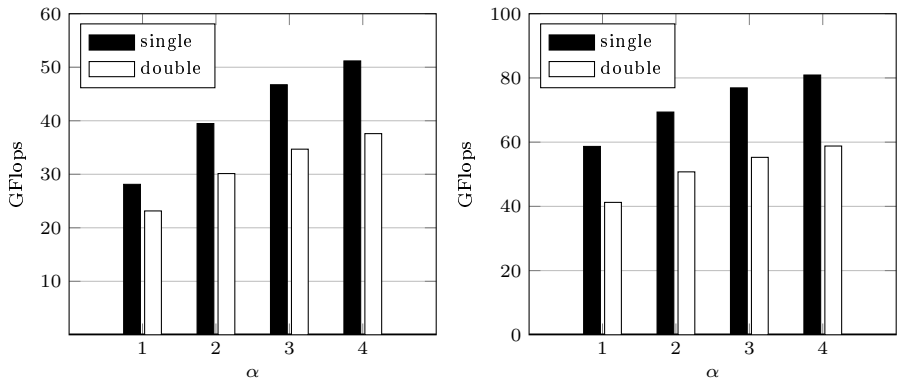
```
rhs_type rhs(alpha);            // Create right-hand side operator
time_integrator_type solver;    // Create time integrator
solver(&rhs,u,0.0f,tend,dt);    // Integrate from 0 to tend using dt
```

Listing 2.7: creating a time integrator and the right-hand side Laplacian operator.

The last line invokes the forward Euler time integration scheme defined in Listing 2.5. If the developer decides to change the integrator into another explicit scheme, only the time integrator type definition in Listing 2.5 needs to be changed. The heat conduction solver is now complete.

2.2.1.2 Numerical solutions to the heat conduction problem

Solution time for the heat conduction problem is in itself not very interesting, as it is only a simple model problem. What is interesting for GPU kernels, such as the finite differences kernel, is that increased computational work often comes with a very small price, because the fast computations can be hidden by the relatively slower memory fetches. Therefore, we are able to improve the accuracy of the numerical solution via more accurate finite differences (larger stencil sizes), while improving the computational performance in terms of floating point operations per second (flops). Figure 2.3 confirms, that larger stencils improve the kernel performance. Notice that even though these performance results are favorable compared to single core systems ($\sim$ 10 GFlops double precision on a 2.5-GHz processor), they are still far from their peak performance, e.g., $\sim$ 2.4 TFlops single precision for the GeForce GTX590. The reason is that the kernel is bandwidth bound, i.e., performance is limited by the time it takes to move memory between the global GPU memory and the chip. The Tesla K20 performs better than the GeForce GTX590 because it obtains the highest bandwidth. Being bandwidth bound is a general limitation for matrix-vector-like operations that arise from the discretization of PDE problems. Only matrix-matrix multiplications, which have a high ratio of computations versus memory transactions, are able to reach near-optimal performance results [KmWH10]. These kinds of operators are, however, rarely used to solve PDE problems.



(a) GeForce GTX590. **(b)** Tesla K20c.

Figure 2.3: Single and double precision floating point operations per second for a two dimensional stencil operator on a numerical grid of size 4096^2. Various stencil sizes are used $\alpha = 1, 2, 3, 4$, equivalent to 5pt, 9pt, 13pt, and 17pt stencils. Host memory transfers are not included in timings.

2.2.2 Poisson equation

The Poisson equation is a second-order elliptic differential equation, often encountered in applications within scientific fields such as electrostatics and mechanics. We consider the two- dimensional BVP defined in terms of Poisson's equation with homogeneous Dirichlet boundary conditions of the form

$$\nabla^2 u = f(x, y), \qquad (x, y) \in \Omega([0, 1] \times [0, 1]), \tag{2.8a}$$

$$u = 0, \qquad (x, y) \in \partial\Omega. \tag{2.8b}$$

Notice the similarities to the heat conduction equation (2.4). In fact, (2.8) could be a steady-state solution to the heat equation, when there is no temporal change $\frac{\partial u}{\partial t} = 0$, but a source term $f(x, y)$. Since the Laplace operator and the boundary conditions are the same for both problems, we are able to reuse the same implementation with few modifications.

In contrast to the heat equation, there are no initial conditions. Instead, we seek some $u(x, y)$ that satisfies (2.8), given a source term $f(x, y)$, on the right-hand side. For simplicity, assume that we know the exact solution, u_{true}, corresponding to (2.5). Then we use the method of manufactured solutions to derive an expression for the corresponding right-hand side $f(x, y)$:

$$f(x, y) = \nabla^2 u_{\text{true}} = -2\pi^2 \sin(\pi x) \sin(\pi y). \tag{2.9}$$

The spatial derivative in (2.8) is again approximated with finite differences, similar to the example in (2.2), except boundary values are explicitly set to zero. The discrete form of the system can now be written as a sparse linear system of equations:

$$\mathcal{A}\mathbf{u} = \mathbf{f}, \qquad \mathbf{u}, \mathbf{f} \in \mathbb{R}^N, \quad \mathcal{A} \in \mathbb{R}^{N \times N}, \tag{2.10}$$

where $\mathcal{A}$ is the sparse matrix formed by finite difference coefficients, N is the number of unknowns, and $\mathbf{f}$ is given by (2.9). Equation (2.10) can be solved in numerous ways, but a few observations may help do it more efficiently. Direct solvers based on Gaussian elimination are accurate and use a finite number of operations for a constant problem size. However, the arithmetic complexity grows with the problem size by as much as $\mathcal{O}(N^3)$ if the sparsity of $\mathcal{A}$ is not exploited. Direct solvers are therefore mostly feasible for dense systems of limited sizes. Sparse direct solvers exist, but they are often difficult to parallelize, or applicable for only certain types of matrices. Regardless of the discretization technique, the discretization of an elliptic PDE into a linear system as in (2.10) yields a very sparse matrix $\mathcal{A}$ when N is large. Iterative methods for solving large sparse linear systems find broad use in scientific applications, because they require only an implementation of the matrix-vector product, and they

often use a limited amount of additional memory. Comprehensive introductions to iterative methods may be found in any of [Saa03, Kel95, BBC+94].

One benefit of the high abstraction level and the template-based library design is to allow developers to implement their own components, such as iterative methods for solving sparse linear systems. The library includes three popular iterative methods: conjugate gradient, defect correction, and geometric multigrid. The conjugate gradient method is applicable only to systems with symmetric positive definite matrices. This is true for the two-dimensional Poisson problem, when it is discretized with a five-point finite difference stencil, because then there will be no off-centered approximations near the boundary. For high-order approximations ($\alpha > 1$), we use the defect correction method with multigrid preconditioning. See e.g., [TOS+01] for details on multigrid methods.

We will not present the implementation details for all three methods but briefly demonstrate the simplicity of implementing the body of such an iterative solver, given a textbook recipe or mathematical formulation. The defect correction method iteratively improves the solution to $\mathcal{A}\mathbf{x} = \mathbf{b}$, given an initial start guess $\mathbf{x}^0$, by continuously solving a preconditioned error equation. The defect correction iteration can be written as

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \mathcal{M}^{-1}(\mathbf{b} - \mathcal{A}\mathbf{x}^k), \quad \mathcal{A}, \mathcal{M} \in \mathbb{R}^{N\times N}, \quad \mathbf{x}, \mathbf{b} \in \mathbb{R}^N, \tag{2.11}$$

where k is the iteration number and $\mathcal{M}$ is the preconditioner which should be an approximation to the original coefficient matrix $\mathcal{A}$. To achieve fast numerical convergence, applying the preconditioner should be a computationally inexpensive operation compared to solving the original system. How to implement (2.11) within the library context is illustrated in Listing 2.8. The host CPU traverses each line in Listing 2.8 and tests for convergence, while the computationally expensive matrix-vector operation and preconditioning, can be executed on the GPU, if GPU-based components are used. The defect correction method has two attractive properties. First, global reduction is required to monitor convergence only once per iteration, during convergence evaluation, which reduces communication requirements and provides a basis for efficient and scalable parallelization. Second, it has a minimal constant memory footprint, making it a suitable method for solving very large systems.

```
1  while(r.nrm2() > tol)
2  {
3      // Calculate residual
4      A.mult(x,r);
5      r.axpby(1, -1, b);
6
7      // Reset initial guess
8      d.fill(0);
9
10     // Solve M*d=r
11     M(d,r);
12
```

```
13      // Defect correction update
14      x.axpy(1, d);
15  }
```

Listing 2.8: Main loop for the iterative defect correction solver: the solver is instantiated with template argument types for the matrix and vector classes, allowing underlying implementations to be based on GPU kernels

In the following section we demonstrate how to assemble a solver for the discrete Poisson problem, using one of the three iterative methods to efficiently solve (2.10).

2.2.2.1 Assembling the Poisson solver

Assembling the Poisson solver follows almost the same procedure as the heat conduction solver, except the time integration part is exchanged with an iterative method to solve the system of linear equations (2.10). For the discrete matrix-vector product we reuse the Laplace operator from the heat conduction problem in Listing 2.4 with few modifications. The Laplace operator is now a matrix component, so to be compatible with the component interface rules in Figure 2.1, a `mult` function taking two vector arguments is implemented instead of the parentheses operator. We leave out this code example as it almost identical to the one in Listing 2.4.

At the beginning of the solver implementation we list the type definitions for the Poisson solver that will be used throughout the implementation. Here we use a geometric multigrid method as a preconditioner for the defect correction method. Therefore the multigrid solver is assembled first, so that it can be used in the assembling of the defect correction solver. Listing 2.9 defines the types for the vector, the matrix, the multigrid preconditioner, and the defect correction solver. The geometric multigrid method needs two additional template arguments that are specific for multigrid, namely, a smoother and a grid restriction/interpolation operator. These arguments are free to be implemented and supplied by the developer if special care is required, e.g., for a custom grid structure. For the Poisson problem on a regular grid, the library contains built-in restriction and interpolation operators, and a red-black Gauss-Seidel smoother. We refer the reader to [TOS+01] for extensive details on multigrid methods. The monitor and config types that appear in Listing 2.9 are used for convergence monitoring within the iterative solver and to control runtime parameters, such as tolerances and iteration limits.

```
1  typedef double                                  value_type;
```

```
typedef gpulab::grid<value_type>                    vector_type;
typedef laplacian<vector_type>                      matrix_type;

// MULTIGRID solver types
typedef gpulab::solvers::multigrid_types<
      vector_type
    , matrix_type
    , gpulab::solvers::gauss_seidel_rb_2d
    , gpulab::solvers::grid_handler_2d>              mg_types;
typedef gpulab::solvers::multigrid<mg_types>        mg_solver_type;
typedef mg_solver_type::monitor_type                monitor_type;
typedef monitor_type::config_type                   config_type;

// DC solver types
typedef gpulab::solvers::defect_correction_types<
      vector_type
    , matrix_type
    , monitor_type
    , mg_solver_type>                                   dc_types;
typedef gpulab::solvers::defect_correction<dc_types> dc_solver_type;
```

Listing 2.9: Type definitions for the Laplacian matrix component and the multigrid preconditioned iterative defect correction solver

With the type definitions set up, the implementation for the Poisson solver follows in Listing 2.10. Some of the initializations are left out, as they follow the same procedure as for the heat conduction example. The defect correction and geometric multigrid solvers are initialized and then multigrid is set as a preconditioner to the defect correction method. Finally the system is solved via a call to `solve()`.

```
matrix_type A(alpha);                // High-order matrix
matrix_type M(1);                    // Low-order matrix

/* Omitted: create and init vectors u, f */

config_type config;                  // Create configuration
config.set("iter",30);               // Set max iteration count
config.set("rtol",1e-10);            // Set relative tolerance
monitor_type monitor(config);        // Create monitor
dc_solver_type solver(A,monitor);    // Create DC solver
mg_solver_type precond(M,monitor);   // Create MG preconditioner
solver.set_preconditioner(precond); // Set preconditioner
solver.solve(u,f);                   // Solve M^-1(Au = f)
if(monitor.converged())
  printf("SUCCESS\n");
```

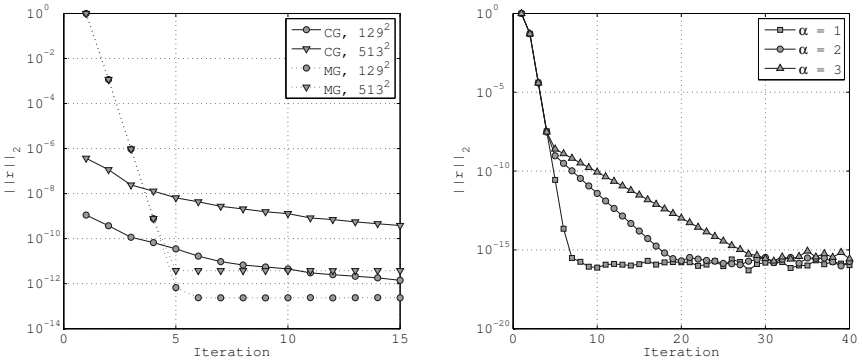
Listing 2.10: Initializing the preconditioned defect correction solver to approximate the solution to $\mathcal{A}\mathbf{u} = \mathbf{f}$

2.2.2.2 Numerical solutions to the Poisson problem

The discrete Poisson problem (2.10) has been solved using the three iterative methods presented above. Convergence histories for the conjugate gradient

method and geometric multigrid method, using two different resolutions, are illustrated in Figure 2.4a. Multigrid methods are very robust and algorithmically efficient, independent of the problem size. Figure 2.4a confirms that the rate of convergence for the multigrid method is unchanged for both problem sizes. Only the attainable accuracy is slightly worsened, as a consequence of a more ill-conditioned system for large problem sizes.

Defect correction in combination with multigrid preconditioning enables efficient solution of high-order approximations of the Poisson problem, illustrated in Figure 2.4b. The multigrid preconditioning matrix $\mathcal{M}$ is based on a low-order approximation to (2.10), whereas matrix $\mathcal{A}$ is a high-order approximation. When $\mathcal{M}$ is a close approximation to $\mathcal{A}$, defect correction converges most rapidly. This is the effect that can be seen between the three convergence lines in Figure 2.4b.



(a) Convergence histories for the conjugate gradient (CG) and multigrid (MG) methods, for two different problem sizes.

(b) Defect correction convergence history for three different stencil sizes.

Figure 2.4: Algorithmic performance for the conjugate gradient, multigrid, and defect correction methods, measured in terms of the relative residual per iteration.

2.3 Multi-GPU systems

CUDA-enabled GPUs are optimized for high memory bandwidth and fast on-chip performance. However, the role as a separate co-processor to the CPU can be a limiting factor for large-scale scientific applications, because the GPU memory capacity is fixed and is only in the range of a few Gigabytes. In comparison,

it is not unusual for a high-end workstation to be equipped with $\sim$ 32GB of main memory, plus a terabyte hard disk capacity for secondary storage. Therefore, large-scale scientific applications that process Gigabytes of data, require distributed computations on multiple GPU devices. Multi-GPU desktop computers and clusters can have a very attractive peak performance, but the addition of multiple devices introduces the potential performance bottleneck of slow data transfers across PCIe busses and network interconnections. The ratio between data transfers and computational work has a significant impact on the possibility for latency hiding, and thereby on overall application performance.

Developing software that exploit the full computational capabilities of modern clusters, GPU-based or not, is no trivial matter. Developers are faced with the complexity of distributing and coordinating computations on nodes consisting of many-core CPUs, GPUs and potentially other types of accelerators as well. These complexities give rise to challenges in finding numerical algorithms, that are well suited for such systems, forcing developers to search for novel methods that utilize concurrency. In Chapter 4 and 5 we address some of the difficulties in designing software for distributed systems and demonstrate both spatial and temporal decomposition techniques as a means for enhanced performance throughput and large-scale simulation.

Chapter 3

Free surface water waves

Applications for hydrodynamic simulations are used in many areas of coastal and offshore engineering. Most models today are based on Boussinesq-type formulations, where the three-dimensional unified potential flow problem is simplified analytically to a two-dimensional problem under the assumption that the vertical profile has polynomial variation. Much research has been done on Boussinesq-type models for several decades, since the original work of Peregrine [Per67] and later pioneered by Abott et al. [APS78, AMW84]. More recent research focuses on higher-order formulations that more accurately capture the effects of fully nonlinear and dispersive water waves traveling over varying depths to far offshore. These formulations tend to introduce numerical difficulties due to the higher order derivatives.

The quality and application range of hydrodynamic simulations based on Boussinesq-type formulations are traditionally evaluated against the more accurate potential flow theory, which has been perceived as too computationally expensive for practical engineering applications [Lin08]. Potential flow theory requires the solution of a Laplace problem arising from the fully three-dimensional problem. An efficient scalable strategy for solving the transformed Laplace problem, based on low-order finite difference discretization and geometric multigrid, was first proposed by Li and Fleming in 1997 [LF97]. Their strategy led the way for more accurate and efficient methods, as it was still unsuitable for large-scale engineering applications.

Our work takes offset in the development and improvements to the strategy originally proposed by Li and Fleming. The strategy extended in [BZ07] with high-order discretizations via high-order numerics and vertical grid clustering to enable accurate and efficient solutions in two dimensions. A flexible-order discretization for the three-dimensional problem was then proposed by [EKBL09], with a more robust discretization strategy and a Laplace solver based on multigrid preconditioned GMRES. The total strategy proposed by [EKBL09] led to the first Fortran90 implementation of what was then referred to as the *OceanWave3D* model[1].

The development of new massively parallel high-performance commodity hardware, is a perfect driver for developing and revisiting these techniques for efficient simulation of maritime engineering applications. In [EKMG11], we implemented a dedicated massively parallel solver based on flexible-order discretization and a multigrid preconditioned defect correction method, and we demonstrated that it is indeed possible to significantly reduce the barriers for practical use of potential flow theory as a modeling basis for hydrodynamic simulations. To establish the model as an efficient parallel tool, the entire algorithm was first redesigned, implemented, and evaluated using a dedicated solver based on the CUDA for C programming model. The CUDA for C extension was used because no extensions were available for Fortran90 at the time of the proof-of-concept. The promising intermediate performance results convinced us to continue the development of the GPU-accelerated OceanWave3D solver and to implement the solver using our generic library in order to enhance portability and productivity of future development. The OceanWave3D model, based on unified potential flow theory, is a relevant modeling basis, because it emphasizes some of the new engineering possibilities that modern affordable hardware brings. In addition, a potential flow model is a good test bed for prototyping and designing software algorithms for PDE solvers, as it contains components that are essential in a broad range of scientific applications, such as a computationally demanding elliptic problem along with hyperbolic free surface conditions. When this work began, there were few engineering applications that utilized massively parallel heterogeneous systems. Our focus is to try and demonstrate by a proof-of-concept some of the possibilities that these modern programming paradigms can offer.

[1] Development and research based on the OceanWave3D model are currently employed by researches at the Department of Mechanical Engineering and the Department of Applied Mathematics and Computer Science at the Technical University of Denmark.

3.1 Potential flow theory

The unified potential flow equations are a subclass of the more complete Euler equations, and are valid only under the assumption of incompressible, irrotational, and inviscid flow. A comprehensive introduction to hydrodynamics is given by Svendsen and Jonnson [SJ76]. Details on numerical modeling aspects can be found in e.g., [Lin08] or in the shorter survey by Dias and Bridges [DB06]. A variety of numerical techniques for solving such hydrodynamic model problems have been presented throughout the literature, see e.g., [Yeu82, LF97, EK06]. A concise derivation of the the fully nonlinear water wave model based on potential flow theory is presented in [EKGNL13]. In this section we intend only to present the governing equations as they will be used in the remainder of this work and we refer the reader to the above mentioned literature for a more thorough introduction.

Under the assumption of irrotational flow, the velocity vector field $\mathbf{u}(x, y, z) \equiv (u,\, v,\, w)^T$, is uniquely defined by the gradient of the velocity potential $\phi(x, y, z)$, such that $\mathbf{u} \equiv (\partial_x\phi,\, \partial_y\phi,\, \partial_z\phi)^T$. The evolution of the free surface elevation η, and the potential $\tilde{\phi}$ at the free surface, are described by a kinematic and a dynamic free surface boundary condition

$$\partial_t\eta = -\,\boldsymbol{\nabla}\eta\cdot\boldsymbol{\nabla}\tilde{\phi} + \tilde{w}(1+\boldsymbol{\nabla}\eta\cdot\boldsymbol{\nabla}\eta), \tag{3.1a}$$

$$\partial_t\tilde{\phi} = -\,g\eta - \frac{1}{2}(\boldsymbol{\nabla}\tilde{\phi}\cdot\boldsymbol{\nabla}\tilde{\phi} - \tilde{w}^2(1+\boldsymbol{\nabla}\eta\cdot\boldsymbol{\nabla}\eta)), \tag{3.1b}$$

where $\boldsymbol{\nabla}$ is the horizontal differential operator $\boldsymbol{\nabla} \equiv (\partial_x\, \partial_y)^T$, g is the gravitational acceleration, and $\tilde{w}$ is the vertical velocity evaluated at the free surface $\tilde{w} \equiv \partial_z\phi|_{z=\eta}$.

The unsteady free surface elevation $\eta(\mathbf{x}, t)$ is measured from the still-water level at $z = 0$. A conceptual illustration of a two-dimensional numerical wave tank with a free surface water wave setup is given in Figure 3.1.

The vertical velocity at the surface $\tilde{w}$, can be computed from the full velocity potential. Due to mass continuity, the velocity potential satisfies the Laplace equation everywhere inside the domain. Along with boundary conditions the following (elliptic)

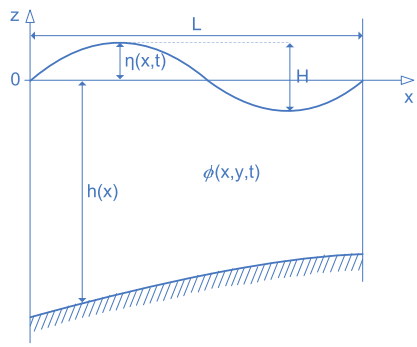


Figure 3.1: Setup of a free surface water wave model.

Laplace equation uniquely defines the velocity potential

$$\phi = \tilde{\phi}, \quad z = \eta \tag{3.2a}$$

$$\boldsymbol{\nabla}^2\phi + \partial_{zz}\phi = 0, \quad -h \leq z < \eta \tag{3.2b}$$

$$\partial_z\phi + \boldsymbol{\nabla}h \cdot \boldsymbol{\nabla}\phi = 0, \quad z = -h, \tag{3.2c}$$

where h is the still water depth. The system of equations is closed by imposing proper boundary conditions on the outer boundaries $\partial\Omega$, surrounding the domain of interest. In the vertical direction, the flow is bounded at the bottom $z = -h$, by the kinematic free-slip condition (3.2c) and at the surface $z = \eta$, with Dirichlet conditions given by the time-dependent dynamic free surface condition(3.1b).

Along the vertical boundaries we assume only bottom-mounted, vertical surface-piercing, and fully reflecting (impermeable) walls with free slip boundary conditions. In this case all outward pointing boundary normals are fixed to the horizontal plane, e.i. $\mathbf{n} \equiv (n_x, n_y, 0)^T$, and the fluid flows only along the tangential direction of the walls,

$$\begin{aligned} \mathbf{n} \cdot \mathbf{u} &= n_x\, u + n_y\, v \\ &= n_x\, \partial_x\phi + n_y\, \partial_y\phi = 0, \quad \forall \mathbf{x} \in \partial\Omega. \end{aligned} \tag{3.3}$$

Wave generation and wave absorption are considered in Section 3.2.4, where we find that with impermeable boundaries in combination with sufficiently large generation and absorption zones, the absence of radiation boundaries is not significant. Thus, we are able to reuse the boundary condition (3.3), in combination with wave generation and absorption zones.

Under some circumstances, when wave amplitudes are small and dispersive wave effects are minor , i.e., $\nabla\eta(\mathbf{x}, t) \approx \mathbf{0}, \forall \mathbf{x}, t$, the nonlinear free surface equations can be reduced to linear form,

$$\partial_t\eta = \tilde{w}, \tag{3.4a}$$

$$\partial_t\tilde{\phi} = -\, g\eta. \tag{3.4b}$$

The linear free surface conditions are useful for stability analysis and validation cases where analytic solutions are known. We emphasize that when we refer to the linear or nonlinear system, it refers to the PDE, and not the Laplace problem, which is always a linear system of equations.

In free surface water wave modeling it is often convenient to use dimensionless numbers for comparison and characterization of various waves. The wave number k is given by $k \equiv 2\pi/L$, where L is the wavelength. Also the wave number

times the still water depth kh, is used to determine if the wave is propagating in a deep or a shallow water situation. The wave steepness S is defined as the ratio between wave height and wavelength $S \equiv H/L$. Sometimes we refer to wave steepness in terms of maximum attainable value, i.e., $H/L = 50\%(H/L)_{max}$. For a thorough introduction to wave characterization we refer the reader to [SJ76].

3.2 The numerical model

Numerical solutions to potential flow model problems are attractive to gain unique insight due to the underlying accurate properties. To ensure practical solutions of large-scale problems based on potential flow theory, it is of great importance that the numerical strategy is based on efficient, scalable, distributed, data-local, and parallel methods. We use a finite difference discretization for approximating the spatial derivatives in (3.1) and (3.2), based on the generic matrix-free implementation presented in Section 2.1.2. The order of accuracy can be controlled via a flexible order implementation, which allow the user to control the size of the discretization stencils. Finite difference methods are among the simplest and best performing discretization methods, due to high data-locality (at least along one dimension) and the structured domains, which require no index lookup table.

In the context of GPUs, the above mentioned numerical method translates well into the GPU programming model, because the computation of each finite difference approximation per grid point is embarrassingly parallel, since no communication between threads is required. Furthermore, structured grids are easily represented in memory and allow parallel execution with efficient collocated memory access patterns, such that adjacent threads also access adjacent memory locations.

3.2.1 Efficient solution of the Laplace equation

The Laplace problem (3.2) with boundary conditions is a fully three-dimensional problem, whereas the free surface boundary conditions (3.1a) and (3.1b) are only two-dimensional. Computing the solution to the discrete Laplace problem therefore involves significantly more computational work, which makes it the performance bottleneck. We have implemented and analyzed a scalable and low-memory iterative strategy for efficiently solving the Laplace problem on heterogeneous hardware in [EKMG11, GEKM11]. The numerical strategy is

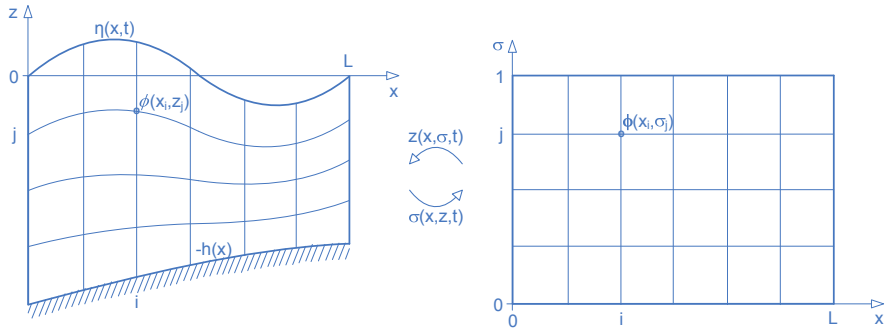


Figure 3.2: Vertical σ-transormation.

briefly outlined in the following sections.

From a numerical point of view it is impractical that the shape of the physical domain changes due to the time-dependent free surface conditions. A σ-transformation is therefore applied in the vertical direction such that

$$\sigma(\mathbf{x}, z, t) = \frac{z + h(\boldsymbol{x})}{d(\boldsymbol{x}, t)}, \quad 0 \leq \sigma \leq 1, \tag{3.5}$$

where $d(\boldsymbol{x}, t) = \eta(\boldsymbol{x}, t) + h(\boldsymbol{x})$ is the total water depth from the sea bottom to the water surface. This enables a transformation of the physical problem into a time-invariant computational domain, at the expense of time-varying coefficients and mixed derivatives. The σ-transformation is illustrated in Figure 3.2. Once the σ-transformation is applied, the original Laplace problem (3.2) is transformed as well,

$$\Phi = \tilde{\phi}, \quad \sigma = 1, \tag{3.6a}$$

$$\nabla^2\Phi + \nabla^2\sigma(\partial_\sigma\Phi) + 2\nabla\sigma \cdot \nabla(\partial_\sigma\Phi) + (\nabla\sigma \cdot \nabla\sigma + (\partial_z\sigma)^2)\partial_{\sigma\sigma}\Phi = 0, \quad 0 \leq \sigma < 1, \tag{3.6b}$$

$$\mathbf{n} \cdot (\nabla, \partial_z\sigma\partial_\sigma)\Phi = 0, \quad (\mathbf{x}, \sigma) \in \partial\Omega, \tag{3.6c}$$

where the transformed velocity potential $\Phi(\boldsymbol{x}, \sigma, t) = \phi(\boldsymbol{x}, z, t)$ holds all information about the flow kinematics in the entire fluid volume. The spatial derivatives

of σ appearing in (3.6) can be directly derived from (3.5) as

$$\boldsymbol{\nabla}\sigma = \frac{1-\sigma}{d}\boldsymbol{\nabla}h - \frac{\sigma}{d}\boldsymbol{\nabla}\eta, \tag{3.7a}$$

$$\boldsymbol{\nabla}^2\sigma = \frac{1-\sigma}{d}\left(\boldsymbol{\nabla}^2 h - \frac{\boldsymbol{\nabla}h\cdot\boldsymbol{\nabla}h}{d}\right) - \frac{\sigma}{d}\left(\boldsymbol{\nabla}^2\eta - \frac{\boldsymbol{\nabla}\eta\cdot\boldsymbol{\nabla}\eta}{d}\right) - \frac{1-2\sigma}{d^2}\boldsymbol{\nabla}h\cdot\boldsymbol{\nabla}\eta - \frac{\boldsymbol{\nabla}\sigma}{d}\cdot(\boldsymbol{\nabla}h+\boldsymbol{\nabla}\eta), \tag{3.7b}$$

$$\partial_z\sigma = \frac{1}{d}. \tag{3.7c}$$

All of these σ-coefficients can be computed using finite difference approximations with the same accuracy as the remaining approximations, given the known free surface elevation and bottom function at any given instance of time. Under the assumption of linearized free surface conditions, $\nabla\eta(\mathbf{x},t) \approx \mathbf{0}, \forall \mathbf{x}, t$, the transformed Laplace problem and the transformation coefficient simplifies as well.

We use flexible-order finite difference stencils to discretize the spatial derivatives in (3.6) as well. All derivatives are computed from one dimensional approximations, using a controllable number of adjacent grid points to meet the desired order of accuracy. We use a matrix-free implementation to exploit that only a few different finite difference coefficients are required. This strategy significantly reduces the memory consumption, as the matrix-free operator uses a constant amount of memory, independent of the problem size. When the spatial derivatives in (3.6) are approximated with finite differences, the transformed Laplace problem can be written as a linear system of equations,

$$\mathcal{A}\boldsymbol{\Phi} = \mathbf{b}, \quad \mathcal{A} \in \mathbb{R}^{N\times N}, \quad \boldsymbol{\Phi}, \mathbf{b} \in \mathbb{R}^N, \tag{3.8}$$

where N is the product of the number of discrete grid points in each dimension N_x, N_y, and N_z, respectively. This discrete system of equations is fully coupled, sparse and non-symmetric. The matrix-free stencil operator based on flexible-order finite difference presented in Section 2.1.2 is utilized to create a custom matrix component that computes the matrix-vector product in (3.8).

3.2.2 Preconditioned defect correction method

In [LF01], the discrete Laplace problem (3.8) was discretized using low-order finite differences and solved with a geometric multigrid method. The idea of multigrid methods as proposed by Brandt in 1977[Bra77] has since proven to be among the most efficient iterative methods to solve sparse linear systems arising from the discretization of partial differential equations. Multigrid methods have

successfully been used to solve a broad range of applications [TOS+01, BHM00]. Multigrid methods possess the very attractive feature, that the computational time to satisfy a given convergence criterion is linearly proportional to the number of unknowns. Unfortunately geometric multigrid is not directly applicable for computing solutions based on high-order discretizations due to stability issues [TOS+01].

In recent work we have proposed a preconditioned iterative defect correction method (PDC), to allow high-order accurate discretization with a constant low-storage overhead and scalable work effort as outlined in Section 2.2.2. Instead of solving (3.8), we solve the left-preconditioned system

$$\mathcal{M}^{-1}\mathcal{A}\mathbf{\Phi} = \mathcal{M}^{-1}\mathbf{b}, \quad \mathcal{M}^{-1} \in \mathbb{R}^{N\times N}, \tag{3.9}$$

Where $\mathcal{M}$ is a preconditioner, such that $\mathcal{M}^{-1} \approx \mathcal{A}^{-1}$ can be computed at a low computational cost. The preconditioned defect correction method then generates a sequence of iterates, starting from an initial guess $\mathbf{\Phi}^{[0]}$ and continues,

$$\mathbf{\Phi}^{[k]} = \mathbf{\Phi}^{[k-1]} + \mathcal{M}^{-1}\mathbf{r}^{[k-1]}, \quad \mathbf{r}^{[k-1]} = \mathbf{b} - \mathcal{A}\mathbf{\Phi}^{[k-1]}, \quad k = 1, 2, \ldots, \tag{3.10}$$

until a given convergence criteria is satisfied. We can also formulate the defect correction method as a stationary iterative method, in terms of the time-invariant iteration matrix $\mathcal{G}$,

$$\begin{aligned}
\mathbf{\Phi}^{[k]} &= \mathbf{\Phi}^{[k-1]} + \mathcal{M}^{-1}\mathbf{r}^{[k-1]} && (3.11)\\
&= \mathbf{\Phi}^{[k-1]} + \mathcal{M}^{-1}(\mathbf{b} - \mathcal{A}\mathbf{\Phi}^{[k-1]}) && (3.12)\\
&= \underbrace{(\mathcal{I} - \mathcal{M}^{-1}\mathcal{A})}_{\mathcal{G}}\mathbf{\Phi}^{[k-1]} + \underbrace{\mathcal{M}^{-1}\mathbf{b}}_{\mathbf{c}} && (3.13)\\
&= \mathcal{G}\mathbf{\Phi}^{[k-1]} + \mathbf{c}, && (3.14)
\end{aligned}$$

in which the iteration matrix $\mathcal{G}$ can be used for convergence analysis. The choice of the preconditioner $\mathcal{M}$, and therefore $\mathcal{G}$, is important for the convergence of the defect correction method, and has been analyzed in the context of free surface water waves in [EK14]. The preconditioner $\mathcal{M}$ is chosen as a coarse grid solver based on nested grids to form a geometric multigrid preconditioner. In previous work it has been demonstrated that $\mathcal{M}$ can be based on a linearized second-order finite difference approximation, for efficient solution of the fully nonlinear system in (3.6), as first used in [EKBL09]. The second-order approximation and linearization reduce the computational work of computing the residuals in the preconditioning phase. A textbook recipe for the defect correction method, preconditioned with one geometric multigrid V-cycle is presented in in Table 3.1.

The most important subroutines are outlined in Table 3.1 and will be profiled in Section 3.4. The number of multigrid levels can be controlled with the value of

PDC$(\mathcal{A}, \mathbf{x}, \mathbf{b})$

```
i = 0
r = b − Ax            // Residual
while ||r|| > τ and i < i_max
  GMG(M, d, r)        // Precondition
  x = x + d           // Correct
  r = b − Ax          // Residual
  i = i + 1
end while
```

GMG$(\mathcal{M}, \mathbf{x}, \mathbf{b})$

```
k = K                 // User defined
VCYCLE(k, M, x, b)
```

VCYCLE$(k, \mathcal{M}, \mathbf{x}, \mathbf{b})$

```
if k == 0
  Smooth(x, b) // Coarse-smooth
else
  Smooth(x, b)        // Pre-smooth
  r = b − Mx          // Residual
  Restrict(r, d)
  VCYCLE(k − 1, M, e, d)
  Prolongate(e, r)
  x = x + r           // Correct
  Smooth(x, b)        // Post-smooth
end if
```

Table 3.1: Recipe for the iterative defect correction (PDC) method, preconditioned with a geometric multigrid (GMG) V-cycle using K coarsenings.

K and it will influence the algorithmic convergence rate. On the coarsest level either a direct solve or multiple smoothings should be performed. In order to reuse library components we perform a controllable number of smoothings on the coarsest grid level. For good numerical and algorithmic convergence it is advised to restrict until the coarsest level[TOS+01]. However, coarse grid levels reduce the parallel performance, as the ratio between the number of parallel threads and discrete grid points become less favorable. In Section 4.4.4 we demonstrate, based on numerical experiments, that there is a less strict requirement for solving the Laplace problem arising from the discretization of a water wave problem. There are several smoothing (or relaxation) techniques to choose from. Unless otherwise stated, we will use a Red-Black Z-line Gauss-Seidel smoother based on a modified Thomas algorithm, see [EKMG11] for details. This smoothing strategy in combination with semi-coarsening has proven to be very algorithmic effective for both deep and shallow water problems and it can be implemented to run efficiently in parallel as a per-thread line solver.

The PDC and GMG algorithms are implemented into our library in the same generic way as previously described. Building the solver using a predefined type binder class could look as follows, assuming that proper types for the vector, matrix, and preconditioner are set beforehand.

```
// Defect Correction type binder
typedef solvers::defect_correction_types<
      vector_type
    , matrix_type
    , preconditioner_type>                    dc_types;
typedef solvers::defect_correction<dc_types> dc_solver_type;

// Create solver, assume vectors x and b, matrix A, and preconditioner P are
    already created
dc_solver_type solver( A );             // Create solver
solver.set_preconditioner( P );         // Set preconditioner
solver.solve(x, b);                     // Solve Ax = b
```

Listing 3.1: Assembling the defect correction solver to compute the solution of $\mathcal{A}\mathbf{x} = \mathbf{b}$

3.2.3 Time integration

The Initial Value Problem (IVP) defined by the free surface conditions are discretized with a method of lines approach. The discretization of the spatial derivatives yields a system of ordinary differential equations that we can write as a semi-discrete system,

$$\partial_t(\eta, \tilde{\phi}) = \mathcal{N}(\eta, \tilde{\phi}), \quad \mathcal{N} : \mathbb{R}^{2M} \to \mathbb{R}^{2M}, \tag{3.15}$$

where $\mathcal{N}$ is a nonlinear operator that approximates the spatial derivatives. We use the classical four stage and 4^{th}-order accurate explicit Runge-Kutta method (ERK4) to solve (3.15). Explicit Runge-Kutta schemes are attractive as stability constraints are often well understood and they are straightforward to parallelize. However, for explicit time integration schemes, the Courant-Friedrichs-Lewy (CFL) condition defines a necessary condition on the time step size for stable integration. For temporal integration of free surface water waves we define the CFL condition in terms of the dimensionless Courant number Cr and the wave celerity c,

$$Cr = c\frac{\Delta t}{\Delta x} \leq Cr_{max}. \tag{3.16}$$

where Cr_{max} depends on the method used to solve (3.15). For explicit time integration typically $Cr_{max} = 1$, though the stability analysis based on the local linearization (3.4), carried out in [EKGNL13], indicates that the time step is not severely limited by the horizontal resolution given a specific choice of vertical discretization, N_z. Numerical experiments find that this less strict condition is also true for the nonlinear model, as long as the waves are not too steep. These are attractive properties as they imply that the model is robust, insensitive to time step sizes and applicable for local grid refinements.

3.2.4 Wave generation and wave absorption

Wave generation and wave absorption are essential in order to perform a range of validation and engineering cases where waves propagate through or into the domain of interest. In order to generate and absorb waves we insert a generation

zone and an absorption zone in each end of the domain. Unless otherwise stated, the generation zone will be in the west side of the domain and the absorption zone in the east side. The generation zone forces the impact of an analytic wave function, while the absorption zone dampens the waves amplitudes towards zero. Several techniques exist for generation of waves. One common method forces a given wave function directly onto the numerical solution in the relaxation zones after each intermediate time step, see e.g. [LD83]. Such an approach requires that the time integrator is modified, which is not favorable, because it would violate the generic component design and require that the time integrator is designed specifically for a given problem. We prefer a more generic approach, introducing an embedded penalty forcing technique that adds a forcing term to the right hand side of the IVP. With this setup, the right hand side operator is modified and not the time integrator itself. In terms of a general IVP, $\partial_t q = \mathcal{L}(q)$, we add a forcing term such that

$$\partial_t q = \Gamma(\mathbf{x})\mathcal{L}(q) + \frac{1-\Gamma(\mathbf{x})}{\gamma}(q_0(\mathbf{x},t_1) - q(\mathbf{x},t)), \quad \mathbf{x} \in \Omega_\Gamma, \tag{3.17}$$

where γ is scaled to be approximately equal to the time step size $\gamma \approx \Delta t$, q_0 is the target solution at time t, and Ω_Γ defines the relaxation zone. $\Gamma(\mathbf{x})$ is a function $\mathbb{R}^2 \to \mathbb{R}$, that controls the strength of the forcing terms within the relaxation zones. Outside the relaxation zones $\Gamma = 1$ and inside it is $0 \leq \Gamma < 1$. Furthermore Γ should be chosen so that there is a soft transition between the zones and the inside domain, i.e. $\partial_x \Gamma = 0$ at the transition. We adopt the following form of relaxation functions, as presented by Engsig-Karup[EK06], for generation and absorption, respectively,

$$\Gamma_{gen}(\hat{x}) = 1 - (1-\hat{x})^p \tag{3.18}$$

$$\Gamma_{abs}(\hat{x}) = -2\hat{x}^3 + 3\hat{x}^2, \tag{3.19}$$

where p can be used to control the curvature of Γ_{gen}, see examples in Figure 3.3. The size of the relaxations zones can be controlled as an input configuration parameter and will be set in terms of the wavelength, $\hat{x} = x/(\beta L_x)$, where L_x is the wavelength and β controls the size of the relaxation zone. It is advisable to use at least one wavelength for each of the relaxation zones. Short zones may cause instability and wave reflections.

Propagation of linear sinusoidal waves over a flat bottom in a closed two-dimensional domain with generation and absorption zones given by (3.18) and (3.19), with $p = 4$ is demonstrated. The length of the wave tank is $L_x = 8L$ and the waves travel in a time period of $t = [0, 40]$. The wavelength is $L = \pi$, $kh = 1$ and $H/L = 30\%(H/L)_{max}$. The size of the spatial resolution is $(N_x, N_z) = (513, 9)$ and a Courant number $C_r = 0.75$ is used. Both relaxation zones are of length L. The free surface elevation at $t = 40\,s$ is illustrated in Figure 3.4 and the free surface elevation at $x = 5$ as a function of time is

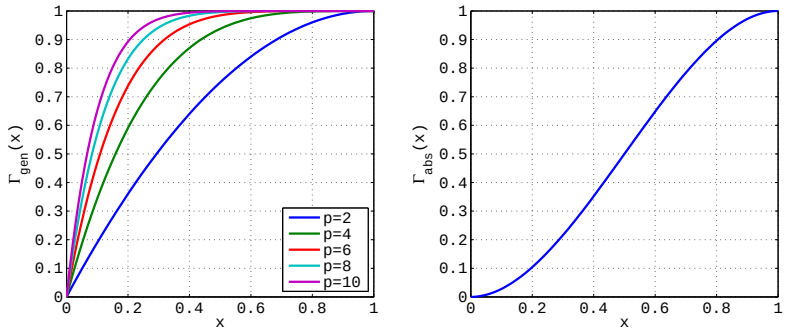


Figure 3.3: Relaxation functions for the generation (left) and absorbtion (right) zones.

illustrated in 3.5. There is a good match between the analytic and the numerical solution both in the spatial and temporal domain, with less than two percent relative amplitude and phase error.

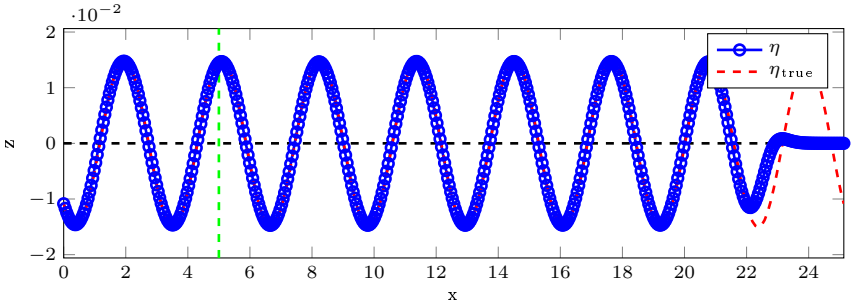


Figure 3.4: Linear numeric and analytic free surface elevation at time $t = 40$ using relaxation zones.

3.3 Validating the free surface solver

Stable, robust, and accurate free surface water wave solutions with a consistency proposed by the spatial and temporal discretizations are paramount for a valid implementation. The OceanWave3D solver, assembled from library parts, has therefore been subject to a long range of validation tests in both two and three dimensions, most of which we will omit from this work. The spatial flexible-order finite difference approximations and the 4^{th}-order accurate Runge-Kutta

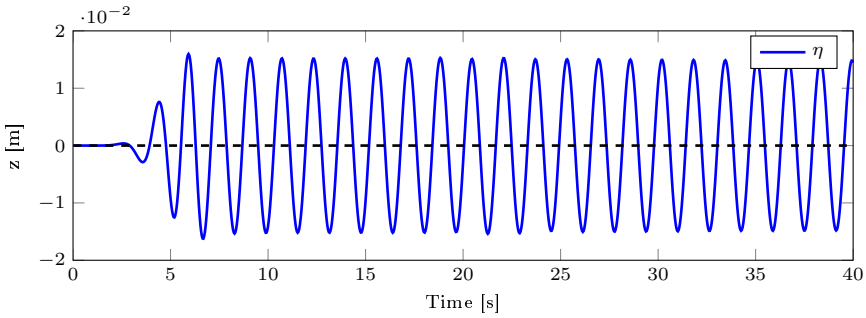


Figure 3.5: Linear free surface elevation at $\eta(x = 5, t)$ using relaxation zones.

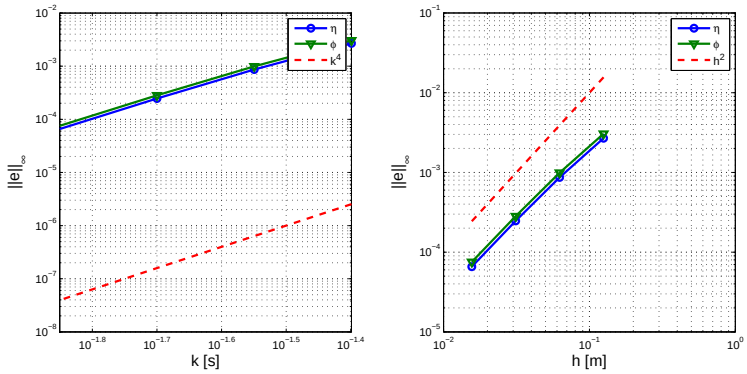
integration scheme is however verified here. The configuration is based on a non-linear traveling wave, due to Fenton & Rienecker[FR82], in a two-dimensional periodic wave tank of length $L_x = 2$. The wave number is $k = \pi$, $kh = 2\pi$, and $H = 0.04244$. The order of accuracy for the spatial approximations depends on the size of the stencil and is of the order $\mathcal{O}(\Delta x^{2\alpha})$, where α is the stencil half width. The temporal approximations are always of 4^{th}-order accuracy. Consistency results based on $\alpha = 1, 2, 3$ with the expected convergence rates are demonstrated in Figure 3.6. The expected orders of accuracy for both the temporal and spatial approximations of $\mathcal{O}(k^4 + h^{2\alpha})$ are confirmed, where k is the time step size and h is the grid space size.

3.3.1 Whalin's test case

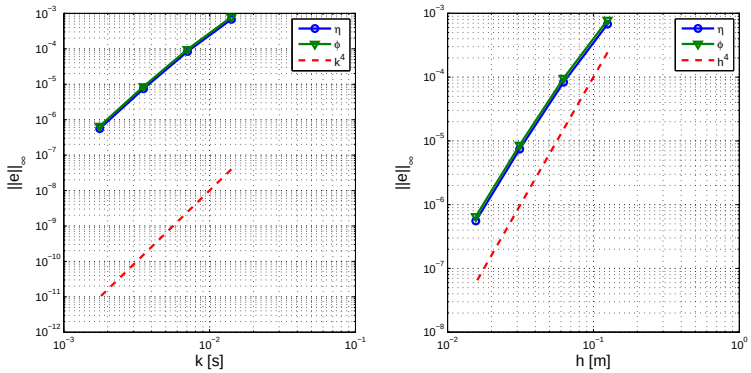
We here validate the solver using a classical benchmark for propagation of non-linear waves over a semicircular shoal. The benchmark is based on Whalin's experiment [WoEU71], which is often used in validation of dispersive water wave models for coastal engineering applications because it captures all relevant features of the implemented model. Experimental results exists for incident waves with wave periods $T = 1, 2, 3\,\mathrm{s}$ and wave heights $H = 0.0390, 0.0150, 0.0136\,\mathrm{m}$. All three test cases have been discretized with a computational grid of size $(257 \times 41 \times 7)$ to resolve the physical dimensions of $L_x = 35\,\mathrm{m}$, $L_y = 6.096\,\mathrm{m}$. The still water depth decreases in the direction of the incident waves as a semi-circular shoal from $0.4572\,\mathrm{m}$ to $0.1524\,\mathrm{m}$ with an illustration of the free surface given in Figure 3.7a at $t = 50\,\mathrm{s}$. The time step Δt is computed based on a constant Courant number of $Cr = c\Delta x/\Delta t = 0.8$, where c is the incident wave speed and Δx is the grid spacing. Waves are generated in the generation zone $0 \leq x/L \leq 1.5$, where L is the length of incident waves, and absorbed again in the zone $35 - 2L \leq x \leq 35\,\mathrm{m}$. All computations are carried out with

single-precision floating-points, indicating that single-precision is sufficient for achieving engineering accuracy for this test case, as there has been no difference, other than machine-precision, between the single- and double-precision results. There is an overt performance improvement for single-precision compared to double-precision, also reported in [GEKM11]. The reduced memory requirement impacts both the total memory footprint, but also doubles the throughput at all memory levels.

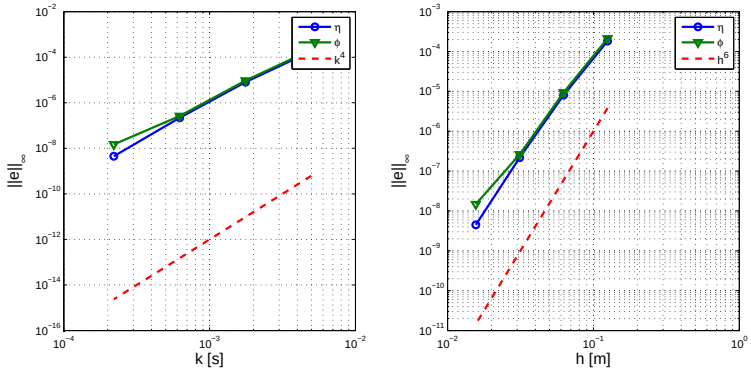
A harmonic analysis of the wave spectrum at the shoal center line is computed and plotted in Figure 3.7 for comparison with the analogous results obtained from the experiments. The three harmonic amplitudes are computed via a Fast Fourier Transform (FFT) method using the last three wave periods up to $t = 50\,\mathrm{s}$. There is good agreement between the computed and experimental results, in addition no loss of accuracy resulting from the use of single-precision arithmetic has been recorded for the Whalin test case. Double-precision results are intentionally left out, as they equal the single-precision results.



(a) $\alpha = 1$

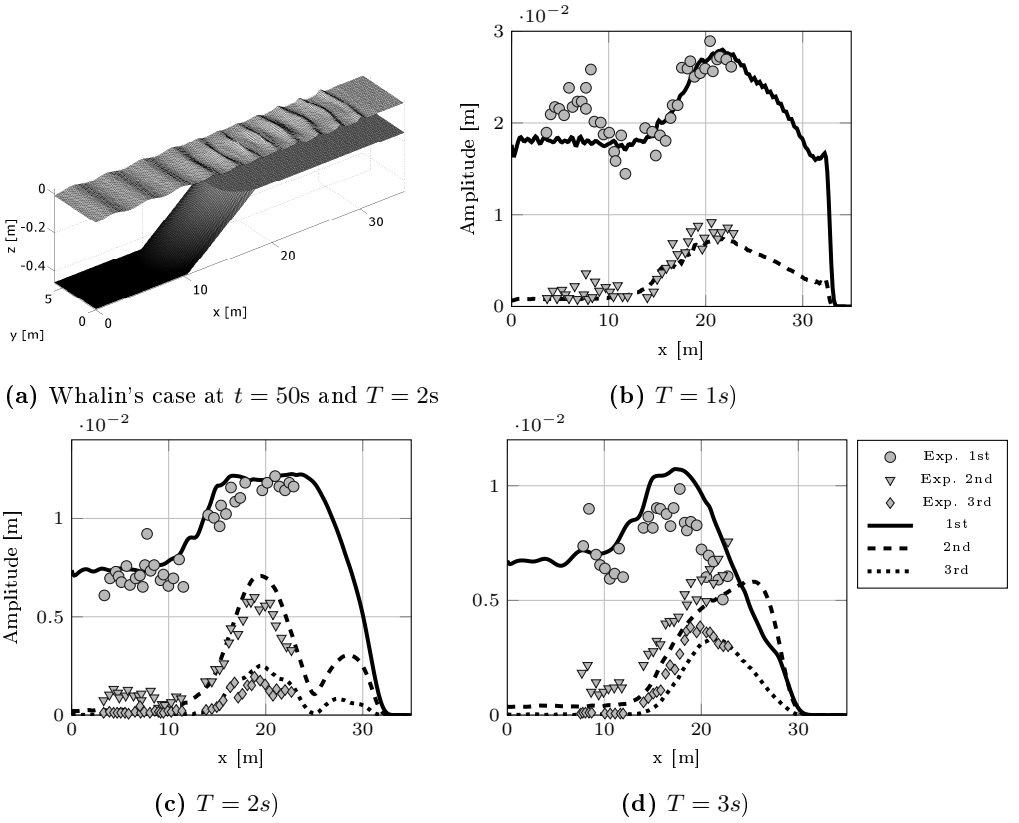


(b) $\alpha = 2$



(c) $\alpha = 3$

Figure 3.6: Consistency for the temporal ERK4 scheme and the spatial finite difference approximations with varying stencils sizes. Results are measured for the nonlinear travelling wave over.



(a) Whalin's case at $t = 50$s and $T = 2$s

(b) $T = 1s$)

(c) $T = 2s$)

(d) $T = 3s$)

Figure 3.7: Harmonic analysis for the experiment of Whalin for $T = 1, 2, 3$ s respectively. Experimental and computed results are in good agreement. Figures are only showing single-precision resutls, however double-precision results are equal to machine precision.

3.4 Performance breakdown

Performance measurements for the free surface model are presented throughout this work, in the context where new features are introduced. Here we present a performance breakdown of the basic single-GPU implementation of the free surface solver, on some of the most recent GPU architectures.

The free surface water wave problem based on potential flow theory has the special property that only the free surface itself has to be integrated in time and not the fully three-dimensional problem described by the Laplace problem. Thus, the four stage ERK4 time integrator requires a smaller memory footprint, as the state variables are only two-dimensional, and the memory required by the Laplace solver can be reused for each stage evaluation. A memory scaling test is illustrated in Figure 3.8 for increasing problem sizes in terms of total number of grid points $N = N_x\, N_y\, N_z$, based on both single- and double-precision. The problem size is based on a constant vertical resolution $N_z = 9$ and increasing horizontal resolutions. The scaling test confirms that there is linear scaling for both single- and double-precision as the problem size increases. An impressive problem size of almost 10^8 degrees of freedom can be used for discretization on a single GPU. For the smaller problem sizes the memory footprint does not scale perfectly. We expect that this effect is caused by the generic vector containers, that may automatically allocate more memory than requested, in order to maintain optimal memory alignment.

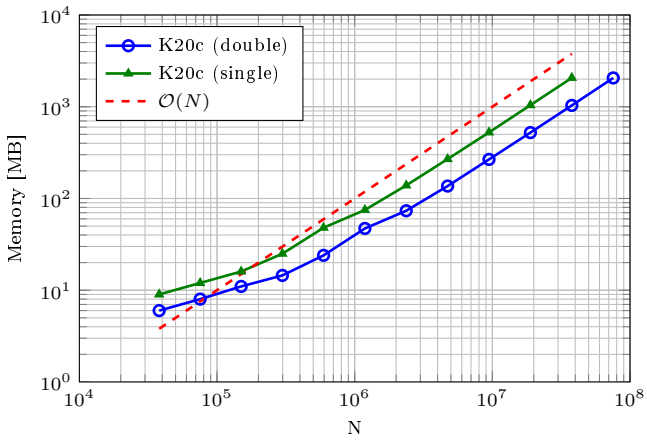
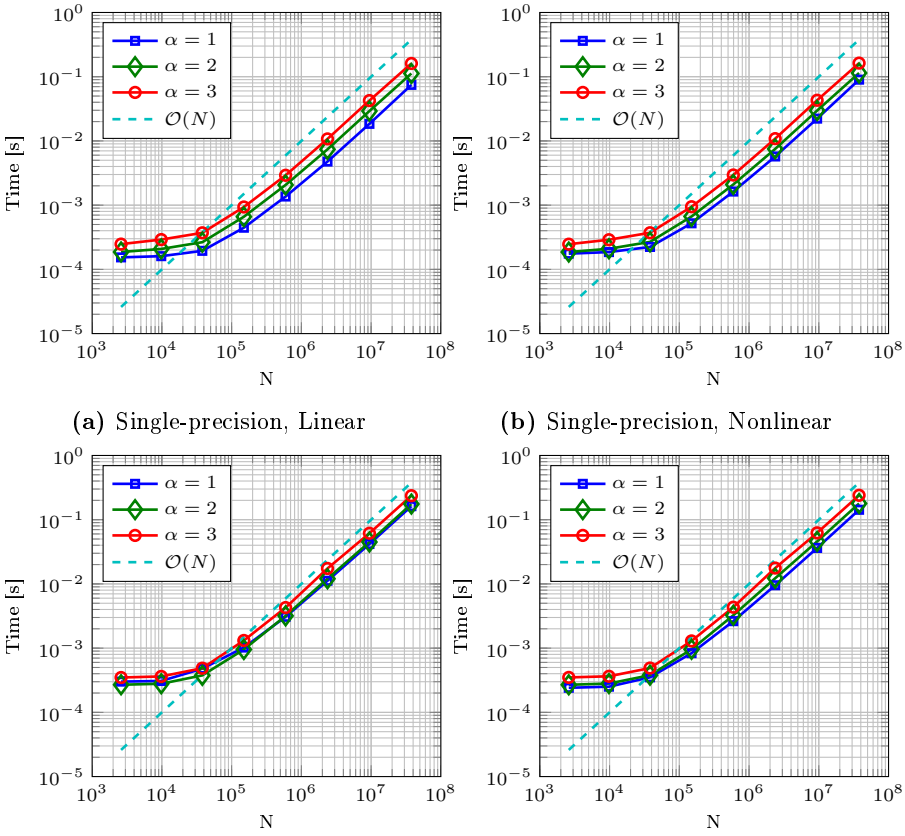


Figure 3.8: Memory scaling test for single- and double-precision.

The memory wall is *the* performance limiting factor for applications based on

matrix-vector like operations, such as the finite difference stencils computations used to compute the solution of (3.8). The reason is simply that the arithmetic intensity is too low compared to the number of memory transactions. Care should therefore be taken to ensure coalesced and minimal memory access to reduce the effect of the memory wall. One benefit from stencil approximations is that we are able to pre-compute the stencil coefficients and access them from either the low-latency shared or constant memory for optimal throughput. When a kernel is bandwidth limited, adding extra floating point operations will not affect the overall execution time significantly, so-called flops-for-free. One technique that will increase the number of floating-point operations is to increase the order of accuracy by increasing the stencil size α. The number of floating-point operations for a one dimensional finite difference approximation is $4\alpha + 1$ and the corresponding number of memory reads are $4\alpha + 2$. However, with cached memory from adjacent grid points and stencil coefficient, memory access will be less expensive and we are able to obtain an improved performance throughput in terms of floating-point operations per second (Flop/s). Absolute performance timings for computing the residual $\mathbf{r} = \mathbf{b} - \mathcal{A}\mathbf{x}$, based on the matrix-free finite difference stencil with $\alpha = 1, 2, 3$ and with single- and double-precision floating-points are illustrated in Figure 3.9 and 3.10. Timings based on a linearized discretization along with the fully nonlinear system are also illustrated. There is good linear scaling for problems sizes above 10^5 grid points, whereas for problems below 10^5 there are not enough degrees of freedom for the GPU to sufficiently hide memory latency while also occupying all cores. As one would expect, computations based on higher order approximation are more time consuming. However, as before proposed, the corresponding increase in floating-point operations leads to a favorable throughput increase as demonstrated in Figure 3.11. Based on the power consumption of a GPU ($\sim 250\,\mathrm{W}$), an estimated flops per watt performance figure could also be formed on the basis of the results in Figure 3.11, again highlighting that methods based on high-order discretizations are more energy-efficient.



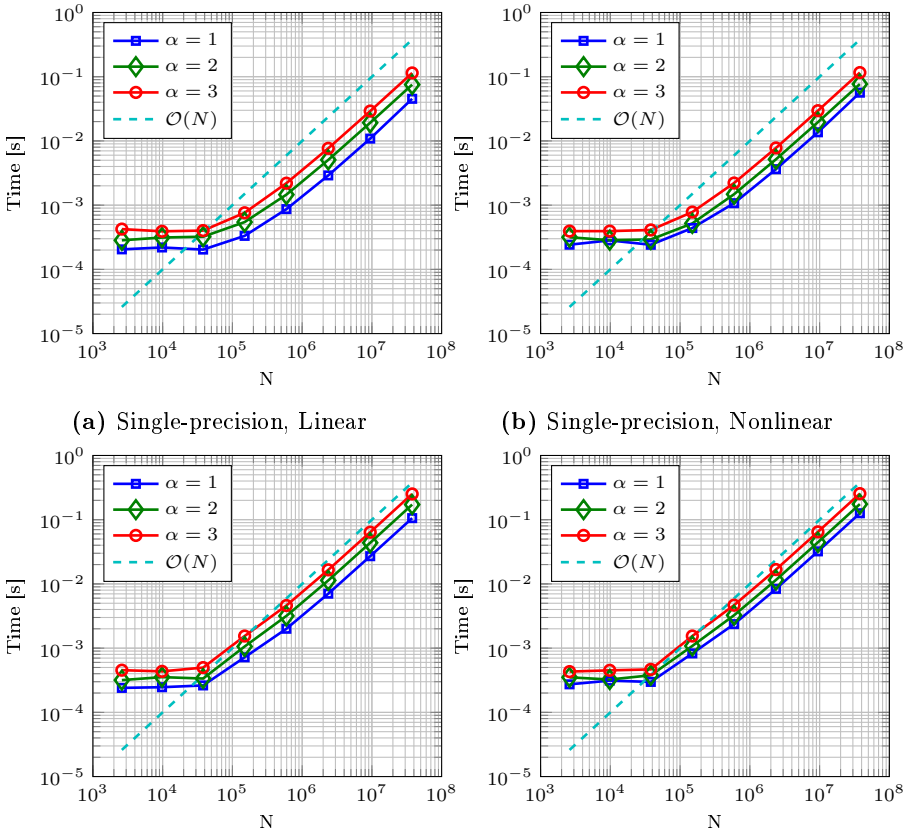
(a) Single-precision, Linear

(b) Single-precision, Nonlinear

(c) Double-precision, Linear

(d) Double-precision, Nonlinear

Figure 3.9: Absolute timings for computing $\mathbf{r} = \mathbf{b} - \mathcal{A}\mathbf{x}$, based on the linear and nonlinear discretizations with different stencil sizes. Using single- and double-precision, on *G4* (GeForce GTX590).



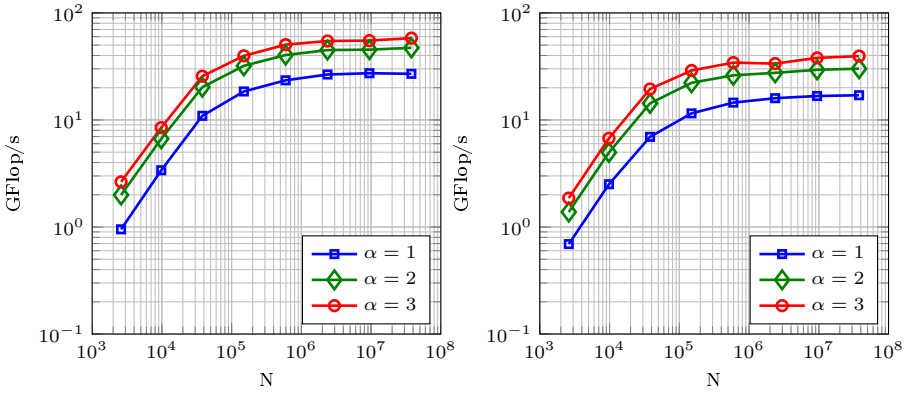
(a) Single-precision, Linear

(b) Single-precision, Nonlinear

(c) Double-precision, Linear

(d) Double-precision, Nonlinear

Figure 3.10: Absolute timings for computing $\mathbf{r} = \mathbf{b} - \mathcal{A}\mathbf{x}$, based on the linear and nonlinear discretizations with different stencil sizes. Using single- and double-precision, on *G6* (Tesla K20c).



(a) Single-precision **(b)** Double-precision

Figure 3.11: Performance throughput for computing $\mathbf{r} = \mathbf{b} - \mathcal{A}\mathbf{x}$. Using single- and double-precision, on *G4* (GeForce GTX590).

3.4.1 Defect correction performance breakdown

Computing the solution of the σ-transformed Laplace problem is the performance bottleneck of the free surface solver. A performance breakdown of the preconditioned defection correction is therefore carried out, with each of the subroutines outlined in Table 3.1. We measure the time it takes to complete one defect correction iteration, because this measure is independent of the physical problem and only depends on the discretization parameters. For this performance test, the vertical resolution is kept fixed at $N_z = 9$ while the horizontal resolution is increased repeatedly in the x- and y-directions. The fully nonlinear system is solved with a 6^{th}-order accurate ($\alpha = 3$) finite difference approximation in the outer defect correction iteration, while one multigrid V-cycle with two Red-Black Z-line Gauss-Seidel smoothings is used for preconditioning together with a linearized 2^{nd}-order finite differences approximation; in short we write DC+MG-ZLGS-1V(1,1). Absolute and relative performance results are presented in Figure 3.12 and 3.13. From the absolute timings we see the same pattern as from the previous results: that they scale well only for sufficiently large problem sizes. The relative performance results in Figure 3.13 tell us that for large problem sizes, the high-order residual and the smoother are responsible for the majority of the compute times. The combined performance scalability for the compute time per defect correction, is summarized in Figure 3.14. These results confirm again that this is a memory bound application as the results follow directly from the memory bandwidth of the individual GPUs. The Tesla K20c GPU has the highest bandwidth of 208 GB/s while the Tesla M2050 bandwidth is lowest at 144 GB/s. The high-end gaming GPU, the GeForce GTX590 has an intermediate bandwidth of 164 GB/s. From these performance tests we see that with just a single GPU we are able to compute the solution of a free surface water wave problem with 10^7 to 10^8 degrees of freedom in both single- and double-precision, with iteration times well below one second. Furthermore, these performance results are in good agreement with the results previously reported in [EKMG11], which was based on a dedicated solver. Thus, we conclude that the generic library implementation does not introduce significant overhead despite providing a higher level of abstraction.

3.4.2 A fair comparison

In Figure 3.14 we provided the timings based on an fairly optimized single threaded Fortran90 version of the same numerical free surface solver used in [EKBL09, DBEKF10]. The CPU performance results are measured on a Linux-based system equipped with an Intel Core i7 at 2.8 GHz and with an effective CPU-to-RAM bandwidth of 11.5 GB/s.

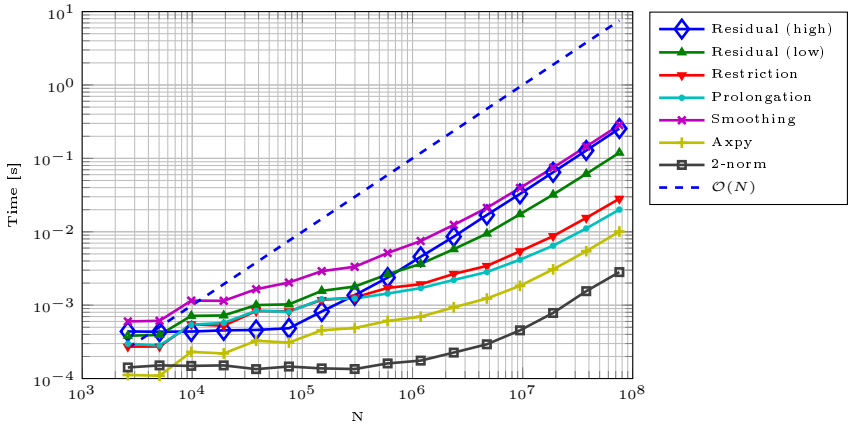


Figure 3.12: Absolute timings on *G6* for each of the subroutines in a defect correction iteration. Double-precision.

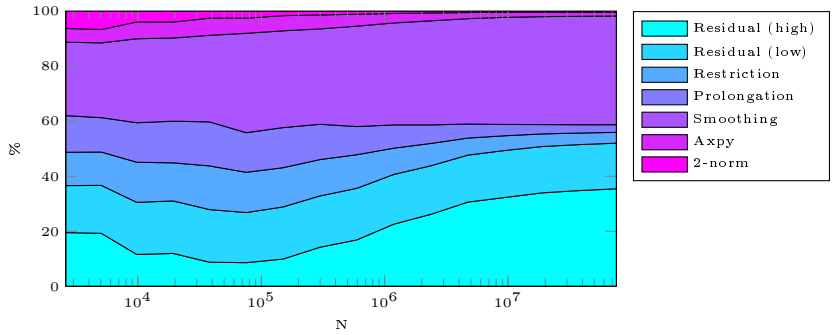
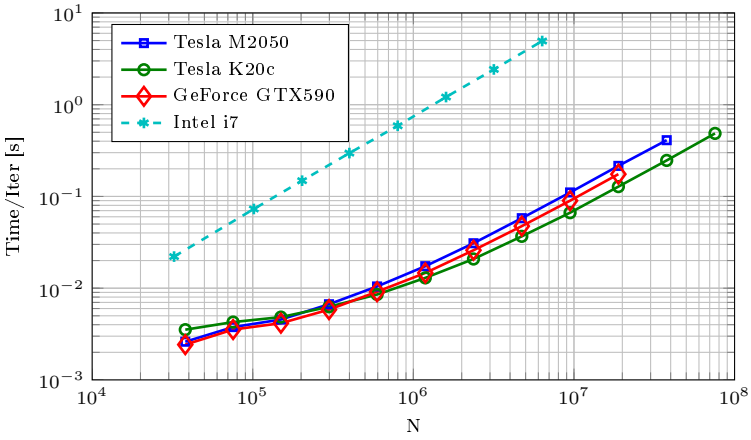
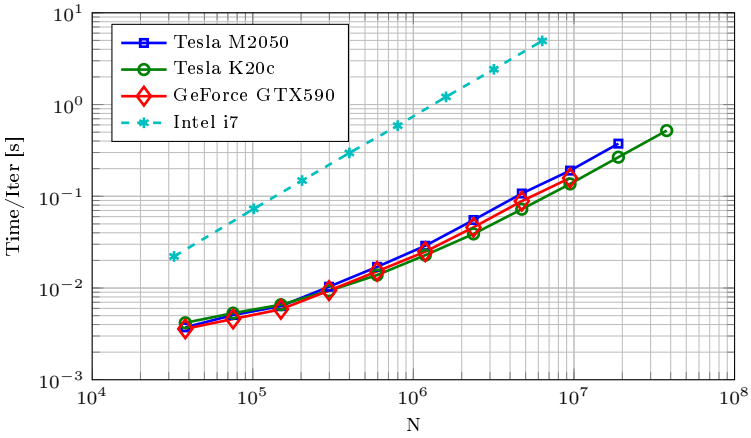


Figure 3.13: Percantage of total compute time on *G6* for each of the subroutines in a defect correction iteration. Double-precision.

It is our experience, that there seems to be some confusion and skepticism in the literature and among researchers, when it comes to comparing performance results. Stating that GPUs are several (100-1000) times faster than equivalent CPUs does not say anything about the reference for achieving such speedups. The key to these problems is that these numbers are often provided with no context and without sufficient details. In fact it can be quite easy to mislead the reader by leaving out details, as humorously described by Bailey [Bai91, Bai92, Bai09]. No optimized GPU application will be thousand times faster than an equally optimized CPU version that utilizes all CPU cores. However, we believe that a fair comparison *is indeed* valid, if the right comparison basis is presented. A fair comparison requires that sufficient details of the test environments are



(a) Single-precision



(b) Double-precision

Figure 3.14: Performance comparisons and scalability of timings per precondtioned defect correction iteration on different heterogeneous hardware.

provided, that details about the algorithms are outlined, and that timings are presented, preferably both absolute and relative timings. When this information is provided, it is fair to compare applications that have not received the same level of optimization, then the comparison will be of the two applications, and not necessarily the hardware on which they are executed.

In Figure 3.14 we provide the comparison between the two versions of the same

numerical solver, based on a fairly optimized single-threaded double-precision Fortran90 implementation and the optimized massively parallel CUDA implementation. The corresponding speedups are reported in Figure 3.15, where significant speedups are obtained for all configurations, in good agreement with [EKMG11]. The single-precision performance results are naturally superior, and we would like to point out, that during our work we have never experienced a case where single-precision would fail to solve a problem that would be solved in double-precision. The robustness of the single-precision calculations stems from the use of a robust iterative solver.

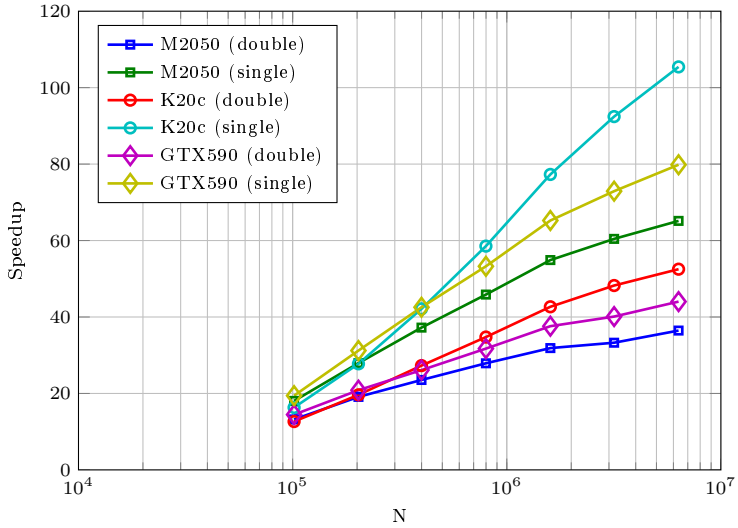


Figure 3.15: Relative speedups obtained with the parallel GPU implementation compared to the single-threaded CPU version of the same solver.

Chapter 4

Domain decomposition on heterogeneous multi-GPU hardware

Decomposing a boundary value problem into smaller subdomains is an attractive way to extract parallelism into an application. There are two main motivations for doing so. First, the computational work can be distributed and solved in parallel to achieve better overall performance. Secondly, memory distribution lowers the memory requirements per node and allows for larger global problems to be solved. However, communication between compute nodes does not come for free, and frequent message passing tends to dominate the overall performance, especially for smaller problem sizes. Domain decomposition techniques, such as the class of classical overlapping Schwartz methods, first introduced by Schwarz in 1870, can be considered as a preconditioners to iterative solvers, see e.g., [SBG96]. Each subdomain solver approximates a local solution (possibly in parallel) to some accuracy, before communicating and updating boundary information with adjacent subdomains. This procedure is repeated until a global accuracy criterion is met. One advantage of these methods is that the local solutions can be approximated with no communication between subdomains, and even different numerical solvers can be used within each subdomain. The disadvantages are that global convergence depends on the size of the overlaps and that larger overlaps lead to increasingly redundant computational work.

Furthermore, the overlap size and position influence how rapidly information travels between subdomains at each Schwartz iteration, and therefore how fast the overall method converges. For some elliptic problems, a global coarse grid correction strategy can improve convergence by ensuring propagation of information between all subdomains. For boundary value problems with finite signal speed, e.g., the weakly dispersive Boussinesq equations, satisfactory convergence speed can be obtained with reasonable overlaps of the order of the water depth, as presented in [GPL04, CPL05].

4.1 A multi-GPU strategy

CUDA enabled GPUs are optimized for high memory bandwidth and fast on-chip performance. However, the role as a separate co-processor to the CPU can be a limiting factor for large-scale scientific applications, because the GPU memory capacity is fixed and is only in the range of a few Gigabytes. Therefore, large-scale scientific applications that process Gigabytes of data, require distributed computations on multiple GPUs. Multi-GPU desktop computers and clusters can have a very attractive peak performance, but the addition of multiple devices introduces the potential performance bottleneck of slow data transfers across PCI Express busses and network interconnections, as outlined in Figure 4.1. The ratio between data transfers and computational work has a significant impact on the possibility for latency hiding, and thereby overall application performance. Thus, speedups should not be expected for smaller problem sizes where there is an unfavorable ratio between communication and computations. In the following sections we investigate the cost of data transfers between GPUs and demonstrate how the number of processors and grid topology influence the overall performance and scalability.

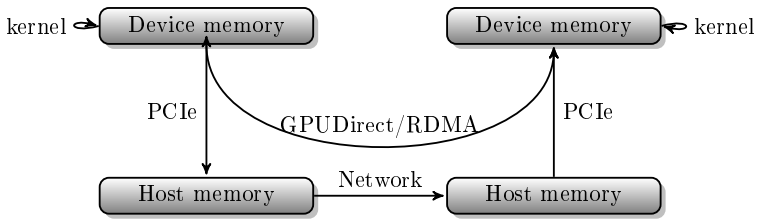


Figure 4.1: Message passing between two GPUs involves memory transfers across lower bandwidth connections. A kernel call is first required if data is not sequentially stored in the GPU memory. Recent generations of Nvidia GPUs, CUDA, and MPI support direct transfers without explicitely transfering data through the host.

The objective of this work is to enable scalable computations on distributed systems to enable large-scale simulations on heterogeneous clusters, where the solver can account for a large number of time steps and many billions degrees of freedom in the spatial discretisation. We therefore introduce a natural (but nontrivial) extension of the GPUlab library, to support distributed data and parallel computations on heterogeneous hardware with multiple GPUs, using a hybrid MPI-CUDA implementation. We implement a multi-block method based on a domain decomposition technique that automatically decomposes the global computational grid into local subgrids. We detail how this technique can be used to transparently solve very large linear system of equations arising from finite difference discretizations of flexible order.

4.1.1 A multi-GPU strategy for the Laplace problem

From previous work [LF97] we know that multigrid convergence for the Laplace problem is close-to-optimal and converges in very few iterations to engineering accuracy for both deep and shallow waters. Also, the system of equations arising from higher order approximations, based on flexible order finite differences, have been shown to have good convergence with the iterative defect correction method and multigrid preconditioning[EKBL09]. We therefore seek a decomposition technique for large scale simulations that preserves the attractive algebraic multigrid convergence with a low penalty on the overall performance. Our multi-block decomposition technique will therefore introduce artificial ghost (halo) layers between adjacent subdomains, with sizes equal to the stencil sizes. The computational domain is first decomposed into non-overlapping subdomains, then ghost layers are added to account for grid points that are distributed across adjacent subdomains, similar to the approach presented e.g. in [ALB98]. The size of the ghost layers can be adjusted to match the size of the finite difference stencils. In practice it can be advantageous to ensure that geometric multigrid operates on grids of uneven dimensions. We therefore allow one layer of grid points, next to the ghost layers, to be distributed to both of the adjacent subdomains, see Figure 4.2. Ghost points are updated block wise, indicated by the arrows, every time information from adjacent domains are queried. Thus, the subdomains are not decoupled and one subdomain cannot solve the local boundary value problem without communicating with its neighbors. As a consequence the iterative solution to the Laplace problem converges exactly as it would do without decomposition. Any solver that works on the non-distributed problem therefore works with the same algorithmic efficiency on the distributed problem.

The vertical resolution for any free surface model is much smaller than the horizontal resolution. Vertical grid points in the order of ten are sufficient for

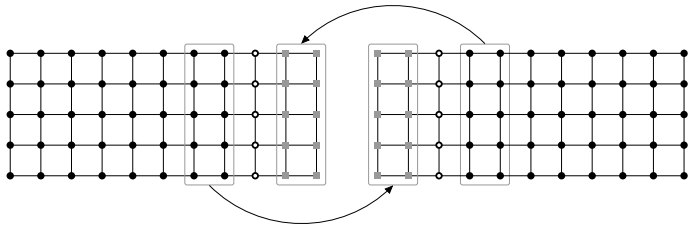


Figure 4.2: A one dimensional topology decomposition of a grid of global size 17×5 into two subgrids with two layers of ghost points. ● and ■ represent internal grid points and ghost points respectively. ○ represents internal points that are shared between the two grids to ensure uneven internal dimensions.

many applications in coastal engineering, while the number of horizontal grid points is dictated by the size of the physical domain and the wavelengths. For practical applications, it is often desirable to extend the physical domain horizontally to restore large maritime areas, long wave propagations or large harbor facilities. In these situations the ratio between internal grid points and ghost points becomes more favorable for large-scale simulations because of the volume to surface ratio. For the same reason communication time is expected to be less dominant for large-scale free surface problems. Furthermore we will present a thorough examination of the multigrid coarsening strategy, to demonstrate that rapid numerical convergence can be maintained with few restrictions and with no need for coarse global grid corrections.

4.2 Library implementation and grid topology

For message passing we use MPI and create a local communication space in order to avoid interference with other communicators, as recommended by Hoefler and Snir [HS11]. Appendix A gives examples on how to initialize the library with MPI support.

The multi-block extension should be implemented so that it will not interfere with existing implementations. The library already contains the grid class for single-block grid representation. We extend this class with an extra grid topology template argument, and set the default value as a non-distributed topology implementation. An update member function is added to the grid class, that simply forwards the grid itself to the topology update function. With this flexible template-based setup the user is able to create custom topology

implementation. The library contains default topology operators for one- and two-dimensional grid distributions. Listing 4.1 illustrates a template for the basic topology interface.

```
template <typename size_type, typename value_type>
class topology
{
public:
    int N, S, E, W, T, B;    // MPI ranks for neighbors, north, south, ...
    int P, Q, R;             // Global grid topology size
    int p, q, r;             // Local grid topology IDs

    /**
    * Create local grid property dimensions based on global dimensions
    */
    template <typename property_type>
    void create_local_props(property_type const& gprops, property_type*
        lprops)
    {
        // Fill out local properties (lprops)
    }

    /**
    * Update grid overlapping zones
    */
    template <typename grid_type>
    void update(grid_type const& g) const
    {
        // Communicate with neighbors to update grid ghost layers
    }
};
```

Listing 4.1: Template for a grid topology implementation.

When a grid is initialized with the global grid dimensions, the `create_local_props` function is automatically called to compute the local grid properties based on the given topology implementation. We use the MPI built-in Cartesian topology routines to help partition our grids. This multi-block topology implementation will be able to significantly improve developer productivity for creating data-distributed grid and vector objects and for updating boundary information. The only thing required to turn a single-block solver into a multi-block solver is to change the type definition for the grid type, exemplified in Listing 4.2.

```
using namespace gpulab;
typedef float                                              value_type;
typedef topology_xy<size_t,value_type>                     topology_type;
typedef grid<value_type,device_memory,topology_type>       grid_type;
```

Listing 4.2: Setting up type definitions for a multi-block grid with horizontal grid decomposition `topology_xy`).

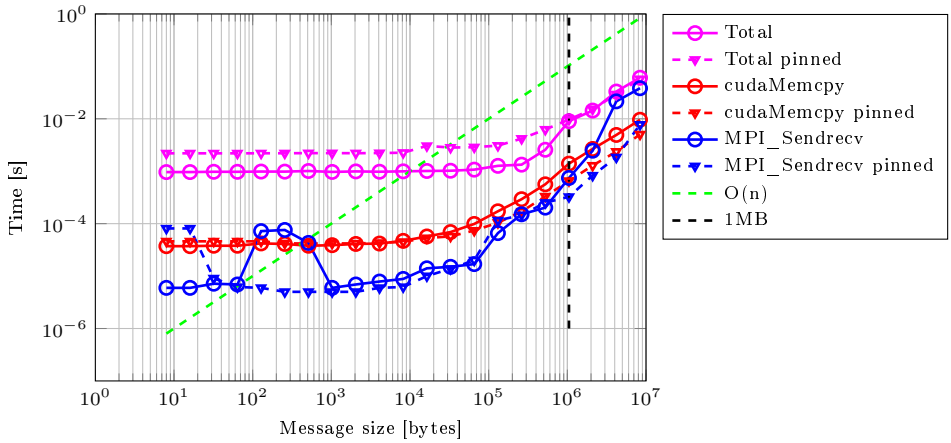
Instead of initializing a grid from the global grid properties, it is also possible to create a custom topology configuration and then use local grid properties. This technique will be utilized in Section 4.5 to manually detach two of the blocks in order to create a breakwater simulation.

One challenge that arises from the multi-block setting is how to ensure good load balancing. In the predefined library implementations, we use the MPI Cartesian topology routines to decompose the grid and we ensure that all local grids are of similar discrete sizes. However, if data is distributed to multiple GPUs with different performance, or the user manually creates a bad distribution, the solver will be limited by the slowest link, because updating boundary information is a collective operation. We have not experienced any noticeable performance issues with load imbalance, but it is an issue that may be relevant in some settings.

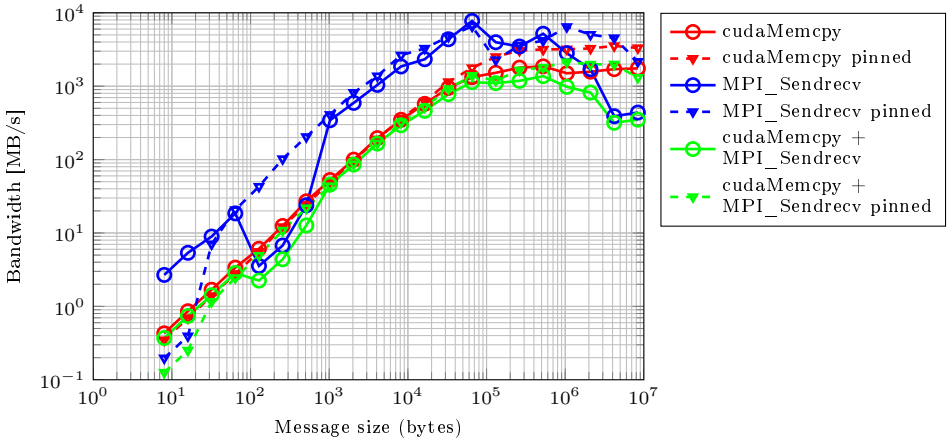
4.3 Performance benchmarks

Message passing between GPUs is nontrivial, and only few stable MPI releases support pointers to GPU memory. Good progress on efficient and transparent GPU-to-GPU communication is being made within MPI communities, particularly MVAPICH2[WPL+11b, WPL+11a, Nvi12a]. However, as of writing, there are restrictions on the supported InfiniBand controllers, and MPI requires specialized configurations with non-default settings. In the absence of access to such a configured system, the following results are based on an implementation where communication between two or more GPUs are performed by first transferring data between the device and host memory, before invoking the appropriate MPI calls, cf. Figure 4.1. We believe that simple and transparent integration of message passing using MPI on heterogeneous systems will be a key feature in the near future as distributors and researches continue to make hybrid message passing easier[ABD+13], and also extending support for non-contiguous data movement as presented by [JDB+12, WPL+11a].

Updating decomposed subgrids with boundary information via ghost layer updates imposes a potential performance bottleneck, thus all communication overhead should be minimized if possible. We have created a mirco-benchmark that will provide us with information about the efficiency and scalability of message passing between multiple GPUs on the two test environments *G4* and *Oscar*. For the first test we use two MPI processes to measure the time it takes to exchange messages of increasing size between two GPUs connected to the same motherboard. For this test we use both pinned (page-locked) and non-pinned host memory. According to the CUDA guidelines[Nvi13], pinned host memory can be utilized for faster memory transfers between the host and device. The performance results are presented in terms of absolute timings in Figure 4.3a and memory bandwidth in Figure 4.3b. There are several important things to notice from these timings. Firstly, since the two GPUs are on the same board, the MPI send/receive call is a local memory operation that requires no network transfer. Therefore this is very efficient for all message sizes and requires the



(a) Absolute timings. Total timings include all transfer times and additional overhead time to allocate and free memory.



(b) Memory bandwidth performance

Figure 4.3: Performance scaling for two processors on *G4*. Send/receive transfers from GPU to GPU, with and without pinned memory.

least time. The irregular peaks that appear for the MPI calls, are something that we often observe at random locations and therefore something we expect to be caused by internal MPI controllers. The next thing we should notice is that even though pinned memory enables the highest peak bandwidth for large memory transfers between the host and the device (cudaMemcpy), there is almost no difference for memory transfers below 1 MB. In order to put these message

sizes into perspective, we know from Section 3.4 that on a single GPU, we are able to solve a discretized boundary value problem with up to almost 10^8 degrees of freedom. Assuming that such a discrete problem is three-dimensional, and has an equal number of grid points in all three directions, then three layers of ghost points with double-precision floating-points would require less than 5 MB. In practice this would often be much smaller, particularly for the free surface water wave model. The final, but most important, thing to notice from Figure 4.3, is that the time to allocate and deallocate host memory requires a significant amount of extra overhead and that for pinned memory it is more time consuming. As a result of this benchmark, we make sure that the generic topology implementation creates a memory pool that will be reused for multiple grid updates in order to minimize allocation overhead.

For the next benchmark we will test device-to-device memory transfers on the *Oscar* GPU-cluster, with a different number of MPI processes and devices. We measure again the time it takes to send and receive a message. The messages are sent in a circle, such that each MPI process has two neighbors, and then sends to the one and receives from the other. The procedure is repeated 100 times to be able to compute an average. The timing results are reported in Figure 4.4. We observe that *Oscar* is very sensitive to interference, visible from the jumping behaviour. Though we have executed the benchmark several times, we continuously get different results with similar behaviour. We have not been able to identify if it is caused by user interference, software issues, or the physical cluster topology. However, during multiple test runs we have observed good results based on all configurations.

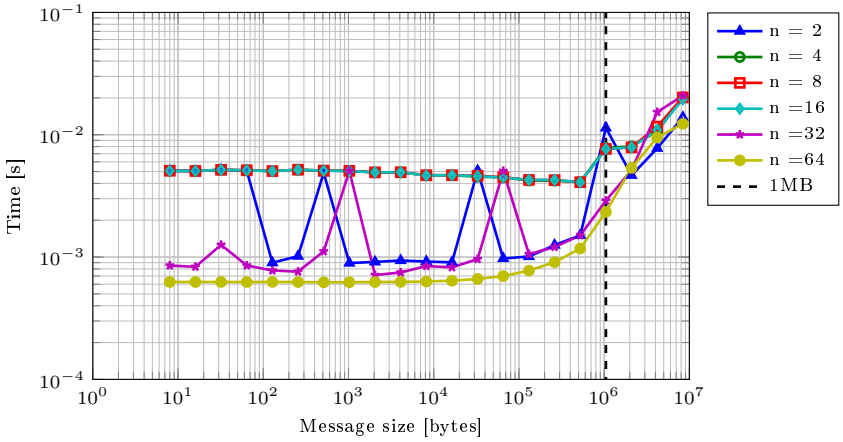
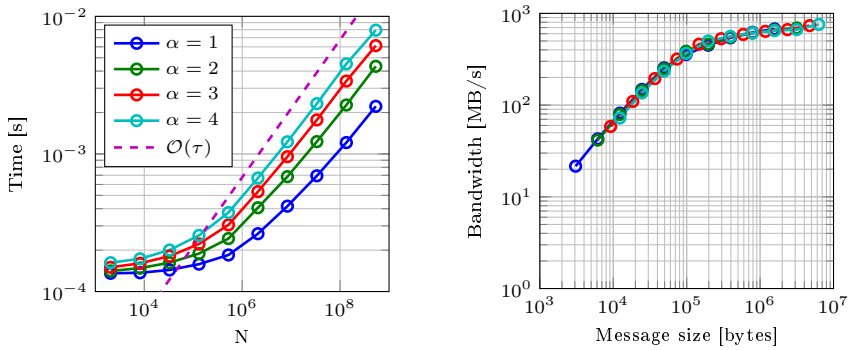


Figure 4.4: Absolute timings for n MPI processors on *Oscar*. Send/receive messages from GPU to GPU.

We now present a benchmark where a grid decomposed into subgrids exchanges boundary information based on a horizontal topology implementation as presented in Section 4.2. We use a numerical grid of increasing global size $(N_x, N_y, N_z) = (2^i, 2^i, 32)$, for $i = 3, 4, \ldots, 12$ with overlaps of size $\alpha = 1, 2, 3, 4$. The total number of ghost grid points per subgrid is then bounded up to,

$$\tau \leq 2\alpha N_z \left(\frac{N_x}{P} + \frac{N_y}{Q} \right), \tag{4.1}$$

Figure 4.5 illustrates performance scalability for updating a grid decomposed into two subrids with $(P, Q) = (1, 2)$ on the same compute node on *Oscar*. In contrast to the previous benchmarks, these timings also include the kernel call that is required to first move ghost layer information into a contiguous memory location before the device to host copies. In [WPL+11b] the authors evaluate non-contiguous MPI data communication on GPU clusters, and propose the same approach for multi-dimensional data sets. They also emphasize that manual data movement in multi-GPU settings poses one of the biggest hurdles to overall performance and programmer productivity, which again emphasizes the importance of well-designed library support. There is a natural extra time cost for updating grids with increasing ghost layer sizes, but as Figure 4.5b indicates, there is no extra cost in terms of bandwidth performance.

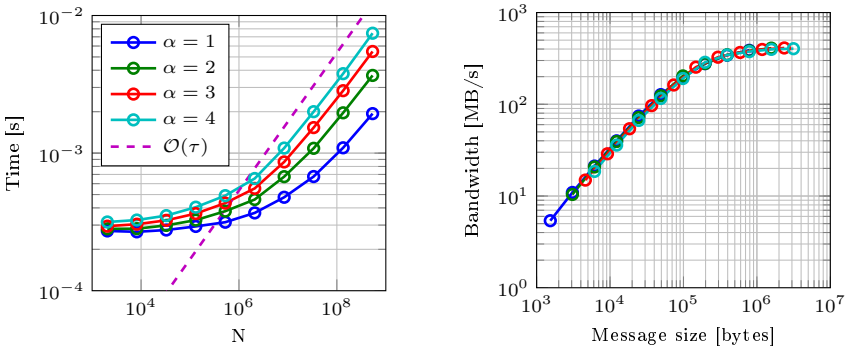


(a) Absolute timings **(b)** Bandwidth

Figure 4.5: Grid update performance on **two** GPUs, using four different overlap sizes. The global grid size is $N = N_x\, N_y\, N_z$. Tested on *Oscar*, single-precision.

In Figure 4.6 the same setup is tested, this time with eight GPUs on multiple nodes, so that $(P, Q) = (2, 4)$. For the smaller problems of sizes less than $N < 10^6$, there is a noticeable larger overhead for sending multiple messages compared to the timings in 4.5a. This extra overhead is not surprising since messages are now transferred across the network. As a results we also see that

the memory bandwidth performance decreases slightly. For larger problem sizes the performances scales well in both cases.



(a) Absolute timings **(b)** Bandwidth

Figure 4.6: Grid update performance on **eight** GPUs, using four different overlap sizes. The global grid size is $N = N_x\, N_y\, N_z$. Tested on *Oscar*, single-precision.

4.4 Decomposition of the free surface model

To solve the free surface water wave problem on multiple GPUs we use the data and domain decomposition technique presented above. We will again focus on efficient solution of the σ-transformed Laplace problem as this is the performance bottleneck. The purpose of the proposed decomposition technique is to preserve the attractive algorithmic convergence rate of the multigrid preconditioned defect correction method. If the artificial ghost boundary information is continuously updated before the information is needed, there will be no penalty on the algorithmic convergence and we will be able to solve the full global Laplace problem without the additional Schwartz iterations. The disadvantage of this technique is that it requires multiple grid updates, but as described in the previous section, the time per grid update is small compared to the time per defect correction as presented in Section 3.4. Thus, we expect the domain decomposition technique to have satisfactory weak scaling properties for the solution of the Laplace problem.

We will start by presenting an algebraic formulation of the decomposed Laplace problem. Then we will validate and test performance scalability of the multigrid preconditioned defect correction method, with special focus on the multigrid

coarsening strategy. Last, we demonstrate a free surface case where the multi-block solver is utilized to decouple domains, in order to emulate a breakwater.

4.4.1 An algebraic formulation of the Laplace problem

The introduction of new artificial ghost layers that connect subdomains can be introduced into the equations for the discrete Laplace problem (3.6). To clarify the conceptual change we derive an algebraic formulation of a discrete Laplace problem, that takes into account the extra conditions for the ghost points introduced across the artificial boundaries between an arbitrary number of subdomains.

Assume that a domain Ω is decomposed into P non-overlapping domains, such that $\Omega = \cup_{i=1}^{P} \Omega_i$. Then the transformation of the original system of equations $\mathcal{A}\mathbf{x} = \mathbf{b}$ into $\hat{\mathcal{A}}\hat{\mathbf{x}} = \hat{\mathbf{b}}$ can be described in terms of two sets of restriction matrices $\mathcal{R}_i$ and $\mathcal{G}_i$. $\mathcal{R}_i$ is the restriction matrix that selects exactly the elements $\mathbf{x}_i$ from $\mathbf{x}$ that belongs to the subdomain Ω_i, i.e. so that we have $\mathbf{x}_i = \mathcal{R}_i\mathbf{x}$. Then $\mathcal{R}_i \in \mathbb{R}^{N_i \times N}$, where N_i is the number of elements in Ω_i (excluding ghost points) and N is the total number of elements in Ω. The restriction matrix $\mathcal{R}_i$ is sparse and contains only one entry per row. The second restriction matrix $\mathcal{G}_i$ selects exactly the elements in $\mathbf{x}$ that will be covered by ghost points belonging to subdomain Ω_i. We have $\mathcal{G}_i \in \mathbb{R}^{Q_i \times N}$, where Q_i is the number of ghost points introduced by subdomain Ω_i. Because all ghost points added by subdomain Ω_i always represents elements from other subdomains $\Omega_{j \neq i}$, the non-zero columns of $\mathcal{R}_i$ and $\mathcal{G}_i$ are distinct.

In the general case, when an arbitrary dimensional domain Ω is decomposed into P subdomains, the corresponding system of equations $\hat{\mathcal{A}}\hat{\mathbf{x}} = \hat{\mathbf{b}}$, based on a finite difference discretization with overlapping ghost points can be written in the form

$$\left[\begin{array}{cccc:cccc}
\mathcal{A}_1 & & & & \mathcal{R}_1\mathcal{A}\mathcal{G}_1^T & & & \\
& \mathcal{A}_2 & & & & \mathcal{R}_2\mathcal{A}\mathcal{G}_2^T & & \\
& & \ddots & & & & \ddots & \\
& & & \mathcal{A}_P & & & & \mathcal{R}_P\mathcal{A}\mathcal{G}_P^T \\
\hdashline
0 & \mathcal{G}_1\mathcal{R}_2^T & \cdots & \mathcal{G}_1\mathcal{R}_P^T & -\mathcal{I} & & & \\
\mathcal{G}_2\mathcal{R}_1^T & 0 & \cdots & \mathcal{G}_2\mathcal{R}_P^T & & -\mathcal{I} & & \\
\vdots & \vdots & \ddots & \vdots & & & \ddots & \\
\mathcal{G}_P\mathcal{R}_1^T & \mathcal{G}_P\mathcal{R}_2^T & \cdots & 0 & & & & -\mathcal{I}
\end{array}\right]
\left[\begin{array}{c}
\mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_P \\ \mathbf{g}_1 \\ \mathbf{g}_2 \\ \vdots \\ \mathbf{g}_P
\end{array}\right]
=
\left[\begin{array}{c}
\mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_P \\ 0 \\ 0 \\ \vdots \\ 0
\end{array}\right],$$

where $\mathbf{g}_i$ is the ghost points added to the system by the i^{th} subdomain. Matrix $\hat{\mathcal{A}}$ can be divided into 4 blocks, indicated by the dotted lines. The upper left block contains the original contributions from $\mathcal{A}$, however only the entries that

apply internally to each subdomain, and not across artificial boundaries, are included. The upper right block connects the cut-off portions of $\mathcal{A}$ to the ghost points added to the system by the extension of $\mathbf{x}$ into $\hat{\mathbf{x}}$. Together, the two lower blocks ensure that each ghost point matches exactly one element in another subdomain. This is guaranteed since each equation from the lower blocks reduces to $x_j - g_i = 0$, such that ghost point g from subdomain Ω_i must be equal to x in subdomain Ω_j.

It should be noted that in practice this system is never formed explicitly and that the introduction of new equations into the original system of equations due to ghost points, merely indicates the communication that will be required by the underlying implementation to update all ghost points values.

4.4.2 Validating algorithmic convergence

One advantage of the multi-GPU method presented here is that it preserves the very attractive algorithmic efficiency of multigrid. Figure 4.7 illustrates the convergence history for the preconditioned defect correction method applied to a simple nonlinear traveling wave simulation. The convergence history confirms that in fact the norm of the residual converges equally for both the single- and multi-GPU implementations within machine precision. The notation $\Omega_{P=4}$ represents decomposition into four subdomains, and thus the use of four GPUs to compute the solution.

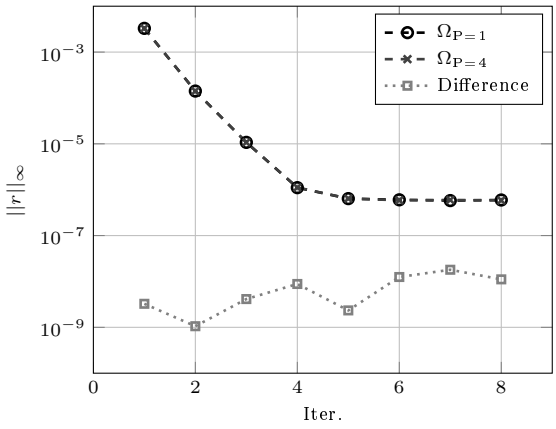


Figure 4.7: Convergence history for the defect correction iterations using one and four subdomains. The results confirm that the algorithmic efficiency is preserved. Single-precision arithmetic is used.

4.4.3 The effect of domain decomposition

As a first test case we consider how the introduction of multiple subdomains, and thereby multiple GPUs, influence the performance of each individual subroutine of the defect correction algorithm. We refer the reader to Table 3.1 for an overview of these subroutines.

We compute the average timings based on 100 defect correction iterations for an increasing number of unknowns. One multigrid V-cycle with Red-Black Z-line smoothing is used for preconditioning together with 6^{th}-order accurate finite difference for the spatial discretization. We denote the number of V-cycle levels by K. In this test we use $K = \infty$, which means that the V-cycle continues until the coarsest level ($5 \times 5 \times 3$) is reached for each subdomain. A semi-coarsening strategy that best preserves the grid isotropy is chosen, meaning that the grid is restricted only in the horizontal dimensions until the grid spacing is of similar size to the vertical grid spacing. Hereafter all dimensions are restricted. At each multigrid level we use 2 pre- and post-smoothings, and 4 smoothings at the coarsest level. It is important to notice that the smoother is used also at the coarsest grid level and not a direct solver. The relative and absolute timings for each subroutine and for an increasing number of subdomains, $\Omega_P|_{P=1,2,4}$, are reported in Table 4.1. 'Residual (high)' in the first column refers to the 6^{th}-order residual evaluation in the defect correction loop, while 'residual (low)' refers to the 2^{nd}-order linear residual evaluation in the preconditioning phase. A graphical representation of the relative time consumption for Ω_1 is illustrated in Figure 4.8.

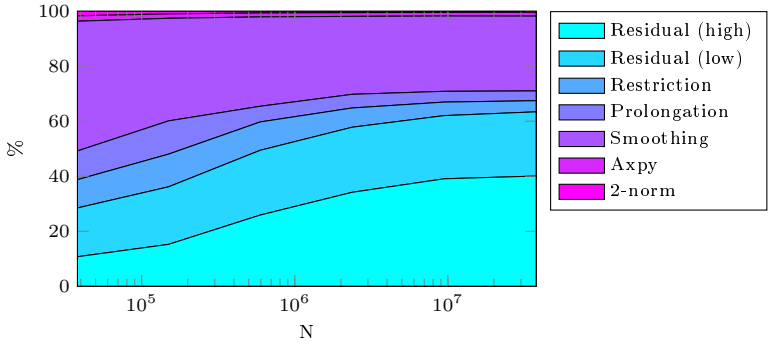


Figure 4.8: Performance breakdown of each subroutine of the preconditioned defect correction method for Ω_1. Tested on *Oscar*.

We note from the numbers in Table 4.1 that increasing the number of subdomains, and thereby the number of GPUs, improves the overall computational times only for problems larger than $513 \times 513 \times 9$. This is acceptable, as in

		129 × 129 × 9		257 × 257 × 9		513 × 513 × 9		1025 × 1025 × 9	
Subroutine	Ω_P	Percent	Time	Percent	Time	Percent	Time	Percent	Time
Residual (high)	1	15.2	0.0010	25.9	0.0036	34.2	0.0133	39.1	0.0529
Residual (low)	1	20.9	0.0014	23.5	0.0033	23.6	0.0092	23.0	0.0311
Smoothing	1	37.3	0.0025	32.5	0.0045	28.3	0.0110	27.4	0.0371
Restriction	1	11.9	0.0008	10.3	0.0014	7.0	0.0027	4.9	0.0067
Prolongation	1	12.1	0.0008	5.7	0.0008	5.0	0.0019	3.9	0.0053
Axpy	1	1.6	0.0001	1.4	0.0002	1.4	0.0005	1.3	0.0018
2-nrm	1	1.0	0.0001	0.7	0.0001	0.6	0.0002	0.4	0.0006
Total	1	100.0	0.0068	100.0	0.0139	100.0	0.0389	100.0	0.1355
Residual (high)	2	6.3	0.0009	10.0	0.0021	17.0	0.0071	27.5	0.0271
Residual (low)	2	18.7	0.0026	19.5	0.0042	22.8	0.0096	22.0	0.0216
Smoothing	2	50.7	0.0070	45.9	0.0099	38.1	0.0160	34.3	0.0337
Restriction	2	15.6	0.0022	14.2	0.0031	14.5	0.0061	9.4	0.0092
Prolongation	2	7.7	0.0011	9.3	0.0020	6.6	0.0028	5.5	0.0054
Axpy	2	0.6	0.0001	0.6	0.0001	0.7	0.0003	1.0	0.0009
2-nrm	2	0.5	0.0001	0.4	0.0001	0.4	0.0002	0.4	0.0004
Total	2	100.0	0.0139	100.0	0.0216	100.0	0.0420	100.0	0.0983
Residual (high)	4	3.7	0.0007	5.2	0.0014	9.9	0.0041	17.8	0.0139
Residual (low)	4	12.6	0.0023	14.1	0.0037	18.0	0.0074	18.9	0.0149
Smoothing	4	64.1	0.0119	59.1	0.0155	51.8	0.0212	42.0	0.0330
Restriction	4	10.8	0.0020	11.5	0.0030	13.2	0.0054	10.5	0.0083
Prolongation	4	7.8	0.0015	9.2	0.0024	6.3	0.0026	9.7	0.0076
Axpy	4	0.4	0.0001	0.4	0.0001	0.5	0.0002	0.7	0.0005
2-nrm	4	0.5	0.0001	0.4	0.0001	0.3	0.0001	0.3	0.0003
Total	4	100.0	0.0186	100.0	0.0262	100.0	0.0409	100.0	0.0784

Table 4.1: Relative and absolute timings for each of the subroutines based on an increasing number of unknowns and subdomains. Tested on *Oscar*.

general there is little or no reason for introducing multiple GPUs to solve a problem that itself fits within the memory of one GPU. Each GPU is massively parallel, thus multiple GPUs increase both load balancing issues and costly communication overhead. Strong scaling, measured in terms of number of GPUs, for problems of moderate sizes should therefore not be expected to be good in general.

Based on the relative numbers we see that in particular smoothing becomes more dominant as the number of subdomains increases. Table 4.2 lists the number of subroutine calls as a function of multigrid levels K. Each of these subroutines requires a ghost layer update. It is evident that even for low K, there are significantly more calls to the smoothing routine compared to the remaining subroutines. It is therefore particularly desirable to decrease the number of restrictions in order to also reduce the number of smoothings and consequently lower communication requirements.

However, multigrid achieves its unique linear work scalability and grid independent convergence properties from the fact that it reduces all error frequencies by smoothing on all grid levels. This is in general important for elliptic prob-

lems where all grid points are strongly coupled. What we will demonstrate by example, is that the coupling between distinct water waves are so weak, that smoothing only on the multigrid levels where the waves are well sampled is sufficient. Our hypothesis is that once the waves are undersampled due to repeated coarsening, there will be little or no effect of further restrictions. Since only the smoother, and not a direct solver, is applied on the coarsest grid level, reducing the total number of grid coarsenings will lead to improve overall performance, provided that the algorithmic convergence is preserved.

Subroutine	$K=1$	$K=3$	$K=5$	$K=7$
Residual (high)	1	1	1	1
Residual (low)	1	3	5	7
Smoothing	8	16	24	32
Restriction	1	3	5	7
Prolongation	1	3	5	7
Axpy*	2	4	6	8
2-nrm†	1	1	1	1

Table 4.2: Number of calls to each subroutine as a function of multigrid V-cycle coarsenings K. * is a local operation and can be computed with no communication. † is a collective operation (MPI_Allreduce), but requires no explicit update of the ghost boundaries.

4.4.4 The performance effect of multigrid restrictions

We now study how the number of multigrid restrictions effect the numerical performance for both the single- and multi-block solver. We use the absolute time per outer defect correction iteration, Γ, as a measure of performance. Thus, these timings are independent of any physical properties within the free surface problem as the algebraic convergence is not considered. Γ is measured and reported in Table 4.3 for a variation of restrictions K and number of subdomains Ω_P. The remainder of the settings are the same as in the previous example.

Two different speedup measures are reported in Table 4.3; Ψ_∞ is the speedup using fewer restrictions: $K = 1, 2, 5$ compared to $K = \infty$, with the same number of subdomains Ω_P. Ψ_1 is the speedup from using multiple subdomains $\Omega_P | P = 2, 4, 8$ compared to Ω_1, with the same number of restrictions K. We see that $\Psi_\infty \geq 1$ and that $\Psi_\infty|_{K=1} \geq \Psi_\infty|_{K=3} \geq \Psi_\infty|_{K=5} \geq \Psi_\infty|_{K=\infty}$, which is to be expected, since the time to compute one V-cycle with $K = i$ is a subproblem of computing a V-cycle with $K = i + 1$. Ψ_1 is a measure of strong scaling with respect to the number of GPUs (subdomains). As mentioned before, multiple GPUs are feasible only for problems of reasonable size. Based on the reported numbers we conclude that there is good potential for improving the overall numerical performance, given that few restrictions (low K) is sufficient for rapid

		$129 \times 129 \times 9$			$257 \times 257 \times 9$			$513 \times 513 \times 9$			$1025 \times 1025 \times 9$		
Ω_P	K	Γ	Ψ_∞	Ψ_1	Γ	Ψ_∞	Ψ_1	Γ	Ψ_∞	Ψ_1	Γ	Ψ_∞	Ψ_1
1	∞	0.0068	1.0	1.0	0.0139	1.0	1.0	0.0389	1.0	1.0	0.1355	1.0	1.0
1	1	0.0042	1.6	1.0	0.0097	1.4	1.0	0.0322	1.2	1.0	0.1218	1.1	1.0
1	3	0.0055	1.2	1.0	0.0119	1.2	1.0	0.0363	1.1	1.0	0.1315	1.0	1.0
1	5	-	-	-	0.0132	1.0	1.0	0.0376	1.0	1.0	0.1336	1.0	1.0
2	∞	0.0139	1.0	0.5	0.0216	1.0	0.6	0.0420	1.0	0.9	0.0983	1.0	1.4
2	1	0.0055	2.5	0.8	0.0108	2.0	0.9	0.0237	1.8	1.4	0.0735	1.3	1.7
2	3	0.0119	1.2	0.5	0.0164	1.3	0.7	0.0335	1.3	1.1	0.0854	1.2	1.5
2	5	-	-	-	0.0216	1.0	0.6	0.0400	1.1	0.9	0.0929	1.1	1.4
4	∞	0.0186	1.0	0.4	0.0262	1.0	0.5	0.0409	1.0	1.0	0.0784	1.0	1.7
4	1	0.0074	2.5	0.6	0.0114	2.3	0.9	0.0185	2.2	1.7	0.0451	1.7	2.7
4	3	0.0153	1.2	0.4	0.0196	1.3	0.6	0.0297	1.4	1.2	0.0590	1.3	2.2
4	5	-	-	-	0.0262	1.0	0.5	0.0377	1.1	1.0	0.0709	1.1	1.9
8	∞	0.0150	1.0	0.5	0.0210	1.0	0.7	0.0308	1.0	1.3	0.0543	1.0	2.5
8	1	0.0069	2.2	0.6	0.0084	2.5	1.1	0.0145	2.1	2.2	0.0288	1.9	4.2
8	3	-	-	-	0.0177	1.2	0.7	0.0230	1.3	1.6	0.0416	1.3	3.2
8	5	-	-	-	-	-	-	0.0308	1.0	1.2	0.0509	1.1	2.6

Table 4.3: Absolute timings per defect corretion iteration and attainable speedups for four different problem sizes. Tested on *Oscar* (Tesla M2050). Time in seconds.

convergence. For example, for a problem size of $(1025 \times 1025 \times 9)$ decomposed into four subdomains and solved on four GPUs Ω_4, there is a 1.3 speedup if three multigrid restrictions ($K = 3$) are sufficient compared to $K = \infty$, and there is a 2.2 speedup compared to the single-block solver. The following test is set up to clarify whether these speedups can in fact be achieved by reducing the number of multigrid levels.

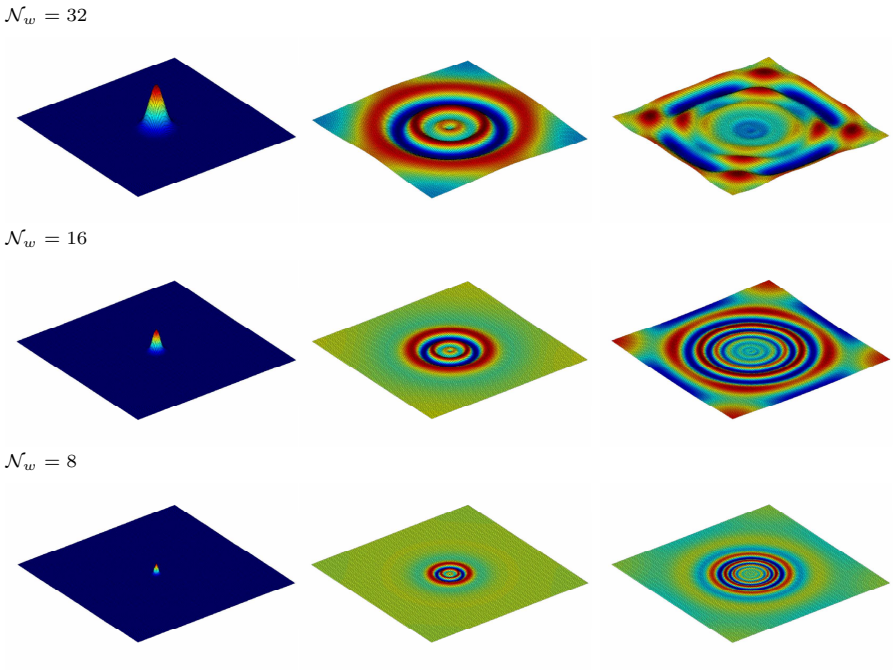
4.4.5 The algorithmic effect of multigrid restrictions

To unify the findings from the previous examples we now introduce and solve a specific nonlinear free surface problem. The purpose of this numerical experiment is to clarify whether the hypothesis of imposing fewer restrictions, and thus fewer smoothings to minimize communication, based on the findings in Table 4.1, can in fact lead to performance improvements as reported in Table 4.3. The final link that we need to demonstrate is whether the algebraic convergence can be maintained with few restrictions. We use a two-dimensional Gaussian distribution as the initial condition to the free surface elevation within a numerical wave tank with no flux boundaries,

$$\eta(x, y, t) = \kappa\, e^{-\frac{x^2+y^2}{2\sigma^2}}, \qquad t = 0, \tag{4.2}$$

where $\sigma = 0.15$ and $\kappa = 0.05$. The wavelength is approximately $\lambda = 1$ and the wavenumber $k = 2\pi$. The distance from the seabed to still water level is $h = 1$

and therefore $kh = 2\pi$, which is intermediate depth. The physical size of the domain is stretched to fit the initial wave with approximately the number of grid points that we wish to examine. The number of grid points per wave is denoted $\mathcal{N}_w$, and we test for four different values $\mathcal{N}_w = 4, 8, 16, 32$. Figure 4.9 illustrates how the nonlinear wave travels within the first seconds with different wave resolutions. We see that the initial wave rapidly propagates throughout the domain and creates multiple waves of various amplitudes and wavelengths.



(a) $T = 0s$ **(b)** $T = 2s$ **(c)** $T = 4s$

Figure 4.9: Nonlinear waves traveling in a closed basin with no flux boundaries, illustrated at three distinct time steps. Horizontal grid dimensions are 129×129. The inital Gaussian wave is discretized with approximatelly $\mathcal{N}_w = 32, 16, 8$ grid points.

At each time stage, the Laplace problem (3.8) is solved to a relative accuracy tolerance of 10^{-4}. Then the number of outer defect corrections are counted. We take sufficient time steps for the average number of iterations to settle within three digits of accuracy. The results are collected in Table 4.4, where the number of grid points per wavelength and number of restrictions are varied. The results are encouraging as they indeed confirm that the number of *useful* restrictions depends strongly on the discretization of the waves. Load balancing between the GPU threads and multiple GPUs will therefore not be an issue as there is no

reason to restrict beyond the wave resolution. The average number of iterations in the table also confirms that convergence is independent on the global grid resolution.

The convergence history in Figure 4.7 confirmed that convergence is independent of the number of subdomains. Thus the average iteration counts reported in Table 4.4 apply, regardless of how many subdomains (GPUs) are used.

		$129 \times 129 \times 9$	$257 \times 257 \times 9$	$513 \times 513 \times 9$	$1025 \times 1025 \times 9$
K	$\mathcal{N}_w$	Avg. Iter.	Avg. Iter.	Avg. Iter.	Avg. Iter.
1	32	19.31	19.30	19.30	19.30
2	32	10.74	10.64	10.64	10.64
3	32	7.41	7.26	7.24	7.24
4	32	5.97	5.85	5.84	5.84
5	32	5.79	5.75	5.75	5.75
∞	32	5.79	5.75	5.74	5.74
1	16	10.95	10.95	10.95	10.95
2	16	7.52	7.51	7.51	7.51
3	16	6.54	6.54	6.54	6.54
4	16	6.46	6.46	6.46	6.46
5	16	6.44	6.44	6.44	6.44
∞	16	6.44	6.44	6.44	6.44
1	8	6.34	6.34	6.34	6.34
2	8	4.78	4.78	4.78	4.78
3	8	4.12	4.12	4.12	4.12
4	8	4.08	4.08	4.08	4.08
5	8	4.08	4.08	4.08	4.08
∞	8	4.08	4.08	4.08	4.08
1	4	4.73	4.73	4.73	4.73
2	4	4.13	4.13	4.13	4.13
3	4	4.12	4.12	4.12	4.12
4	4	4.12	4.12	4.12	4.12
5	4	4.12	4.12	4.12	4.12
∞	4	4.12	4.12	4.12	4.12

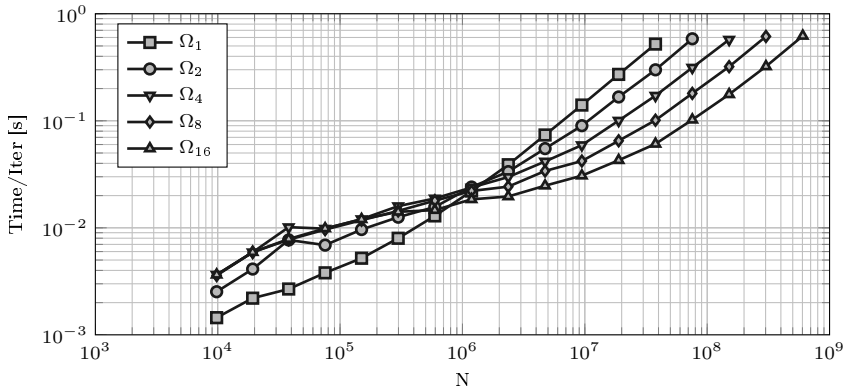
Table 4.4: The average number of iterations for obtaining a relative tolerance of 10^{-4} for various numbers of multigrid levels.

4.4.6 Performance Scaling

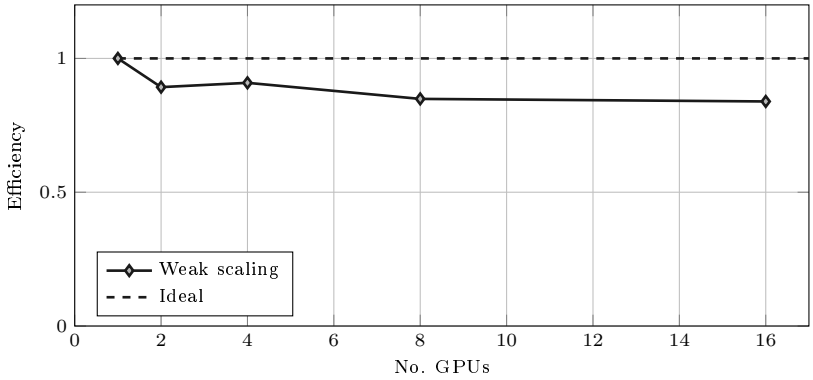
We use again the time per defect correction iteration as a measure of performance. To reduce communication we utilize the multigrid analysis and assume that three multigrid levels are sufficient. The GPU cluster performance results are summarized in Figure 4.10a and 4.10b. The single block timings, Ω_1, evolve as expected: the overhead of launching kernels is evident only for the smallest problem sizes, while for larger problems the time per iteration scales well. From the multi-GPU timings we observe a clear difference when communication overhead dominates and when it does not. Almost exactly after one million degrees

of freedom, communication overhead becomes less significant, and the use of multiple compute units becomes beneficial.

Figure 4.10b illustrates weak scaling relative to one GPU, as the ratio between the number of GPUs and the problem size is kept constant. There is penalty of approximately 15% when introducing multiple GPUs, indicated by the drop from one to multiple GPUs. Hereafter weak scaling remains almost constant and there is no critical penalty for adding extra GPUs.



(a) Absolute timings.

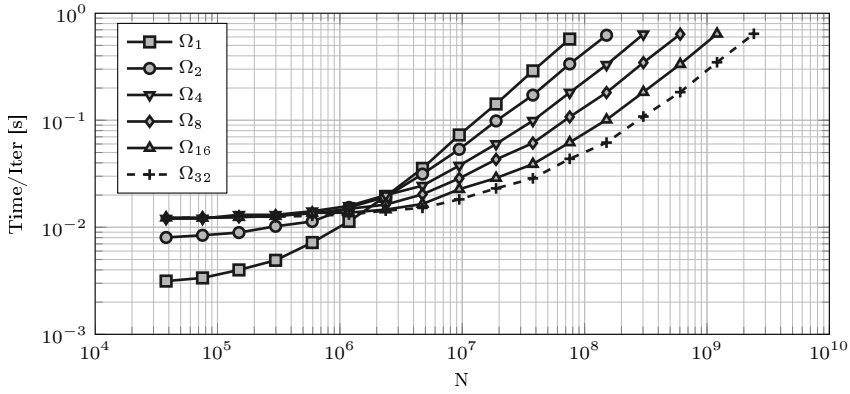


(b) Weak scaling, using $N \approx 3.5 \cdot 10^7$ per GPU.

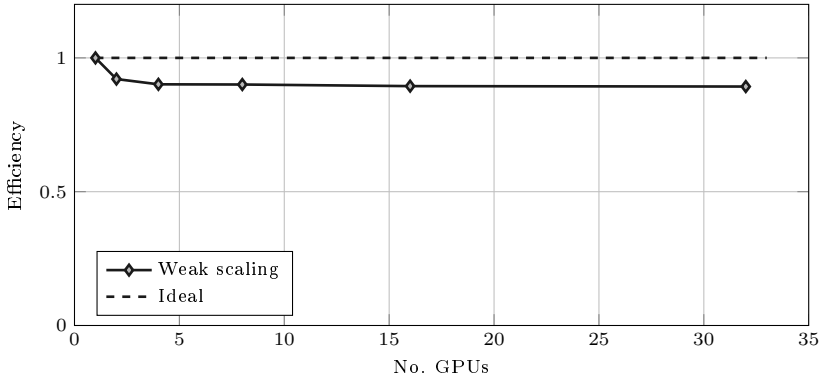
Figure 4.10: Performance scaling for the defect correction iteration, on *Oscar*, single-precision.

For this performance scale test we have gained access to the Stampede cluster at the University of Texas. Stampede is a Dell Linux cluster based on compute nodes equipped with two Intel Xeon E5 (Sandy Bridge) processors, 32GB of host memory, and one Nvidia Tesla K20m GPU, all connected with Mellanox

FDR InfiniBand controllers. The Stampede cluster is configured such that up to 32 GPU nodes can be occupied at once. Performance results are illustrated in Figure 4.11. The configuration is the same as for the previous performance test on *Oscar*. These performance results are even better than those obtained on *Oscar* and we observe weak scaling with less than a 10% efficiency drop. We also notice that with a setup of just 16 GPUs, we are able to solve problems with more than one billion degrees of freedom in practical times. With an approximate time per iteration of $t_{it} = 0.5\,\mathrm{s}$ at $N = 10^9$, it would be possible to compute a full time step in 10 to 20 seconds, assuming convergence in 5 to 10 iterations. With a time step size of $\Delta t = 0.05$, a one minute simulation can be computed in 3 to 6 hours. This is well within a time frame considered practical for engineering purposes. If all 32 GPUs are available, that time would reduce to almost half.



(a) Absolute timings.



(b) Weak scaling, using $N \approx 3.5 \cdot 10^7$ per GPU.

Figure 4.11: Performance scaling for the defect correction iteration, on *Stampede*, single-precision.

4.5 Multi-block breakwater gap diffraction

Breakwaters are built along populated coastal regions, such as beaches and harbors, to reduce and diffract incoming water waves before they reach the coast. Accurate simulation of breakwater diffraction is therefore of engineering interest as a tool for optimizing coastal protection with proper structural designs. Wave diffraction through a breakwater gap transform the water waves into semi-circular wave fronts with the largest wave heights remaining along the direction of incident waves. In these waters it is particularly important to locate unwanted wave accumulation causing damaging effects.

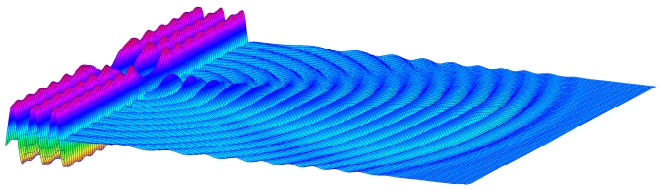


Figure 4.12: Breakwater gap diffraction solution with a gap of size $2L$.

Experimental studies have been carried out by Pos and Kilner [PK87] in 1987 for six different gap configurations, using the breakwater gap-to-wavelength b, as a dimensionless measure. In the same paper they also compare the experimental data to numerical results based on a finite element model with good results that confirm that wave diffraction is primarily a linear wave phenomenon. Numerous numerical studies based on different model equations and discretization strategies have been carried out for analysis of breakwater gap diffraction, see e.g., [EK06]. A common challenge for many numerical models arises due to the presence of the (infinitely) thin breakwater and singularities around the breakwater tip. Ordinary use of ghost points in a finite difference setup is impossible, because such points coincide with internal grid points at the other side of the breakwater. Consequently the user would have to implement ad hoc solutions to circumvent this issue. Numerical treatment of the breakwater tip is also pointed out by Pos and Kilner to be error prone and may be the main reason for the differences between the experimental and numerical data.

In this study our goal is not to contribute with new or thorough analysis of water wave diffraction results, but to demonstrate some of the nice features that the multi-block approach brings. An example of a breakwater gap diffraction

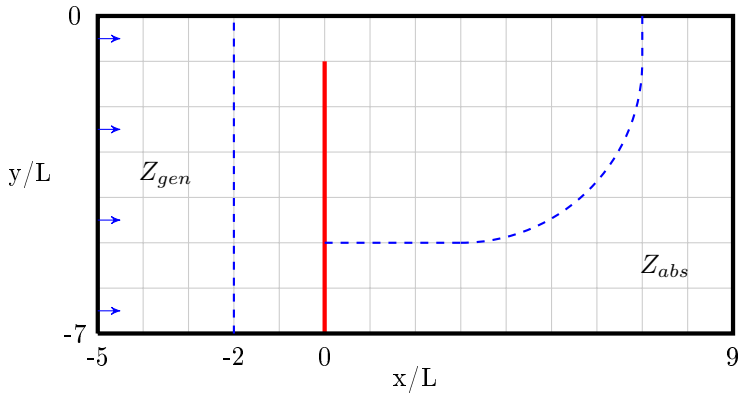


Figure 4.13: Example of a breakwater gap diffraction setup. Incident waves are generated in the generation zone Z_{gen} at the western boundary before encountering the breakwater.

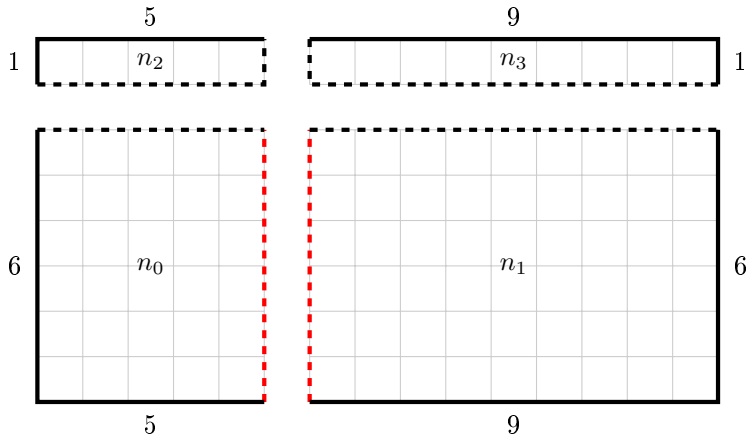


Figure 4.14: Domain decomposition of the breakwater gap diffraction domain from Figure 4.13 into four subdomains. The ghost layer boundary connection (red) between node n_0 and n_1 are deatached and exchanged with a traditional Neumann condition.

setup is illustrated in Figure 4.13, where the breakwater is assumed to be infinitely thin. Symmetry across the direction of incident waves at $y/L = 0$ is utilized to model only half of the domain. Using the multi-block solver with a two-dimensional horizontal topology we are able to split the domain into four subdomains, such that the breakwater becomes the interface between two blocks, see Figure 4.14. Since the boundary conditions can be set individually

for each subdomain, we can easily change the boundary condition for the eastern boundary of subdomain n_0 and western boundary for subdomain n_1 into the classical no flux condition using Neuman boundaries. Manually setting up the subdomain for n_0 using the topology implementation presented in Section 4.2 is illustrated in Listing 4.3.

```
// Setup topology connections and dimensions
topology_type topo;
topo.P = 2; // 2x nodes along x-direction
topo.Q = 2; // 2x nodes along y-direction
topo.R = 1; // 1x node  along z-direction
BC_TYPE bc_east, bc_west, bc_south, bc_north;
if(rank==0) // This is n0
{
    topo.p = 0;              // Grid x-index = 0
    topo.q = 0;              // Grid y-index = 0
    topo.E = MPI_PROC_NULL; // No neighbor to the east (Breakwater)
    topo.W = MPI_PROC_NULL; // No neighbor to the west
    topo.N = 2;              // Node 2 to the north
    topo.S = MPI_PROC_NULL; // No neighbor to the south

    // Boundary conditions
    bc_east  = BC_NEU;
    bc_west  = BC_NEU;
    bc_north = BC_DD;
    bc_south = BC_NEU;

    // Create grid properties here using the bc types ...
}
// ... Do the same for rank=1,2,3 and create grids using topo ...
```

Listing 4.3: Manually setting up a 2×2 topology and the connections for the node with rank$= 0$ (n_0 in Figure 4.14).

The decomposition into individual subdomains solves the problem with insufficient ghost points across the breakwater, because each domain now has an individual set of ghost points that do not need to represent internal grid points. This kind of topology control in combination with boundary fitted domains opens up new perspectives for efficient large-scale simulation of applications with complex scenery.

To test the breakwater gap diffraction model we generate incident monochromatic linear sinusoidal waves within the generation zone $Z_{gen} = 3L$, with wave height of size $H = 0.055m$, wave period $T = 0.59s$, and wavelength $L = 0.495m$. A flat seabed with still-water depth of $h = 0.125$ is used throughout the domain and a gap with a total width of $b = 2L$. For the discretization we use 17 grid points per wavelength, a vertical resolution of $N_z = 9$, and a Courant number $Cr = 0.5$ resulting in a time step $\Delta t = 0.0184s$. In practice we exploit that the numerical solver is very fast, to create a large domain behind the breakwater, large enough for the waves not to reflect and interfere with the solution close to the gap. This also avoids unintended wave reflections from the absorption zones to affect the solution.

Wave diffraction is computed as the size of the wave envelope relative to the incident wave height H_{comp}/H. The diffraction contours are plotted in Figure 4.15 and are in good agreement with the numerical results by e.g., Engsig-Karup [EK06] in the vicinity of the gap. It is again emphasized that these results come with no special treatment or ad hoc solutions to model the breakwater, but is a positive result of a well-designed generic multi-block approach for distributed computing.

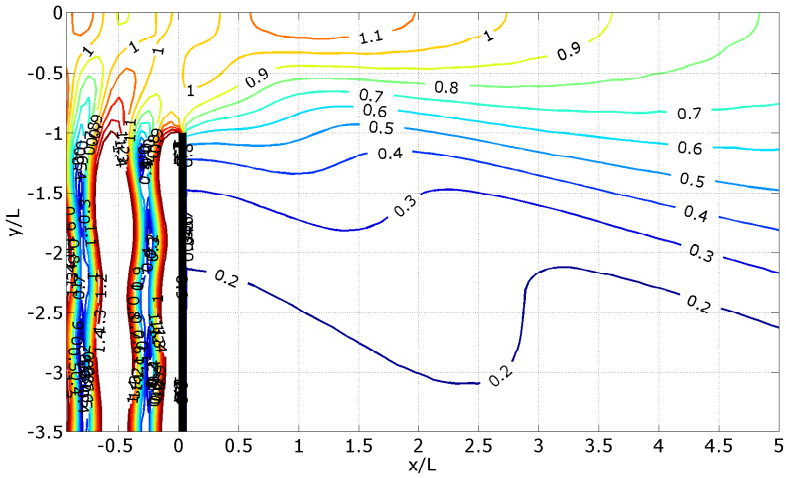


Figure 4.15: Linear diffraction close to the breakwater gap. Values are relative to the incident wave height. The total gap size is equal to two wavelengths, $b = 2L$.

Chapter 5

Temporal decomposition with Parareal

The use of spatial domain decomposition methods is widespread, as they have proven to be efficient for a wide range of problems. These data-parallel methods are efficient for solving large-scale problems and for reducing computational times by distributing data and thereby reducing the work load per processor. However, applications that are facing numerical problems of limited sizes can rapidly reach a speedup limit for a low number of processors, due to a performance decrease when the number of processors increases, as this leads to an increasingly unfavorable ratio between communication and computation. This issue is continuously worsened by the memory wall [ABC+06]; the fact that communication speed is increasing at a far slower pace than compute speed, This trend is even expected to continue for years to come and is considered one of the grand challenges facing development and architectural design of future high-performance systems [Kea11, BMK+10]. Also, there are applications based on ordinary differential equations, where classical domain decomposition methods are not even applicable [Mad08]. For these types of applications, a method for adding parallelism in the temporal domain is of great interest. Decoupling the temporal domain is however, not as straightforward as spatial decomposition, since time is sequential by nature. For this reason these methods have not received the same attention in the literature and have not been proven to work efficiently on as many cases. However, with the continuously increasing number of parallel compute units in modern hardware, alternative parallelization

strategies than data-parallel methods, are becoming more and more attractive.

The book by Burrage[Bur95] in 1995 presents a survey of numerical methods for computing the solution of evolution problems using parallel computers. He classifies these methods into three groups of parallelism - across the system, across the method, and across the time. The first two groups often have strict limitations to their application range and on performance scalability. Parallelism across the method, such as parallel Runge-Kutta methods[IN90], have the disadvantages that they only work with a fixed number of processors and therefore are not suitable for arbitrarily large-scale parallelism. The third category, containing the multishooting methods due to Bellen and Zennaro[BZ89], is of more interest, as these methods are often suitable for a broader range of problems. One such method that introduces concurrency across the time is the Parareal algorithm, proposed by Lion et al. (2001)[LMT01]. Parareal is a parallel iterative method based on *task-parallelism*, via temporal decomposition. Thus, Parareal is an algorithm that is purely designed for parallization, by introducing more concurrency to the solution of initial value problems, as it would only lead to an additional workload in its sequential version. Gander and Vandewalle showed in [GV07] that the algorithm can be written both as a multiple shooting method and as a two-level multigrid-in-time approach, even though the original idea came from spatial domain decomposition. The method has many exciting features: it is fault resilient and has different communication characteristics than those of the classical spatial domain decomposition methods. Fault resilience follows from the iterative nature of the algorithm and implies that a process can be lost during computations and regenerated without restarting the entire computations. Regeneration can be exploited to minimize total run time in case of such temporary hardware failures. This is an attractive feature for HPC systems as the total number of compute nodes continues to grow. Parareal has also been demonstrated to work effectively on a wide range of problems and it is not limited by any number of parallel tasks, thus enabling large-scale parallelism. Also, once the proper distribution infrastructure is implemented, it can be wrapped around any type of numerical integrator, for any type of initial value problem.

Our work on temporal domain decomposition with the Parareal algorithm is a continuation to some of the work presented in [Nie12], where a thorough feasibility study of the algorithm was presented. Parareal results based on our work, on the parallelization of the free surface water wave model, was initiated and based on these experiences we have implemented the Parareal algorithm as a generic library component. By experiments we have demonstrated its applicability to fully three-dimensional free surface water waves. Our main focus is to investigate whether it is possible, based on a number of test cases, to obtain practical speedups with the Parareal algorithm on heterogeneous multi-GPU systems. To the authors knowledge, this is the first time that temporal parallelization using

multple GPUs has been demonstrated.

5.1 The Parareal algorithm

The Parareal algorithm was first presented in 2001, in a paper by Lions et al. [LMT01], and later introduced in a slightly revised predictor-corrector form in 2002 by Baffico et al. [BBM+02]. The Parareal-in-time approach proposes to break the global problem of time evolution into a series of independent evolution problems on smaller intervals, see Figure 5.1. Initial states for these problems are

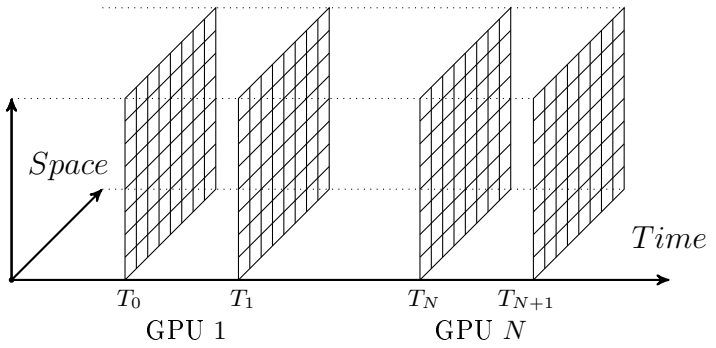


Figure 5.1: Time domain decomposition. A process is assigned to each individual time subdomain to compute the initial value problem. Consistency at the time subdomain boundaries is obtained with the application of a computationally efficient integrator in conjunction with the Parareal iterative predictor-corrector algorithm.

needed and computed by a simple, less accurate, but computationally efficient sequential integrator. The smaller independent evolution problems can then be solved in parallel. The solution generated during the concurrent integration with accurate propagators but inaccurate initial states, is used in a predictor-corrector fashion in conjunction with the coarse integrator to propagate the solution faster, now using the information generated in parallel. We define the decomposition in N intervals, that is,

$$T_0 < T_1 < \cdots < T_n = n\Delta T < T_{n+1} < T_N, \tag{5.1}$$

where ΔT is the size of the time intervals and $n = 0, 1, \ldots, N$. The general initial value problem on the decomposed time domain is defined as

$$\frac{\partial u}{dt} + \mathcal{A}(u) = 0, \qquad u(T_0) = u^0, \quad t \in [T_0, T_N], \tag{5.2}$$

where $\mathcal{A}$ can be a linear or nonlinear operator on u. To solve the differential problem (5.2) we define an operator $\mathcal{F}_{\Delta T}$ that operates on some initial state $U_n \approx u(T_n)$ and returns an approximate solution to (5.2), at time $T_n + \Delta T$. Such an operator is achieved by the implementation of a numerical time integrator, using some small time-step $\delta t \ll \Delta T$ in the integration. The numerical solution to (5.2) can then be obtained by applying the fine propagator sequentially for $n = 1, 2, \ldots, N$.

$$\hat{U}_n = \mathcal{F}_{\Delta T}\left(T_{n-1}, \hat{U}_{n-1}\right), \qquad \hat{U}_0 = u^0. \tag{5.3}$$

For the purpose of Parallel acceleration of the otherwise purely sequential process of computing $\mathcal{F}^N_{\Delta T} u^0 \approx u(T_N)$, we define the coarse propagator $\mathcal{G}_{\Delta T}$. $\mathcal{G}_{\Delta T}$ also operates on some initial state U_n, propagating the solution over the time interval ΔT, now using another time step δT. Typically $\delta t < \delta T < \Delta T$. For the Parareal algorithm to be effective, the coarse propagator $\mathcal{G}_{\Delta T}$ has to be substantially faster to evaluate than the fine propagator $\mathcal{F}_{\Delta T}$. There are many ways of constructing the coarse propagator, the simplest one being to apply the same numerical integrator as for the fine propagator, but taking larger time steps. We refer the reader to [Nie12] for an introduction to other methods. The coarse operator reads

$$\tilde{U}_n = \mathcal{G}_{\Delta T}\left(T_{n-1}, \tilde{U}_{n-1}\right), \qquad \tilde{U}_0 = u^0. \tag{5.4}$$

Using the defined $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ operators, the predictor-corrector form of the Parareal algorithm can be written in a single line as

$$U_n^{k+1} = \mathcal{G}_{\Delta T}\left(U_{n-1}^{k+1}\right) + \mathcal{F}_{\Delta T}\left(U_{n-1}^{k}\right) - \mathcal{G}_{\Delta T}\left(U_{n-1}^{k}\right), \quad U_0^k = u^0, \tag{5.5}$$

with the initial prediction $U_n^0 = \mathcal{G}^n_{\Delta T} u^0$ for $n = 1 \ldots N$ and $k = 1 \ldots K$. N being the number of time subdomains, while $K \geq 1$ is the number of predictor-corrector iterations applied.

5.2 Parareal as a time integration component

The Parareal algorithm is implemented in the GPUlab library as a separate time integration component, using a fully distributed work scheduling model, as proposed by Aubanel [Aub11]. The model is schematically presented in Figure 5.2. The Parareal component hides all communication and work distribution from the application developer. It is generically implemented such that a user only has to decide what coarse and fine propagators to use. Setting up the type definitions for Parareal time integration using forward Euler for coarse

propagation and fourth order Runge-Kutta for fine propagation could then be defined as in Listings 5.1. The number of GPUs used for parallelization depends on the number of MPI processes that execute the application.

```
1 typedef gpulab::integration::forward_euler          coarse;
2 typedef gpulab::integration::ERK4                   fine;
3 typedef gpulab::integration::parareal<coarse,fine>  integrator;
```

Listing 5.1: Assembling a Parareal time integrator using forward Euler for coarse propagation and an explicit Runge-Kutta method for fine propagation

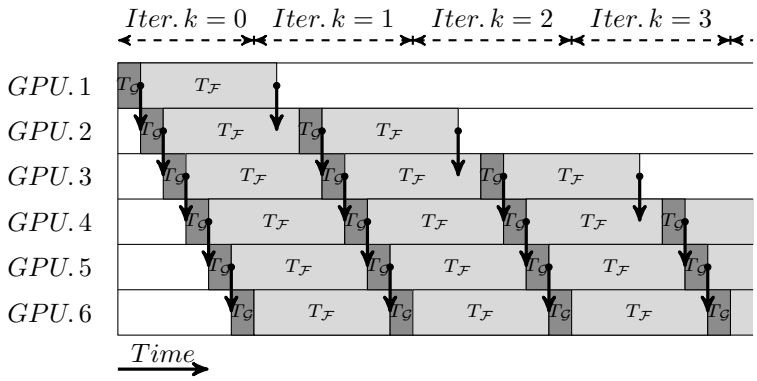


Figure 5.2: Visualization of the fully distributed work scheduling model for the Parareal algorithm. Each GPU is responsible for computing the solution on a single time subdomain. The computation is initiated at rank 0 and cascades through to rank N where the final solution is updated.

5.3 Computational complexity

In the analysis of the computational complexity, we first recognize that both the coarse and the fine propagators, regardless of the type of discretization, involve a complexity that is proportional to the number of time steps being used. Let us define two scalar values $\mathcal{C}_\mathcal{F}$ and $\mathcal{C}_\mathcal{G}$ as the computational cost of performing a single step with the fine and coarse propagators, respectively. The corresponding computational complexity of integrating over an interval ΔT is then given by $\mathcal{C}_\mathcal{F}\frac{\Delta T}{\delta t}$ and $\mathcal{C}_\mathcal{G}\frac{\Delta T}{\delta T}$. R is introduced as the relation between the two; that is, R measures how much faster the coarse propagator is compared to the fine propagator for integrating the time interval ΔT. The total computational

cost for Parareal over N intervals is then proportional to

$$(K+1)\,N\mathcal{C}_{\mathcal{G}}\frac{\Delta T}{\delta T}+KN\mathcal{C}_{\mathcal{F}}\frac{\Delta T}{\delta t}. \tag{5.6}$$

Recognizing that the second term can be distributed over N processors, we are left with

$$(K+1)\,N\mathcal{C}_{\mathcal{G}}\frac{\Delta T}{\delta T}+K\mathcal{C}_{\mathcal{F}}\frac{\Delta T}{\delta t}. \tag{5.7}$$

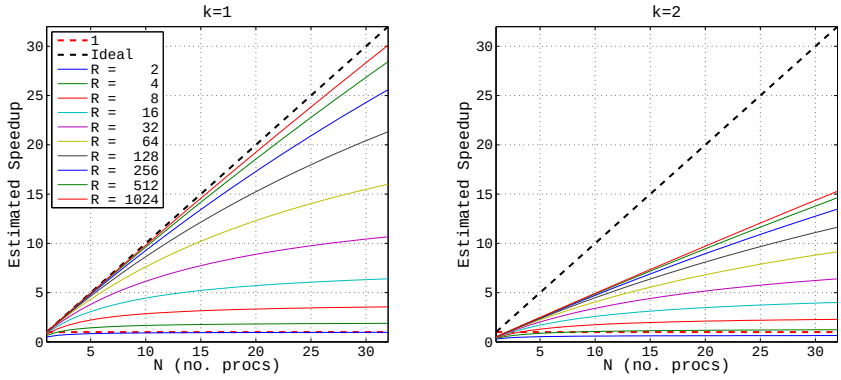
The above should be compared to the computational complexity of a purely sequential propagation, using only the fine operator,

$$\frac{T_N-T_0}{\delta t}\mathcal{C}_{\mathcal{F}}=N\frac{\Delta T}{\delta t}\mathcal{C}_{\mathcal{F}}. \tag{5.8}$$

We can now estimate the speedup, here denoted ψ, as the ratio between the computational complexity of the purely sequential solution, and the complexity of the solution obtained using the Parareal algorithm (5.7). Neglecting the influence of communication speed and correction time, we are left with the estimate

$$\psi=\frac{N\frac{\Delta T}{\delta t}\mathcal{C}_{\mathcal{F}}}{(K+1)\,N\mathcal{C}_{\mathcal{G}}\frac{\Delta T}{\delta T}+k\mathcal{C}_{\mathcal{F}}\frac{\Delta T}{\delta t}}=\frac{N}{(K+1)\,N\frac{\mathcal{C}_{\mathcal{G}}}{\mathcal{C}_{\mathcal{F}}}\frac{\delta t}{\delta T}+K}. \tag{5.9}$$

If we additionally assume that the time spent on coarse propagation is negligible compared to the time spent on the fine propagation, i.e., the limit $\frac{\mathcal{C}_{\mathcal{G}}}{\mathcal{C}_{\mathcal{F}}}\frac{\delta t}{\delta T}\to 0$, the estimate reduces to $\psi=\frac{N}{K}$. It is thus clear that the number of iterations K for the algorithm to converge sets an upper bound on the obtainable parallel efficiency. The number of iterations needed for convergence is intimately coupled with the ratio R between the speed of the fine and the coarse integrators $\frac{\mathcal{C}_{\mathcal{F}}}{\mathcal{C}_{\mathcal{G}}}\frac{\delta T}{\delta t}$. Using a slow, but more accurate coarse integrator will lead to convergence in fewer iterations K, but at the same time it also makes R smaller. Ultimately, this will degrade the obtained speedup as can be deduced from (5.9). Thus, R *cannot* be made arbitrarily large since the ratio is inversely proportional to the number of iterations K needed for convergence. The estimated theoretical speedup for given values of R and K is illustrated in Figure 5.3, it is evident that the efficiency drops rapidly when k increases. This relationship between R and K imposes a challenge in obtaining speedup and is a trade-off between time spent on the fundamentally sequential part of the algorithm and the number of iterations needed for convergence. It is particularly important to consider this trade-off in the choice of stopping strategy; a more thorough discussion on this topic is available in [Nie12] for the interested reader. Measurements on parallel efficiency are typically observed to be less than 50%[Nie12], depending on the problem and the number of time subdomains, which is also confirmed by our measurements using multiple GPUs. This suggests that data-parallel methods

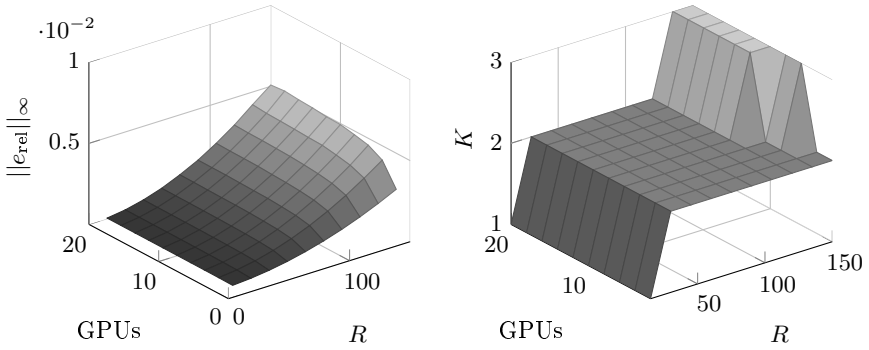


(a) $K = 1$. (b) $K = 2$.

Figure 5.3: Theoretical speedup for two different values of K and an increasing number of processors.

are often a better choice for speeding up PDE solvers, as we have demonstrated in Section 4.4.6. Though, Parareal may have the advantage for problems where there is not enough data for data-parallel methods to be fully effective.

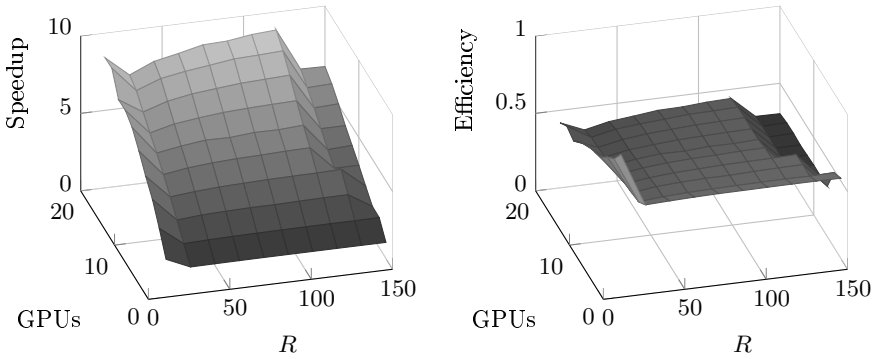
For demonstrative purpose we first present a small parameter study using Parareal on the two-dimensional heat problem (2.4), at a coarse resolution $(N_x, N_y) = (16, 16)$ and for a integration period $t = [0, 1]$ s. The value of R is regulated by using a sufficiently small constant time step size for the fine integrator and then adjusting the time step size for the coarse integrator. We use simple forward Euler integration for both coarse and fine integration. The problem is first solved purely with the fine integrator to be able to compare the solution obtained with the Parareal algorithm. The error after the first Parareal iteration is reported in Figure 5.4a, and the total number of iterations to obtain a solution with a relative error less than 10^{-5} is reported in Figure 5.4b. In Figure 5.5 speedup and parallel efficiency measurements are presented. When R increases, at some point an extra Parareal iteration is required for the relative error to go below the tolerance, as illustrated in Figure 5.4b, this extra iteration clearly impacts the attained speedup and parallel efficiency depicted in Figure 5.5. The results confirm that speeding up a PDE solver is possible on a heterogeneous system, but it also confirms that the attainable speedups depend on several factors and that it can be difficult to predict.



(a) The relative error after one parareal iteration ($k = 1$).

(b) Iterations K needed to obtain a relative error less than 10^{-5}.

Figure 5.4: Parareal convergence properties as a function of R and number of GPUs used. The error is measured as the relative difference between the purely sequential solution and the parareal solution.



(a) Measured parallel speedup

(b) Measured parallel efficiency

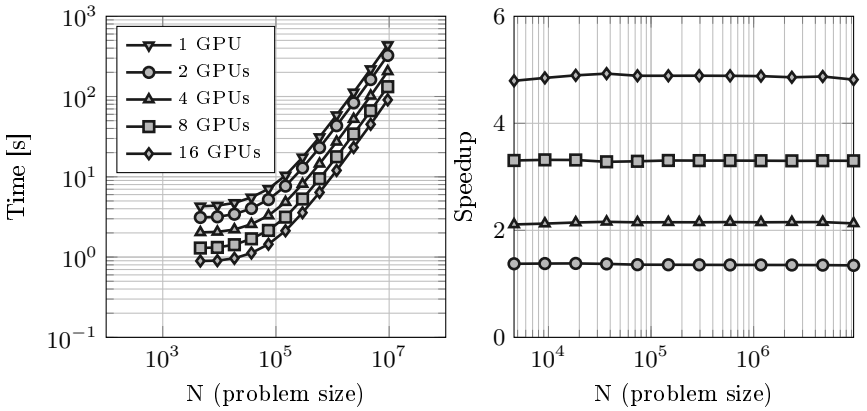
Figure 5.5: Parareal performance properties as a function of R and number GPUs used. Notice how the obtained performance depends greatly on the choice of R as a function of the number of of GPUs. Tested on *Oscar*.

5.4 Accelerating the free surface model using parareal

The Parareal library component makes it possible to easily investigate potential opportunities for further acceleration of the water wave model on a heterogeneous system and to assess practical feasibility of this algorithmic strategy for various wave types.

In section 5.3 it is assumed that communication costs can be neglected and a simple model for the algorithmic work complexity is derived. It is found that there are four key discretization parameters for Parareal that need to be balanced appropriately in order to achieve high parallel efficiency. They are the number of coarse-grained time intervals N, the number of iterations K, the ratio between the computational cost of the coarse to the fine propagator $\mathcal{C}_\mathcal{G}/\mathcal{C}_\mathcal{F}$ and the ratio between fine and coarse time step sizes $\delta t/\delta T$.

Ideally, the ratio $\mathcal{C}_\mathcal{G}/\mathcal{C}_\mathcal{F}$ is small and convergence happens in just one iteration, $K = 1$. This is rarely the case, as it requires the coarse propagator to achieve accuracy close to that of the fine propagator, while at the same time being substantially more efficiently, these two objectives obviously being conflicting. Obtaining the highest possible speedup is a matter of trade-off, typically, the more GPUs used, the faster the coarse propagator should be. The performance of Parareal depends on the given problem and the discretization. Thus, one would suspect that different wave parameters influence the feasibility of the algorithm. This was investigated in [Nie12] and indeed the performance does change with wave parameters. Typically the algorithm works better for deep water waves with low to medium wave amplitude. In this case nonlinear and dispersive effects are minor and only small changes to the wave characteristic happens within each time step.



(a) Performance scaling. **(b)** Speedup.

Figure 5.6: Parareal timings for an increasing number of water waves traveling one wavelength, each wave resolution is (33×9). Speedup for two to sixteen compute nodes compared to the purely sequential single-GPU solver. Tested on *Oscar*.

We have performed a scalability study for Parareal applied to two-dimensional

nonlinear stream function waves to demonstrate that there is a spatial independence on the attainable speedup. Each wave is discretized with $(N_x, N_z) = (33, 9)$ grid points, and the total problem size $N = N_x N_y$ is assembled from multiple waves in a periodic domain. For this test we adjust the time step of the fine and coarse solver such that $R \approx 8$, in which case one iteration $K = 1$, is sufficient. The study shows that moderate speedups are possible for this hyperbolic system, see Figure 5.6. Using four GPU nodes a speedup of slightly more than two was achieved while using sixteen GPU nodes resulted in a speedup of slightly less than five. What should be noticed is the Parareal algorithm is completely insensitive to the size of the problem solved. Parareal is a time decomposition technique, thus scalability applies to the temporal dimension not the spatial. The Parareal algorithm can therefore be a competitive alternative for parallelization of problems of limited spatial sizes. As demonstrated in Figure 5.6, parallel efficiency decreases quite fast for this case when using more GPUs. This limitation is due to the use of a fairly slow and accurate coarse propagator and linked to a known difficulty with Parareal applied to hyperbolic systems. For hyperbolic systems, instabilities tend to arise when using a very inaccurate coarse propagator. This prevents using a large number of time subdomains, as this by Amdahl's law also requires a very fast coarse propagator. The numbers are still impressive though, considering that it comes as additional speedup to an already efficient solver.

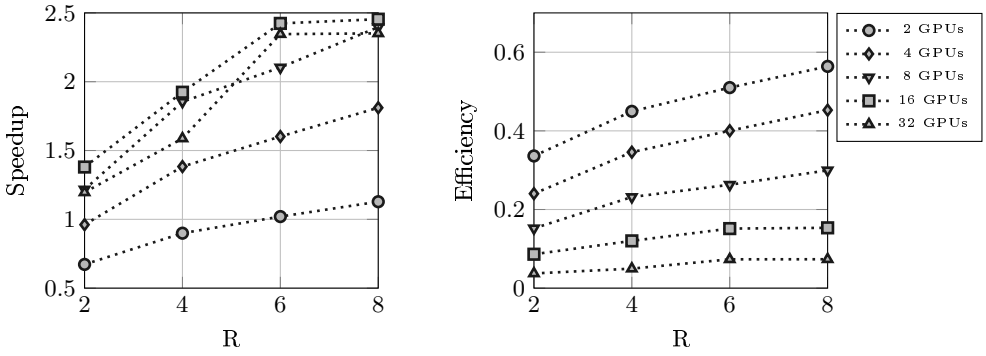


Figure 5.7: Parallel time integration using the Parareal method. R is the ratio between the complexity of the fine and coarse propagators. Tested on *Oscar*.

Performance results for the Whalin test case, as presented in Section 3.3.1, is also reported in Figure 5.7. There is a natural limitation to how much we can increase R, because of stability issues with the coarse propagator. In this test case we simulate from $t = [0, 1]s$, using up to 32 GPUs. For low R and only two GPUs, there is no attained speedup, but for configuration with eight or more GPUs and $R \geq 6$, we are able to get more than 2 times speedup as illustrated in

Figure 5.7. Though these hyperbolic systems are not optimal for performance tuning using the Parareal method, results still confirm that reasonable speedups are in fact possible on heterogenous systems.

5.5 Concluding remarks

The Parareal method is observed to be a potential approach for speeding up small-scale problems due to the reduced communication and overhead involved. For sufficiently large problems, where sufficient work is available to hide the latency in data communication, we find that the spatial domain decomposition method is more favorable, as it does not involve the addition of extra iterations and thereby allows for ideal speedup, something usually out of reach for the Parareal algorithm. An important thing to note here is that it is technically possible to extend the work and wrap the Parareal method around the domain decomposition method, thereby obtaining a combined speedup of both methods. This can be of great interest in the sense that for any problem size, increasing the number of spatial subdomains will eventually degrade speedup due to the latency in communication of boundaries. However, such a combination requires a non-trivial support for handling multiple MPI communicators, which is yet to be supported by the library.

We recognize that some of the results presented for the free surface water wave model were obtained by setting an pessimistic time step for the fine integrator in order to increase R. For explicit time integration of hyperbolic problems there are strict stability requirements on the time step sizes and the coarse integrator cannot violate these. In this work we have only considered fine and coarse integrators based on similar explicit schemes. However, there are options to increase R, by reducing the computational work of the coarse integrator that have not been investigated in present work. We see five possible modifications that will allow a more efficient coarse integration: 1) Use the linearized potential flow system. 2) Use only low-order discretizations. 3) Use a coarser numerical grid, 4) set a less strict tolerance for the solution to the Laplace equation. 5) Use mixed-precision calculations. Though these modifications will be able to significantly speedup the coarse integration part, there are no guarantees that the total time-to-solution will be faster, because it can also lead to extra Parareal iterations. We intend to pursue some of the answers to these questions in future work.

Chapter 6

Boundary-fitted domains with curvilinear coordinates

Spatial discretization based on finite differences is a popular choice within a broad range of scientific applications for several reasons; it is probably the most simple and widely used discretization method available, high-order accuracy is straight forward, consistency analysis is well understood, and the absence of extra index maps reduces the memory requirement, enhances the performance, and possibly increases developer productivity. Furthermore, the one-to-one mapping between discrete grid points and the thread hierarchy of massively parallel processors, such as the GPU, is ideal for high performance throughput, as demonstrated in some of the early articles on heterogeneous computing, e.g., [KDW+06, DMV+08, Mic09]. The above mentioned reasons have been driving motivations for the present work, and to the authors knowledge, the finite difference based free surface water wave solver is now one of the most versatile and efficient tools available for simulation of dispersive and nonlinear water waves.

However, finite difference methods also pose challenges for certain types of applications, particularly because of problems with mass conservation in fluids, and because of the difficulties in representing complex geometries. To addressing these problems, the classes of finite volume or finite element methods in combination with unstructured grids have traditionally been utilized[EGH03]. As an additional step towards a complete and applicable tool for coastal engineering, we address the latter issue by introducing generalized curvilinear

coordinate transformations in order to represent flexible and user-controllable geometries. Curvilinear coordinate transformations is a widely used approach to reduce the limitations of finite difference approximations and have also been used to analyze both linear and nonlinear free surface wave-structure interactions [LF01, SDK+01, BN04, ZZY05, zFlZbLwY12], though only few are concerned with fully three-dimensional models[DBEKF10]. In order to represent the mapping we use a unique one-to-one mapping between all horizontal grid points, from a sufficiently smooth (differentiable) physical region to a logically rectangular region. The coordinate transformation introduces additional terms to the partial differential equations, that will have to be accounted for. We extend the GPUlab library with additional kernel-based finite difference routines that will support developers in computing the solution of coordinate transformed spatial derivatives. Explicit numerical differentiation of the physical coordinates is utilized to allow a generic implementation, that can be used for any valid user-generated input grid. The introduction of algebraic grid transformations reduces the performance throughput because the number of memory transactions is increased. The computation of spatial derivatives based on finite difference approximations are no longer able to rely on constant grid spacing, but has to compute grid-specific transformation coefficients by reading information from a grid that holds the physical information. The challenge is therefore to minimize the additional memory overhead, therefore we present a numerical benchmark that will reveal some of the performance characteristics of the chosen method.

Grid (or mesh) generation can be difficult for highly detailed and complex geometries and therefore developers often rely on third party software to generate the grids. Different grid types exist with various properties to match different geometries and numerical approaches. Either structured or unstructured grids based on triangular, quadrilateral, or tetrahedral elements are among the most used. Fast and optimal grid generation is a topic of its own, and will not be extensively covered in this thesis, instead we refer the reader to the work available in the literature on this subject, e.g., [HX96]. We will comment on some of the issues related to grid generation relevant to the given examples. Many software libraries for grid generation exist and it can be advantageous to rely on such third party packages to avoid cumbersome grid refinements. The test cases presented in this chapter all have boundary-fitted grids that can be computed fairly easy from analytic expressions and so we will not detail the process of grid generation further.

In the following sections we first consider curvilinear transformations of the horizontal coordinates to represent fully surface penetrating and bottom mounted structures. The transformation equations between a physical domain of interest and a classical Cartesian reference domain (computational domain) are presented along with a generic implementation strategy. Examples of different applications are also examined and compared to either analytic or experimental

data. The efficient GPU-based free surface solver allows us to compute the solution of relatively large and complex wave-structure interaction within minutes or up to approximately an hour.

6.1 Generalized curvilinear transformations

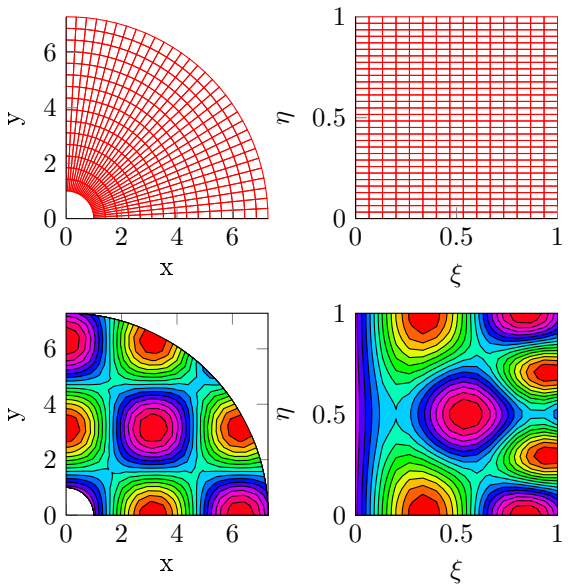


Figure 6.1: Top: One-to-one mapping of a 16×32 discrete grid, representing the quarter annulus in Cartesian coordinates (x, y) to the left and in computational coordinates (ξ_1, ξ_2) within the unit square to the rigth. Bottom: contour plot of a cosine function within the physical domain and its corresponding transformed representation.

We seek to express the relationship between first and second order partial differential equations between the physical space and a computational space. We define the following general relation between a two-dimensional physical domain (x, y) and a time-invariant computational domain (ξ, γ)

$$\xi \equiv \xi(x, y), \quad \gamma \equiv \gamma(x, y), \tag{6.1}$$

such that ξ and γ are independent variables in the transformed computational domain. An example of a transformation between the quarter annulus in the physical domain and the unit squared computational domain is illustrated in

Figure 6.1. If the computational grid is chosen to be rectangular as in the figure, the grid spacing is constant along each dimension. Thus, in the computational domain we are able to reuse the same constant coefficient stencil operators based on finite difference approximations, as we use in a regular (non-curvilinear) setup. However, the coordinate transformation introduces additional terms to the differential equations. Using the above notation and the chain rule for partial differential equations, first order derivatives with respect to the original physical coordinates (x, y) of a function u, can be described in the computational domain (ξ, γ) via the following relations

$$\frac{\partial u}{\partial x} = \frac{\partial \xi}{\partial x}\frac{\partial u}{\partial \xi} + \frac{\partial \gamma}{\partial x}\frac{\partial u}{\partial \gamma}, \tag{6.2a}$$

$$\frac{\partial u}{\partial y} = \frac{\partial \xi}{\partial y}\frac{\partial u}{\partial \xi} + \frac{\partial \gamma}{\partial y}\frac{\partial u}{\partial \gamma}, \tag{6.2b}$$

or simply, using the short notation $\partial u/\partial x = u_x$

$$u_x = \xi_x u_\xi + \gamma_x u_\gamma, \tag{6.3a}$$

$$u_y = \xi_y u_\xi + \gamma_y u_\gamma. \tag{6.3b}$$

These equations contain derivatives with respect to the physical coordinates. In practice we prefer the derivatives to be defined in terms of the computation domain, because it allows us to use constant stencil coefficient operators. In addition, what is known beforehand is usually the mapping from the computational to the physical domain. Thus, since the mapping is required to be unique and any mapping and its inverse would lead to the original point in the starting coordinate system, the following relations hold

$$\xi_x = \frac{1}{J} y_\gamma, \qquad \xi_y = -\frac{1}{J} x_\gamma, \tag{6.4a}$$

$$\gamma_x = -\frac{1}{J} y_\xi, \qquad \gamma_y = \frac{1}{J} x_\xi. \tag{6.4b}$$

where J is the Jacobian, given as the determinant of the Jacobi matrix

$$J = \det(\mathbf{J}) = \det\left(\begin{bmatrix} x_\xi & x_\gamma \\ y_\xi & y_\gamma \end{bmatrix}\right) = x_\xi y_\gamma - x_\gamma y_\xi. \tag{6.5a}$$

The value of the Jacobian determines how much an area under transformation contracts of expands. Given the above equations, the first order derivatives of u can now be described exclusively within the computational reference domain as

$$u_x = \frac{y_\gamma u_\xi - y_\xi u_\gamma}{J}, \tag{6.6a}$$

$$u_y = \frac{x_\xi u_\gamma - x_\gamma u_\xi}{J}, \tag{6.6b}$$

and for the second order and mixed derivatives the following relations can be derived

$$\begin{aligned}
u_{xx} &= (y_\gamma^2 u_{\xi\xi} - 2y_\xi y_\gamma u_{\xi\gamma} + y_\xi^2 u_{\gamma\gamma})/J^2 \\
&\quad + [(y_\gamma^2 y_{\xi\xi} - 2y_\xi y_\gamma y_{\xi\gamma} + y_\xi^2 y_{\gamma\gamma})(x_\gamma u_\xi - x_\xi u_\gamma) \\
&\quad + (y_\gamma^2 x_{\xi\xi} - 2y_\xi y_\gamma x_{\xi\gamma} + y_\xi^2 x_{\gamma\gamma})(y_\xi u_\gamma - y_\gamma u_\xi)]/J^3, \qquad (6.7a) \\
u_{yy} &= (x_\gamma^2 u_{\xi\xi} - 2x_\xi x_\gamma u_{\xi\gamma} + x_\xi^2 u_{\gamma\gamma})/J^2 \\
&\quad + [(x_\gamma^2 y_{\xi\xi} - 2x_\xi x_\gamma y_{\xi\gamma} + x_\xi^2 y_{\gamma\gamma})(x_\gamma u_\xi - x_\xi u_\gamma) \\
&\quad + (x_\gamma^2 x_{\xi\xi} - 2x_\xi x_\gamma x_{\xi\gamma} + x_\xi^2 x_{\gamma\gamma})(y_\xi u_\gamma - y_\gamma u_\xi)]/J^3, \qquad (6.7b) \\
u_{xy} &= [(x_\xi y_\gamma + x_\gamma y_\xi)u_{\xi\gamma} - x_\xi y_\xi u_{\gamma\gamma} - x_\gamma y_\gamma u_{\xi\xi}]/J^2 \\
&\quad + [(x_\xi y_{\gamma\gamma} - x_\gamma y_{\xi\gamma})/J^2 + (x_\gamma y_\gamma J_\xi - x_\xi y_\gamma J_\gamma)/J^3]u_\xi \\
&\quad + [(x_\gamma y_{\xi\xi} - x_\xi y_{\xi\gamma})/J^2 + (x_\xi y_\xi J_\gamma - x_\gamma y_\xi J_\xi)/J^3]u_\gamma, \qquad (6.7c)
\end{aligned}$$

where the derivatives of the Jacobian with respect to the computational coordinates, appearing in (6.7c), can be derived as

$$J_\xi = x_{\xi\xi} y_\gamma - y_{\xi\xi} x_\gamma - x_{\xi\gamma} y_\xi + y_{\xi\gamma} x_\xi, \qquad (6.8a)$$

$$J_\gamma = x_{\xi\gamma} y_\gamma - y_{\xi\gamma} x_\gamma - x_{\gamma\gamma} y_\xi + y_{\gamma\gamma} x_\xi. \qquad (6.8b)$$

The derivations for all of the above equations and their mathematical properties are quite comprehensive and not particularly relevant for the present work. We refer to literature, e.g., the book by Liseikin [Lis99] for a mathematical introduction to grid generation methods, or for a more practical approach, the work by Kopriva [Kop09].

6.1.1 Boundary conditions

Boundary conditions may be defined in terms of the physical boundary normals, such as the no-flux Neumann boundary conditions stating that no fluid is allowed to pass through the boundary,

$$\mathbf{n} \cdot \nabla u = 0, \quad (x, y) \in \partial\Omega, \qquad (6.9)$$

where $\mathbf{n} = (n_x, n_y)^T$ is the normal defined at the physical domain boundary $\partial\Omega$, expressed in Cartesian coordinates. We also need a relation that allow us to approximate (6.9) based on the boundary of the computational domain. We notice that the boundary normals on the regular computational domain are trivial, $n_e = (1, 0)^T$, $n_w = (-1, 0)^T$, $n_n = (0, 1)^T$, and $n_s = (0, -1)^T$ for the east, west, north, and south boundaries respectively. The relationship

between the normals in the two coordinate systems can be described in terms of a rotation matrix R where the following relation holds,

$$\begin{pmatrix} n_x \\ n_y \end{pmatrix} = R^{-1} \begin{pmatrix} n_\xi \\ n_\gamma \end{pmatrix} = \begin{pmatrix} \frac{\xi_x}{\sqrt{\xi_x^2+\gamma_x^2}} & \frac{\gamma_x}{\sqrt{\xi_x^2+\gamma_x^2}} \\ \frac{\xi_y}{\sqrt{\xi_y^2+\gamma_y^2}} & \frac{\gamma_y}{\sqrt{\xi_y^2+\gamma_y^2}} \end{pmatrix} \begin{pmatrix} n_\xi \\ n_\gamma \end{pmatrix}, \tag{6.10}$$

where $n_c = (n_\xi, n_\gamma)^T$ is one of the trivial normals at the boundary of the computational domain. Combining (6.10) and (6.4) we are able to compute the normals at the physical boundary with the same order of accuracy as our finite difference approximations.

6.2 Library implementation

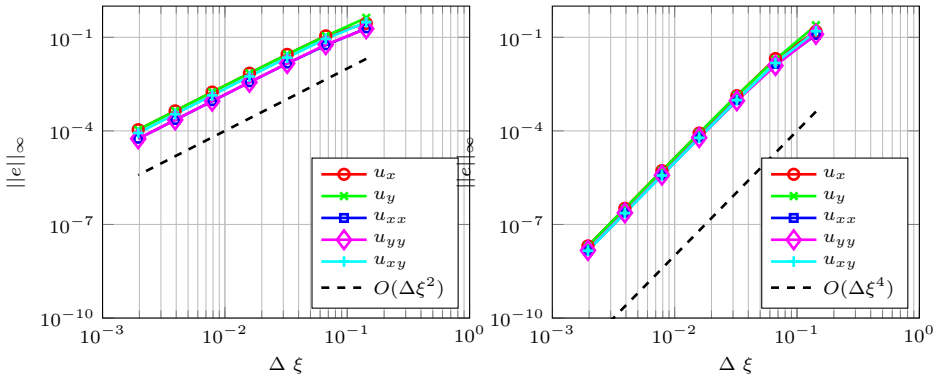
First of all, adding support for curvilinear domains in the GPUlab library must not interfere with existing implementations and should provide an intuitive way of extending solvers to utilize curvilinear domains. To achieve this, we have extended the original grid class with one extra member pointer, pointing to a transformation object. When this pointer is null (default) no curvilinear transformation is associated with the grid, thus existing implementations need no modification. If the pointer is not null, then it points to a simple container that again points to two grids that define the horizontal coordinates of the physical domain (the x and y (6.1)). These two coordinate-grids are of the same object type as the grid itself. Creating a grid object and assigning it a specific transformation is illustrated in Listing 6.1.

```
1  typedef gpulab::grid<double,gpulab::device_memory>  grid_type;
2  typedef grid_type::transformation_type              transformation_type;
3
4  grid_type U;
5  grid_type X;
6  grid_type Y;
7
8  // Fill X and Y with geometric information
9
10 transformation_type transform(alpha);
11 transform.set_X(&X);
12 transform.set_Y(&Y);
13 U.set_transformation(&transform);
```

Listing 6.1: Initialize a grid for the solution (`U`) and two grids for the physical coordinate information (`X` and `Y`).

The transformation object only stores pointers to the two grids to allow multiple solutions to share the same coordinate transformation. This will significantly reduce the memory footprint, as only one copy of `X` and `Y` needs to be stored.

When a grid contains a valid pointer to their transformation object, they should pass along the `X` and `Y` pointers to the GPU kernels, and library subroutines will provide support for computing the spatial derivatives in (6.6) and (6.7). The implementation of the finite difference approximations in curvilinear coordinates is validated in Figure 6.2, based on the quarter annulus grid from Figure 6.1.



(a) $\alpha = 1$ **(b)** $\alpha = 2$

Figure 6.2: Consistency verification for approximating the first and second order derivatives in a curvilinear domain using two different stencil sizes $\alpha = 1, 2$. Results are computed and compared to the analytic known cosine function in the quarter annulus grid as illustrted in Figure 6.1

6.2.1 Performance benchmark

The range of applications that will benefit from accurate representation of complex geometry is greatly increased with the addition of curvilinear grid support. However, as emphasized by the derived transformation equations (6.6) and (6.7), these computations require an increased number of floating-point operations and memory accesses. Thus, a performance decrease is expected for applications relying heavily on the calculation of transformed spatial differential equations. A benchmark of two approaches for computing the first and second order transformed derivatives in one direction is presented. The two versions have distinct advantages, so it is not obvious which is the most optimal.

The first version ($v1$) computes all transformations coefficients based on the x and y coordinates for each grid point. This leads to restricted memory access for these two coordinates, but it also requires each thread to access more than one element in order to perform the stencil computation. The stencil size, α,

determines how many extra memory accesses that are required to compute the derivative. An example of how threads within one thread-block collaborate to read from global memory into shared memory is illustrated in Figure 6.3. The increased number of operations required to compute the derivative also significantly increase the register count, causing the kernel occupancy to drop. All kernel implementations have been evaluated on $G6$, with a Tesla K20c GPU, supporting compute capability 3.5. In this case, there is a decrease from 100% to 75% for the kernel computing the first-order transformed derivatives compared to the non-transformed version. Accordingly, there is a decrease from 100% to 50% for the second-order derivative kernel.

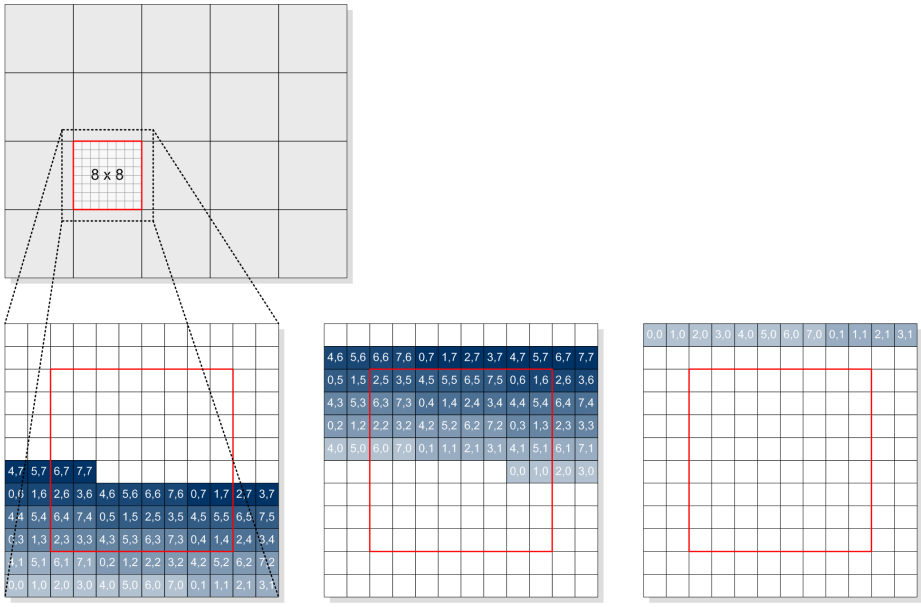


Figure 6.3: An (8×8) thread-block reading global memory into a (10×10) shared memory block. Since $\alpha = 2$ there are two layers on each side of the block that will have to be read from global memory. An (8×8) thread-block is used for illustrative purpose, in pratice a (16×16) thread-block gives better performance.

The second version ($v2$) simply reads the coefficients from pre-computed grids. This version requires additional kernel input arguments. Up to ten extra arguments (x_ξ, x_γ, $x_{\xi\gamma}$, $x_{\xi\xi}$, $x_{\gamma\gamma}$, y_ξ, y_γ, $y_{\xi\gamma}$, $y_{\xi\xi}$, and $y_{\gamma\gamma}$), compared to the two for the first version. However, the second version is somewhat more simple implementation-wise, as there are less on-the-fly computations. The reduced number of registers causes the occupancy to remain at 100%, for the first-order kernel. Computing the second-order derivatives causes the occupancy

to decrease to 50%, because of additional registers required to compute the first-order and cross-derivatives of u according to (6.7). The time it takes to pre-compute the transformation coefficients is not included in the following timings, because these coefficients can be pre-computed once and will then remain constant throughout the application lifetime, assuming the domain is time independent.

For both of the above versions, it applies that increased memory access to the grid u, from which the derivatives are computed, is also increased according to (6.6) and (6.7). A template for the kernels that have been benchmarked is illustrated in Listing 6.2. Best performance has been found with a CUDA kernel configuration of 16×16 threads per thread-block, we refer the reader to [Nvi13] for details on kernel configurations. Performance timings per grid point are plotted in Figure 6.4, as a function of increasing problem size. The numerical domain ratio is kept constant such that $N_x = N_y$ for all tests and a five point stencil ($\alpha = 2$) is used for approximation of all spatial derivatives. The blue lines indicate the time it takes to compute the derivatives in the computational domain, without applying any transformation to it. Thus, it represents a lower bound for the two transformed versions, as they include the same amount of work plus the computation of the additional transformation coefficients. Notice that the blue lines are almost identical for the first- and second-order derivatives ($u_\xi, u_{\xi\xi}$). This is because the number of memory accesses and computations are the same, only the values of the stencil coefficient differs. This is not the case for the transformed derivatives, as the computation of second-order derivatives involves significantly more floating-point and memory transactions.

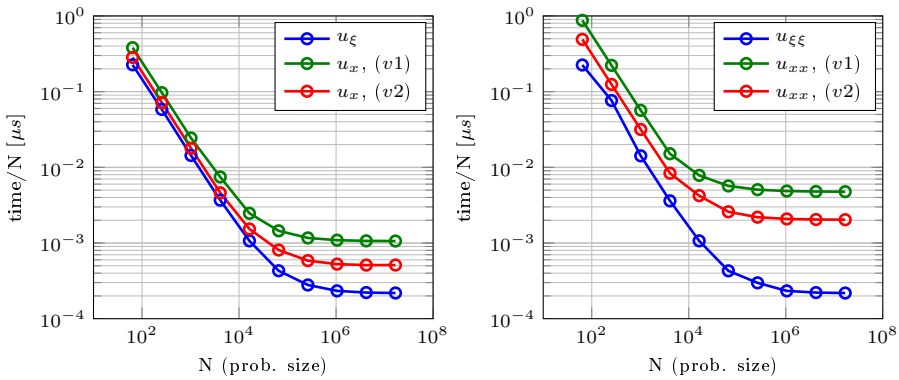
```
__global__
void Ux(double const* u      // Input
      , double* ux           // Output
      , double const* X
      , double const* X_xi1 // [v2]
      , double const* X_xi2 // [v2]
      , double const* Y
      , double const* Y_xi1 // [v2]
      , double const* Y_xi2 // [v2]
      , double dxi1
      , double dxi2
      , int Nx
      , int Ny
      , double const* Dx // Stencil coeffs
      , int alpha)
{
  int i = threadIdx.x*blockDim.x + threadIdx.x;
  int j = threadIdx.y*blockDim.y + threadIdx.y;

  // Shared memory index identifiers
  dim3 T(threadIdx.x+alpha,threadIdx.y+alpha);
  dim3 B(blockDim.x+2*alpha,blockDim.y+2*alpha);

  // Load Dx to shared memory Dxs
  // Load u to shared memory Us
  // [v1] Load X to shared memory Xs
  // [v1] Load Y to shared memory Ys
```

```
28
29    // Only internal grid points
30    if(i>=alpha && i<Nx-alpha && j>=alpha && j<Ny-alpha)
31    {
32      double idxi1 = 1.0/dxi1;
33      double idxi2 = 1.0/dxi2;
34
35      // First order u-derivatives
36      double u_xi1 = FD::FD2D_x(Us,idxi1,T,B,alpha,Dxs);
37      double u_xi2 = FD::FD2D_y(Us,idxi2,T,B,alpha,Dxs);
38
39      // First order X- and Y-derivatives
40      double x_xi1 = // [v1] Compute, [v2] Read
41      double x_xi2 = // [v1] Compute, [v2] Read
42      double y_xi1 = // [v1] Compute, [v2] Read
43      double y_xi2 = // [v1] Compute, [v2] Read
44
45      ux[i+j*Nx] = FD::transformation::FD2D_x(u_xi1, u_xi2, x_xi1, x_xi2, y_xi1
            , y_xi2);
46    }
47  }
```

Listing 6.2: and [v2] refer to code that only applies to version 1 or 2, respectively.]Template for computing the derivative of a two dimensional grid in curvilinear coordinates. [v1] and [v2] refer to code that only applies to version 1 or 2, respectively.

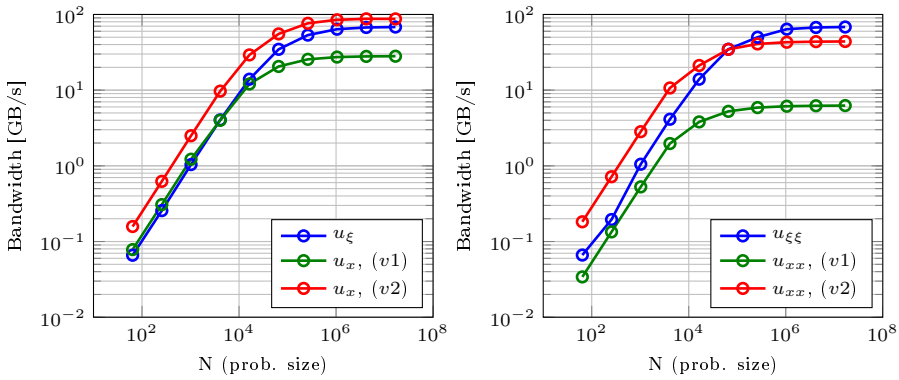
Interestingly, the second version ($v2$), outperforms the first version for all problem sizes. Thus, the increased number of input arguments and distinct memory locations do not slow performance as much as the additional computational work, misaligned memory accesses, and increased register count required by the first version ($v1$). We also recognize the classical two-phase GPU performance characteristic; an intermediate phase where there is not enough work to fully exhaust all processors and a second phase (in this case $N > 10^5$) where the computational time scales strongly with the problem size.

The corresponding bandwidth throughput for each kernel is illustrated in Figure 6.5. The throughput is computed as the number of bytes required to store the discrete grid times the number of effective read/writes. All kernels will be at best memory bound, as the computational work per grid point is constant and relatively low, independent of the problem size. The non-transformed and the second version approach a throughput of 100GB/s, which is relatively close to the efficiently bandwidth on this system (Tesla K), measured to be $\sim$ 140GB/s using the standard CUDA bandwidth test. Thus, these results are satisfactory, taking into account that also the stencil coefficients are loaded from global memory and are not included in the throughput computations. These performance results lead to the–somewhat surprising–conclusion, that for this setup, it is beneficial to pre-compute the transformation coefficients and avoid the extra on-the-fly computations. The results also indicate, that with the introduction of curvilinear coordinate transformation, we should expect up to one order of



(a) First order derivatives. **(b)** Second order derivatives.

Figure 6.4: Absolute timings per grid point for computing the non-transformed (u_ξ, $u_{\xi\xi}$) and transformed (u_x, u_{xx}) derivatives. Timings are based on five point finite difference stencils, $\alpha = 2$. Tested on $G6$, double-precision, and with ECC on.



(a) First order derivatives. **(b)** Second order derivatives.

Figure 6.5: Bandwidth $\alpha = 2$. Tested on $G6$, double precision, and with ECC on.

magnitude performance reduction for the computation of second-order derivatives, which is also what we experience for the potential flow solver. Even with this performance reduction, we expect this approach to be superior compared to unstructured methods, where irregular memory patterns and load balancing can be significant performance barriers.

6.3 Free surface water waves in curvilinear coordinates

The addition of curvilinear coordinates finds many applications in free surface water wave modeling, where the boundary can be fitted to match those of a real scene. It enables complex modeling of off-shore structures, harbors, shorelines, or combinations hereof, that have significant engineering value over traditional regular domains. A flexible representation of the discrete domain can also be utilized to adapt the grid to the wavelengths, such that a higher resolution is used where shorter waves are present and thus minimize over- or under-resolved waves.

The use of curvilinear grids in free surface modeling has been proposed for various numerical models and applications in literature before. A movable curvilinear two-dimensional shallow water model was derived in [SS95] to simulate and study the effect of storm surge flooding in the Bohai Sea. A nonlinear Boussinesq model was later proposed by the same author in generalized curvilinear coordinates, and was applied to several test examples, including the Ponce de Leon Inlet in Florida [SDK+01]. More work on free surface models in curvilinear coordinates are found in [LZ01, BN04, ZZY05, zFlZbLwY12].

6.3.1 Transformed potential flow equations

The kinematic and dynamic free surface boundary conditions in (3.1) along with the σ-transformed Laplace problem contain first- and second-order derivatives in both horizontal directions, described by $\boldsymbol{\nabla} \equiv (\partial_x, \partial_y)^T$. These equations transform due to the curvilinear coordinate transformation according to (6.6) and (6.7), such that,

$$\boldsymbol{\nabla} = \begin{pmatrix} \partial_x \\ \partial_y \end{pmatrix} = \begin{pmatrix} \frac{y_\gamma \partial_\xi - y_\xi \partial_\gamma}{J} \\ \frac{x_\xi \partial_\gamma - x_\gamma \partial_\xi}{J} \end{pmatrix}, \tag{6.11}$$

and

$$\begin{aligned} \boldsymbol{\nabla}^2 = \partial_{xx} + \partial_{yy} =& \frac{1}{J^2}[(y_\gamma^2 + x_\gamma^2)\partial_{\xi\xi} - 2(y_\xi y_\gamma + x_\xi x_\gamma)\partial_{\xi\gamma} + (y_\xi^2 + x_\xi^2)\partial_{\gamma\gamma}] \\ &+ \frac{1}{J^3}[(T_1 + T_3)(x_\gamma \partial_\xi - x_\xi \partial_\gamma) + (T_2 + T_4)(y_\xi \partial_\gamma - y_\gamma \partial_\xi)] \end{aligned} \tag{6.12}$$

where the four variables

$$T_1 = (y_\gamma^2 y_{\xi\xi} - 2y_\xi y_\gamma y_{\xi\gamma} + y_\xi^2 y_{\gamma\gamma}), \tag{6.13a}$$
$$T_2 = (y_\gamma^2 x_{\xi\xi} - 2y_\xi y_\gamma x_{\xi\gamma} + y_\xi^2 x_{\gamma\gamma}), \tag{6.13b}$$
$$T_3 = (x_\gamma^2 y_{\xi\xi} - 2x_\xi x_\gamma y_{\xi\gamma} + x_\xi^2 y_{\gamma\gamma}), \tag{6.13c}$$
$$T_4 = (x_\gamma^2 x_{\xi\xi} - 2x_\xi x_\gamma x_{\xi\gamma} + x_\xi^2 x_{\gamma\gamma}), \tag{6.13d}$$

are all constant under the same coordinate transformation. The free surface solver has been modified according to these new expressions based on the implementation techniques outlined in the previous section. An eigenvalue test has been carried out to confirm the stability of the linearized system in curvilinear coordinates. The linearized system matrix-vector notations is given by

$$\begin{pmatrix} \partial_t \eta \\ \partial_t \tilde{\phi} \end{pmatrix} = \underbrace{\begin{pmatrix} \mathbf{0} & \partial_z \\ -g & \mathbf{0} \end{pmatrix}}_{\mathcal{A}} \begin{pmatrix} \eta \\ \tilde{\phi} \end{pmatrix}. \tag{6.14}$$

We have assembled the full matrix $\mathcal{A}$ for a small problem size of $(N_x, N_y, N_z) = (17, 33, 9)$, based on the coordinate transformation of the quarter annulus from Figure 6.1. The water depth is $h = 1\,\mathrm{m}$ and a symmetric three point stencil is used for approximation of the transformation coefficients. With this discretization, the eigenvalues of $\mathcal{A}$ must be purely imaginary, which is confirmed by the eigenspectra in Figure 6.6.

6.3.2 Waves in a semi-circular channel

To test and verify the capability of the boundary-fitted free surface solver, we demonstrate two classical water wave problems where solutions are available for linear waves and a nonlinear problem where experimental data are provided.

For the first test we propagate linear waves through a semi-circular channel with vertical walls and a constant water depth, for which analytic solutions have been studied, see work by Dalrymple et al.[DKM94]. There exist several numerical results based on curvilinear coordinat transformed Boussinesq-type models[SDK+01, wZsZ10], with finite difference approximations in staggered grids[zFlZbLwY12] and based on mild slope equations as well[ZZY05]. The physical coordinates to the circular channel can be described in terms (ξ, γ),

$$x = (r_1 + \xi(r_2 - r_1))\cos(\pi\gamma), \tag{6.15}$$
$$y = (r_1 + \xi(r_2 - r_1))\sin(\pi\gamma), \tag{6.16}$$

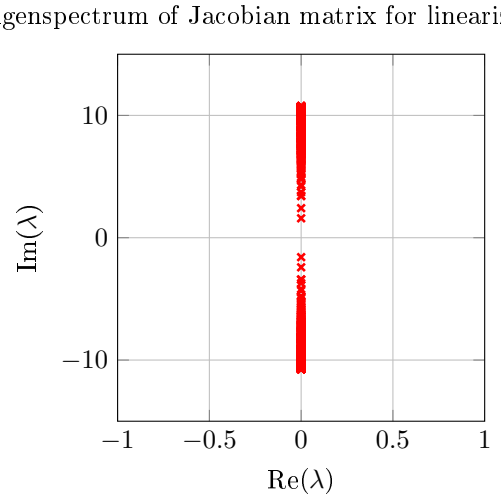


Figure 6.6: Eigenspectrum based on the linearized system with coordinate transformations. The imaginary part of all eigenvalues are less than 10^{-14}.

where r_1 is the inner radius and r_2 is the outer radius. We create a semi-circular channel with dimensions $r_1 = \pi$ and $r_2 = 2\pi$, discretized with a numerical grid of size $(N_\xi, N_\gamma, N_z) = (129, 257, 9)$. The incoming waves are generated with a wavelength of $L = 1\,\text{m}$, the wave depth is $h = 1\,\text{m}$, and a wave period of $T = 0.8\,\text{s}$. The wave height is $H = 0.042\,\text{m}$ corresponding to $(H/L) = 30\%(H/L)_max$. We use 6^{th}-order accurate finite difference approximations and a Courant number $Cr = 0.4$. Waves are generated in a generation zone in front of the channel corresponding to the technique presented in Section 3.2.4. However, to support more flexible generation and relaxation zones we now use a discrete grid function to determine the location of the two zones as illustrated in Figure 6.7. Positive values correspond to generation zones, wheres negative values are absorption zones. We utilize our fast GPU-based solver to create a long channel behind the circular channel to avoid a negative impact from waves reflected from the absorption zone. The total discrete problem size therefore amounts to $(N_\xi, N_\gamma, N_z) = (129, 1025, 9)$.

After the waves are propagated through the channel and settled at a steady state, the wave profile at the inner and outer walls are captured. The numerical wave profiles are depicted in Figure 6.8 together with the analytic solution by Dalrymple. As a measure of the quantitative mismatch between the analytic and the numerical solutions we use the *index of agreement* proposed by Willmott

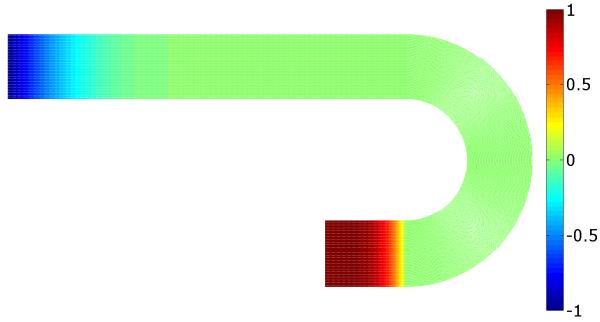


Figure 6.7: Wave generation and absorption zones. The long relaxation channel is used to minimize spurious wave reflections.

in 1981 [Wil81],

$$d = 1 - \frac{\sum_{i=1}^{N} (y_i - x_i)^2}{\sum_{i=1}^{N} (|y_i - \hat{x}| + |x_i - \hat{x}|)^2}, \tag{6.17}$$

where x_i are the true values, y_i are the numerically approximated values, and $\hat{x}$ is the mean value of x_i. The index of agreement d, was developed as a standardized agreement measure for model predictions and varies between 0 and 1, where 1 is a perfect match. Based on the solution wave profiles in Figure 6.8 we get $d_{in} = 0.9992$ and $d_{out} = 0.9989$. These numbers agree well with the results presented both by Zhang et al. [ZZY05] and Shi et al. [SDK+01]. However, due to the high-order finite differences approximation, we have obtained these results with only half the number of grid points at the free surface inside the channel.

An illustrative example of the initial waves propagating into the channel at four distinct stages is given in Figure 6.9. The first waves travel almost in a straight line until they hit the outer wall and are reflected around the bend. A qualitative comparison to the analytic solution is given in Figure 6.10, where there are almost no noticeable differences.

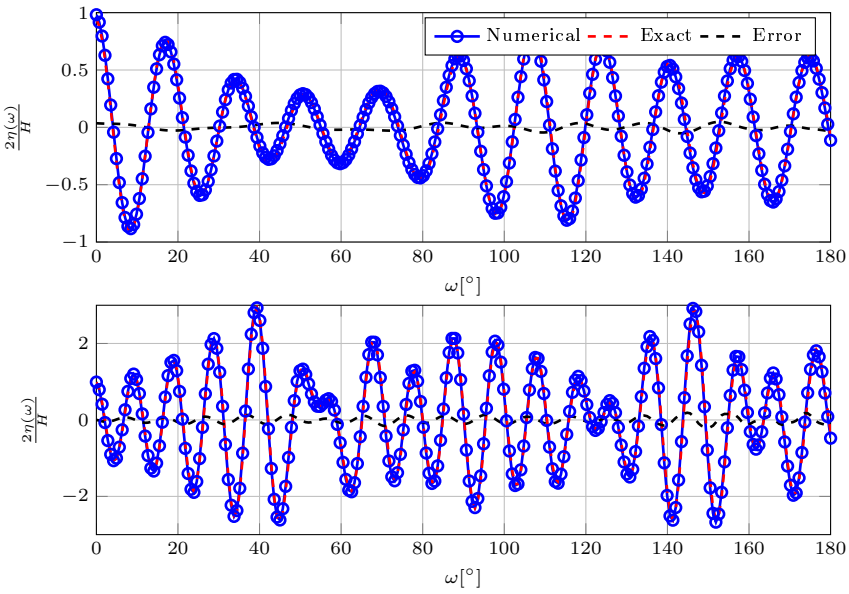
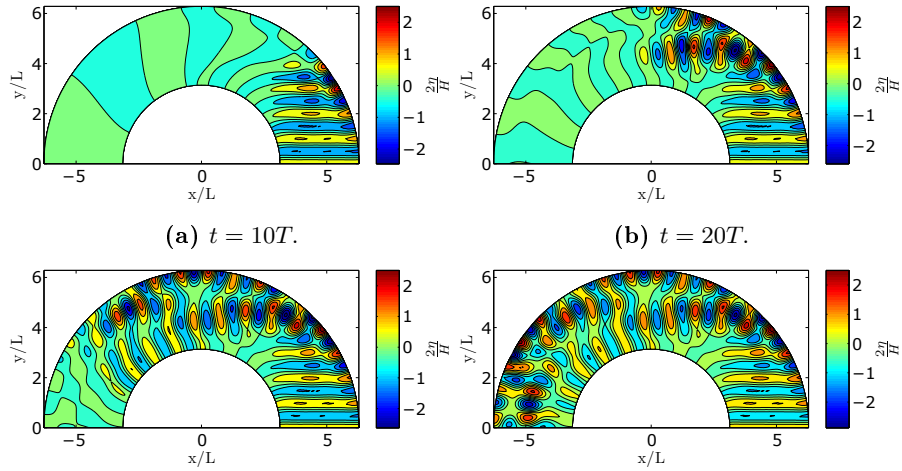


Figure 6.8: The solution at the inner (top) and outer (bottom) walls of the semi-curcular channel at $t = 65T$.



(a) $t = 10T$. **(b)** $t = 20T$.

(c) $t = 30T$. **(d)** $t = 40T$.

Figure 6.9: Propagating wave contours in a semi circular channel at four differnt stages. The waves enter at the southeast corner and are reflected in the northeastern region of the channel creating large wave amplitudes.

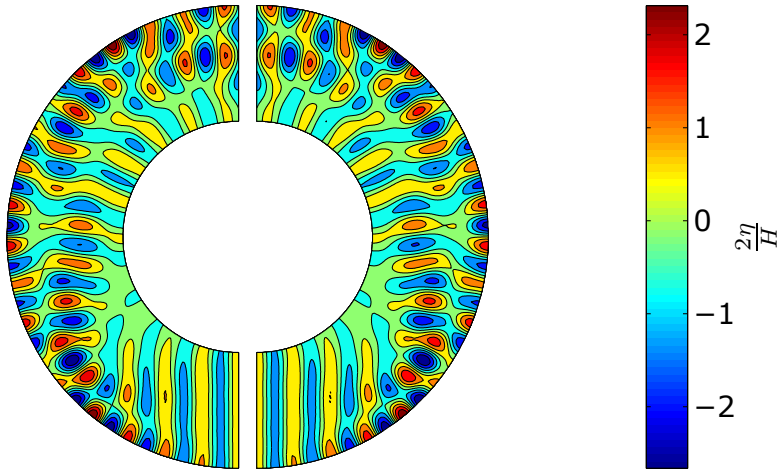


Figure 6.10: Comparison between the analytic (right) and numerical (left) solution at $t = 60T$. High-order approximations with $\alpha = 3$ are used. Visually the match is close to perfect as there are almost no detectable differences.

6.3.3 Wave run-up around a vertical cylinder in open water

Accurate prediction of wave scattering and wave loads around a bottom mounted circular cylinder is a valuable engineering tool for construction design of e.g., offshore windmills or oil rigs. Most often it is desirable to avoid ill-positioned geometries to cause wave amplification close to offshore structures. However, the opposite can also be true, as demonstrated by Hu and Chan (2005) [HC05], where a carefully organized forest of pillars are placed in a lens-shaped array to refract waves into a wave energy converter.

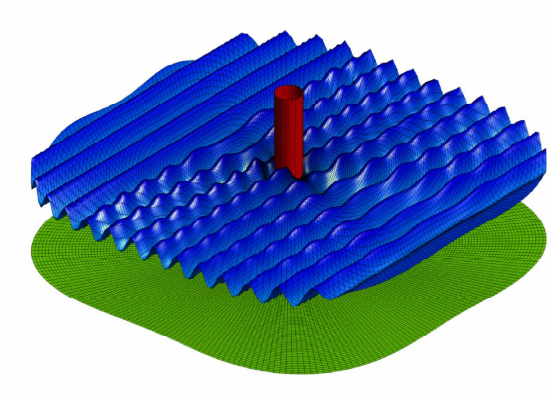


Figure 6.11: Wave run-up around a vertical cylinder.

In this test we consider the wave run-up around a single cylinder surface boundary in open water. This is one of only a few cases including wave-structure interaction, where an analytic solution is known. For linear plane incident waves scattering around a cylinder with flat sea bed, the closed form solution due to MacCamy and Fuchs [MF54] is used for comparison.

As for the breakwater gap diffraction test in Section 4.5, one approach to represent the cylinder is to decomposed the global domain of interest and use the multi-block solver in combination with boundary-fitted domains to reassemble the cylinder geometry. A grid decomposed into three subdomain that reassembles the vertical cylinder well is illustrated in Figure 6.12. Though this approach seems promising, there is a grid singularity that will have to be addressed. There are no discontinuities within each subdomain, but there exists two critical points in this setup, directly at the front and at back of the cylinder. These points are shared between two adjacent subdomains and the transition across the border is discontinuous and the corner finite difference approximations would be wrong, see close up in Figure 6.13. Ad hoc solutions can be implemented to impose the no-flux boundary condition explicitly at the corner points, but we prefer not to make case-specific corrections as they interfere with the generic library design.

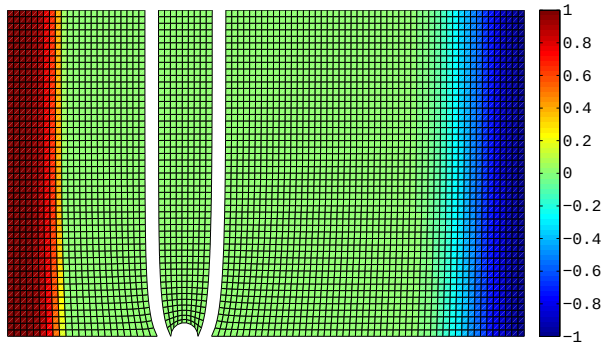


Figure 6.12: Decomposition of the computational domain into three subdomains. The colors indicate wave generation (red) and wave absorption (blue). The grid is coarsened in order to enhance the visual presentation.

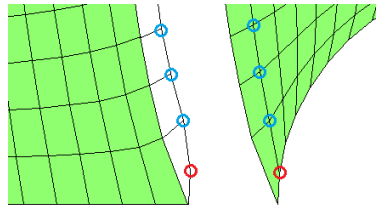


Figure 6.13: Ghost point overlap at the front of the cylinder. The blue points are continuous across the interface. The red point creates a discontinuous transition accross the interface.

Instead we use a modified version of the circular coordinates,

$$x = (r_1 + \xi(r_2 - r_1)\cos(\gamma\pi)/2)\,(1 + (\xi(1+\omega) - \omega\cos(4\gamma\pi))\,, \tag{6.18}$$

$$y = (r_1 + \xi(r_2 - r_1)\sin(\gamma\pi)/2)\,(1 + (\xi(1+\omega) - \omega\cos(4\gamma\pi))\,, \tag{6.19}$$

where ω controls how much the circular channel is stretched, we have uses $\omega = 0.2$ in for the frid illustrated in Figure 6.14. To avoid spurious wave reflections we again set up a computational domain that is large enough to prevent wave reflections to return before the waves are fully evolved at the vicinity of the cylinder. A cylinder centered at $(0, 0)$ with a radius of $a = 0.5m$ is used along with a domain of physical dimensions $L_x \approx 20\,\text{m}$ and $L_y \approx 10\,\text{m}$.

Linear plane waves are generated with wavelength $L = 1\,\mathrm{m}$ over a flat sea bed with a water depth of $h = 1/(2\pi)\,\mathrm{m}$ to give a constant dimensionless depth $kh = 1$. The wave generation absorption zones are illustrated in Figure 6.14. A time step of $\Delta t = 0.01$ is used to ensure stable behavior close to the cylinder together with a resolution of $(N_\xi, N_\gamma, N_z) = (1025, 129, 9)$ and a 6^{th}-order finite difference approximations.

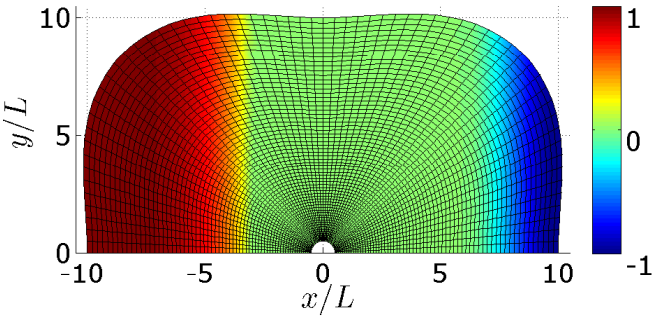


Figure 6.14: Numerical grid and relaxation zones for the vertical cylinder in open water. The grid is coarsened in order to enhance the visual presentation.

As a measure of the wave load that would impact the cylinder, the maximum wave crest around the cylinder is recorded. For comparison we show the computed and the analytic wave envelopes in Figure 6.15. If we again use the index of agreement (6.17), we get an almost perfect match of $d = 0.9999$. The contour plot and the polar coordinate representation of the wave run-up also confirm the good match with the analytic solution in Figure 6.16.

The influence of nonlinear diffraction on bottom mounted cylinder structures is demonstrated by Kriebel[Kri90, Kri92], to be of significant importance, where the inclusion of second-order Stokes theory is found to change the wave amplitudes around the cylinder predicted by linear theory with up to 50%. Results also indicate that Stokes second-order theory is insufficient to fully capture all nonlinear effects present in their experimental results for steep waves. We repeat the experiment with one of the experimental setup as proposed by Kriebel in [Kri92], numerical results are also presented in [DBEKF10]. The cylinder

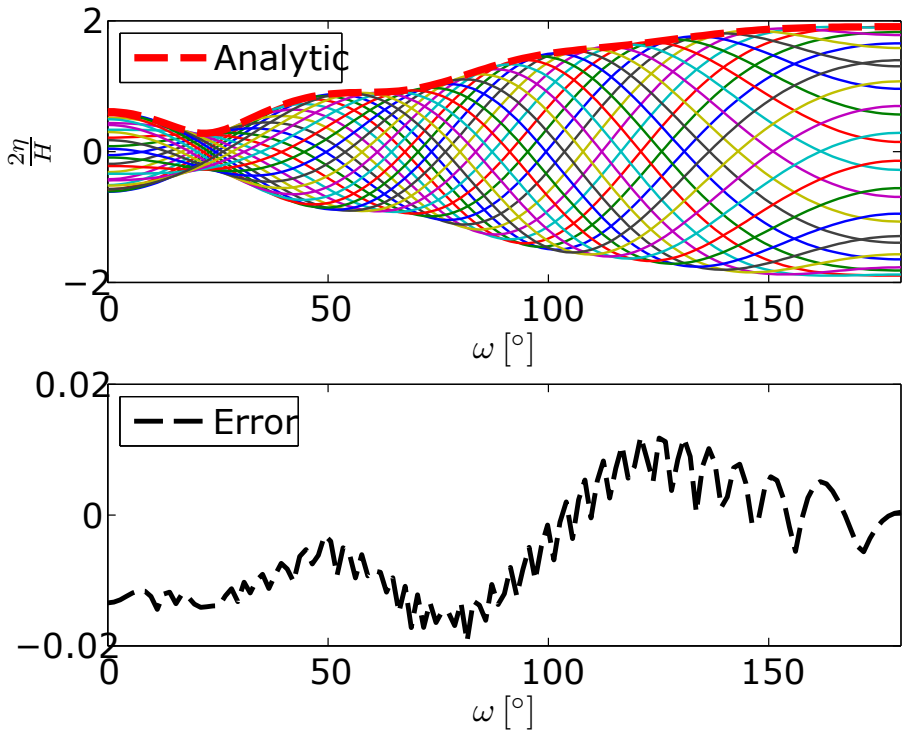
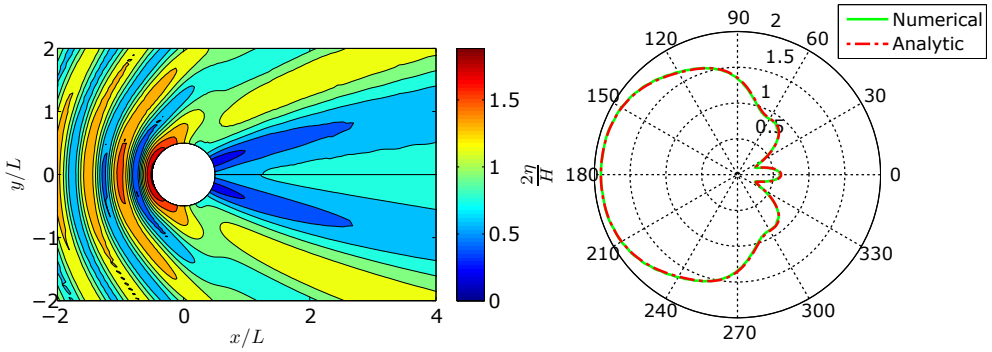


Figure 6.15: Linear wave run-up around the vertical cylinder relative to the incident wave height. The left side of the plot (low ω) represents the back side of cylinder. There is a good match between the numerical envelopes and the analytic solution.

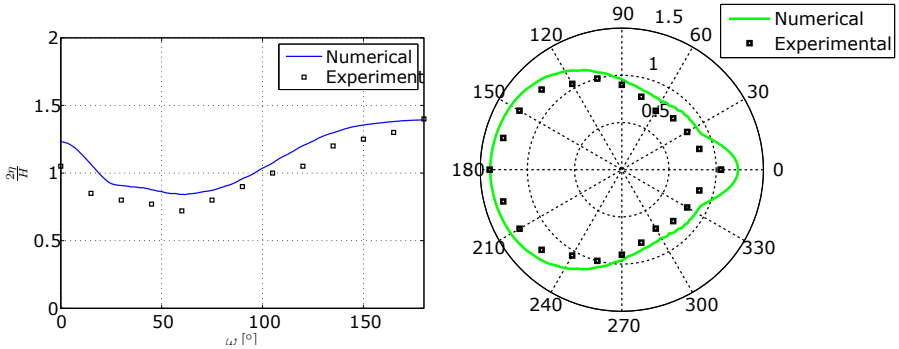
radius is $a = 0.1625\,\mathrm{m}$ and the water depth is $h = 0.45\,\mathrm{m}$. Nonlinear incident waves of length $L = 2.730\,\mathrm{m}$, and wave height $H = 0.053\,\mathrm{m}$ are generated, such that $kH = 0.122$. The same numerical grid is used as for the linear test. The nonlinear wave run-up around the cylinder is illustrated in Figure 6.17. Our numerical solution predicts a slightly larger wave run-up than the experimental data. This is however in agreement with the numerical results presented both by [Kri92] and [DBEKF10].



(a) Maximum wave envelope contours. Top: numerical solution. Bottom: analytic solution.

(b) Wave run-up in polar coordinates.

Figure 6.16: The linear wave envelopes relative to the incident wave heights.



(a) Wave run-up.

(b) Polar

Figure 6.17: Nonlinear wave run-up around the cylinder. $kH = 0.122$, $kh = 1.036$, $ka = 0.374$. Experimental data from Kriebel [Kri92].

6.4 Concluding remarks

The introduction of generalized curvilinear coordinates has significantly increased the range of applications that can benefit from boundary-fitted domains to better reassemble real engineering problems. With some classical examples we have demonstrated how the free surface water wave solver can be used to simulate both linear and nonlinear waves in complex scenery. Combined with the multi-block solver to solve large-scale problems, as indicated by the performance scaling results in Section 4.4.6, the present free surface tool offers some

unique opportunities for fast and accurate assessment of coastal engineering applications.

We point out, that we have experienced a degradation of the algorithmic performance of the Laplace solver for some of the boundary-fitted examples. The number of preconditioned defect correction iterations has increased between 50% to 100% for the above examples. We know that the most efficient multigrid coarsening strategy is based on coarsening along those dimensions that will best preserve the physical grid isotropy. When working on the computational domain, the multigrid coarsening strategy selects the coarsening dimensions based on $\Delta\xi$ and $\Delta\gamma$ and not the physical properties Δx and Δy. However, due to the curved coordinate lines, Δx and Δy are not constant, and it can be impossible to select a coarsening strategy that always preserves isotropy in the global domain. This remains a challenge, and more analysis has to be carried out to fully understand the impact.

Chapter 7

Towards real-time simulation of ship-wave interaction

A solid foundation for further development and collaborations with industry has been established with the implementation and demonstration of the generic and high performance free surface water wave simulation tool. As motivated in the former chapters, the portable GPUlab library and the efficient free surface solver, accurately accounting for dispersive and nonlinear effects can be a valuable tool in many aspects of marine engineering. The remainder of this chapter is dedicated to the research that has been carried out in close collaboration with FORCE Technology, a Danish approved Technology Service (GTS) company with leading world-wide assets in maritime technologies.

The joint collaboration focuses on the development of a ship hydrodynamic model for real-time simulation, including ship-wave and ship-ship interaction as part of a full mission marine simulator. Such full-scale marine simulators are used for educating and training of marine officers and maritime engineering. Accurate and realistic interaction with ship generated wave forces are important to maintain as realistic an environment as possible. Effects such as the forces that occur when two ships approach each other are particular critical and important for safely maneuvering of tugboats. Current state-of-the-art hydrodynamic

models implemented in full-mission simulators are based on fast interpolation and scaling of experimental model data and the results are therefore limited by the amount and accuracy of the available data. The main challenge for real-time ship-wave interaction is to find a proper balance between an approximate representation of ships that maximizes computational performance, but it also gives more accurate and reliable results compared to previous interpolation methods. There is–to the author's knowledge–no other tools that are able to overcome the real-time restriction with a model as accurate as the one proposed. Thus, this will be pioneering work and a unique opportunity for FORCE Technology to be first movers on an international marked.

The collaboration with FORCE Technology has led to an intermediate step on the path towards achieving real-time ship-wave interaction. Research at FORCE Technology continues to be active and the following sections demonstrate a selection of the most important findings achieved until now. The generic and component-based GPUlab library significantly improved the developer productivity for the extension of OceanWave3D into supporting ship-wave and ship-ship interactions.

7.1 A perspective on real-time simulations

With the present GPU-based implementation of the OceanWave3D solver, real-time simulations should be within reach and the solver would be suitable for these applications as it is both scalable, efficient, and robust. The potential flow model is also very attractive since it accurately accounts for dispersive waves from deep to shallow waters, setups that will be relevant in any marine simulator. In previous work we roughly estimated the time to compute a wave period for various wave resolutions[EKMG11], from which we concluded that real-time simulations can be achieved within an additional speedup factor of approximately one order of magnitude for three dimensional problems with one million degrees of freedom. Let us reconsider some of these approximations and include recent hardware trends to better estimate and predict future capabilities. If we wish to estimate the compute time t it takes to compute one wave period T, as a function of time per defect correction iteration $\mathcal{I}_t$, we get,

$$\frac{t}{T} \approx \mathcal{I}_t \, K, \qquad K \approx \frac{\mathcal{S}_{RK} \, \mathcal{I}_{avg} \, N_{PPW}}{C_r}, \tag{7.1}$$

where $\mathcal{S}_{RK}$ is the number of Runge-Kutta ODE solver stages, $\mathcal{I}_{avg}$ is the average number of defect correction iterations per solve, N_{PPW} is the number of points per wavelength, and C_r is the Courant number.

The value of K depends heavily on the wave characteristics, but we are able to estimate a lower and upper bounds based on a few assumptions. The number of Runge-Kutta stages is constant for this method, at $\mathcal{S}_{RK} = 4$. We will also assume a constant Courant number $C_r = 0.8$. Nonlinear wave effects in combination with the water depth determine the algorithmic convergence rate of the preconditioned defect correction method together with the stopping criteria [EK14]. The average number of iterations for mildly to fully nonlinear waves and a tolerance of 10^{-5} are usually within the range $\mathcal{I}_{avg} = 4$-15. The number of points per wavelength should be set according to the order of the finite difference approximations and will influence the accuracy of wave dispersion. Reasonable results are often achievable with $N_{PPW} = 8$-16. In combination these numbers give us the following estimates $K_{low} = 160$ and $K_{high} = 1200$. In order to achieve real-time simulations it is evident that the time to compute one wave period is less than the wave period itself, i.e., $t < T$, which leads to,

$$\mathcal{I}_t < \frac{1}{K}. \tag{7.2}$$

For a given K we can now compute $1/K$ and compare it directly to the performance measurements. Timings based a high-end consumer GPU, GeForce GTX590 is illustrated in Figure 7.1. We can immediately see that real-time is achievable only for low values of K, thus, when there is fast convergence and few points per wavelength, and for resolutions up to $3 \cdot 10^5$ degrees of freedom. That would cover a fully three dimensional simulation with e.g, $257 \times 129 \times 9$ degrees of freedom. As demonstrated in Chapter 4, even the multi-GPU setup will not improve performance at these resolutions. From the figure we see that there is almost no difference between high- and low-order discretizations in the region feasible for real-time computations, which only favors high-order discretizations further, because K would be able to be smaller for high-order discretizations, due to a less strict requirement on the number of points per wavelength.

For practical ship hydrodynamics setups in a numerical wave tank, the tank would have to be at least $3Lpp$ long in the sailing direction and $1Lpp$ in the transverse direction, where Lpp is the ship length. The number of points per ship length is then determined by the wavelength l, generated by the ship, and the number of points per wavelength N_{PPW}, which leads to the following estimate for total number of degrees of freedom,

$$N \approx 3 \left(\frac{Lpp}{l} N_{PPW} \right)^2 N_z, \tag{7.3}$$

where N_z is the number of vertical grid points. As an example, consider a large tanker of length $Lpp = 270\,m$ sailing with an intermediate speed, such that the Froude number $Fr = 0.08$, defined as $Fr = U/\sqrt{g\,Lpp}$, where U is the sailing speed. Then the waves generated by the tanker will be of length $l \approx 11\,m$

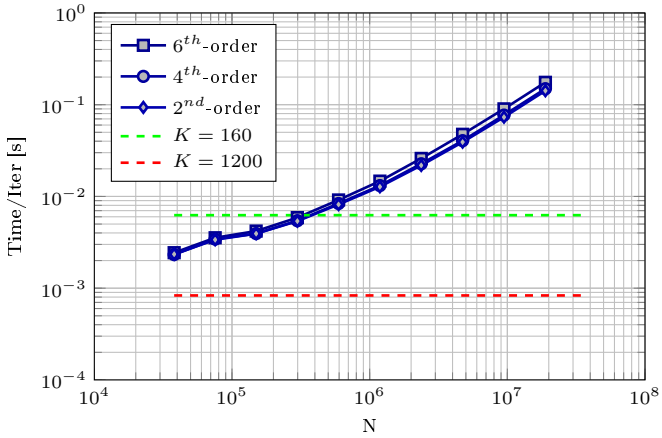


Figure 7.1: Real-time perspectives based on the two extreme values of K and the defect correction iteration time $\mathcal{I}_t$. Using the GeForce GTX590 ($G4$), single precision, MG-RBZL-1V(1,1).

resolved with e.g., $N_{PPW} = 10$ grid points. The total degrees of freedom, assuming that 9 vertical grid points are sufficient, will then be $N \approx 1.6 \cdot 10^6$. Figure 7.1 demonstrates that this is not within the real-time limit for the given setup. However, for $N \approx 1.6 \cdot 10^6$ we see that $\mathcal{I}_t \approx 0.02$, from which we can conclude that K can be no more than 50 for this to be real-time, or in other words, we need at least another speedup factor of 3.2 to achieve the goal of real-time computations with the given setup.

It is now tempting to ask *when* will it be possible to do real-time computations for these hydrodynamic ships models? One should be careful when speculating about future hardware trends, but if we assume the following: 1) The solver is completely memory bound. 2) No further optimizations are made. 3) The GPU memory bandwidth increases with the same rate as the previous generations. Then we should be able to make an estimated guess. Comparing two consecutive generations of GPUs, e.g., the GeForce GTX480 and GTX580 or Tesla C2070 and K20, there is a bandwidth increase of approximately 40%. If this trend continues, a 3.2 speedup is achievable within two to three GPU generations, which historically translates to a maximum of five years.

Though all these numbers are based on rough estimates and heavily depends on the wave characteristics and ship hydrodynamics, they *do* indicate that within a reasonable near time horizon, interactive free surface water wave simulations with engineering accuracy will be a reality. This suggest that now is a good time to invest and pursue this goal.

7.2 Ship maneuvering in shallow water and lock chambers

A feasible model for fast and accurate ship hydrodynamics has been developed and improved continuously during the collaboration period. The first functional model was developed and evaluated for the 3^{rd} *International Conference on Ship Maneuvering in Shallow and Confined Water*, with non-exclusive focus on ship behavior in locks [VDM12, LGB+13]. The objective was to reconstruct numerically a set of experiments carried out with an 1/80 scale laboratory model as seen in Figure 7.2. The experiments reassemble a large 12.000 TEU container carrier entering the newly designed locks in the Panama canal. Experimental details are available in [VDM12].

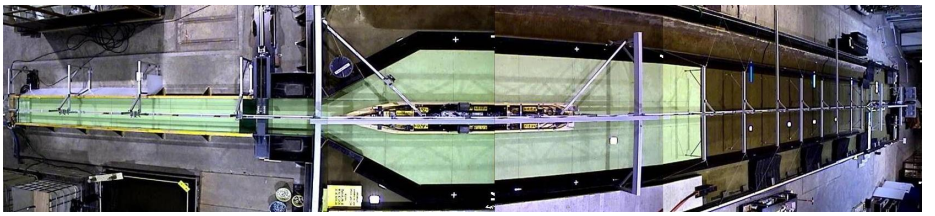


Figure 7.2: Experimental setup of a large vessel entering a lock chamber in the new Panama Canal. 1/80 scale.

In order to maximize performance the first version of the ship hydrodynamic model is based on potential flow theory with *linear* free surface boundary conditions and 2^{nd}-order finite difference approximations. The motivation for these simplifications is to first produce a proof-of-concept implementation that will be stable and have the best possible performance before introducing the more advanced nonlinear parts. In future versions the nonlinear conditions will be considered when compute times are within or close to the real-time restriction for interactive applications.

A moving frame of reference is introduced in the time dependent equations to keep the ship fixed at the center of the numerical wave field such that

$$x = x_0 - Ut, \quad y = y_0, \quad z = z_0, \tag{7.4}$$

where (x_0, y_0, z_0) is origin in the fixed global coordinate system and U is the ship velocity in the x-direction. The linear kinematic and dynamic free surface boundary conditions are likewise modified to reflect the moving frame of

reference,

$$\partial_t \eta - U\partial_x \eta - w = 0, \quad z = 0, \tag{7.5a}$$

$$\partial_t \phi - U\partial_x \phi + g\eta + \frac{p}{\rho} = 0, \quad z = 0, \tag{7.5b}$$

where the new quantities ρ are the water density and p is the free surface pressure. The velocity vector $\mathbf{u} \equiv (u, v, w)$ is defined in the forward, transverse, and vertical directions, respectively. We consider again no-flux boundary conditions on the seabed and on the surface of the rigid bodies,

$$\mathbf{n} \cdot \nabla \phi = 0, \qquad z = -h \tag{7.6a}$$

$$\mathbf{n} \cdot \nabla \phi = \mathbf{n} \cdot \mathbf{u}_B, \qquad (x, y, z) \in S_B \tag{7.6b}$$

where $\mathbf{n} \equiv (n_x, n_y, n_z)^T$ is the normal to the boundary surface S_B, and $\mathbf{u}_B = (U, 0, 0)^T$.

A pressure distribution and a double body linearization is utilized to capture the main flow characteristics as a steady state solution [Rav10], that can be pre-computed and scaled linearly with respect to the reference velocity U. This rather simple approach was selected to minimize its influence on the overall efficiency. For further details on this hydrodynamic ship model we refer the reader to [LGB+13].

In addition to the hydrodynamic ship model, the GPUlab library was extended with support for handling multiple captive objects in the computational domain. A captive body was implemented, derived as a special instance of a floating body, with the main objective of controlling physical position and velocity as a function of time. A brief example of creating and adding a captive body (e.g., a ship) to the simulation is illustrated in Listing 7.1. The brief example demonstrates that once the underlying implementations are complete, the assembly phase can be implemented generically using an object oriented approach.

```
1  using namespace gpulab;
2
3  // Create body and set shape
4  captive_body<double> ship;
5  ship.set_shape(/* set ship hull shape */);
6
7  // Assign a pair of time and position
8  ship.get_time_position().insert( /* time, position */ );
9
10 captive_scene<captive_body<double> > scene;
11 scene.add(ship);
12
13 // Apply pressures on eta at time t=1s
14 scene.update(1.0,eta);
```

Listing 7.1: Constructing a ship object and assigning captive data and a shape. The ship is added to the scene manager and pressure is applied to the free surface elevation according to the hull shape and position at time $t = 1s$.

Both the ship hull and the locks are approximated with a pressure distribution in the dynamic free surface condition (7.5b). The pressure is approximated with a quasistatic assumption such that the pressure on is given by

$$p = -\rho(-U\partial_x\phi + \frac{1}{2}\mathbf{u}\cdot\mathbf{u} + g\eta_0), \tag{7.7}$$

where η_o is the draft from either the ship hull of the lock. The pressure contribution acting on the free surface allows a convex representation of the ship hull and locks. Therefore the bulbous bow was removed from the original container carrier mesh to be represented by a single valued function in the horizontal coordinates. Though this kind of approximation based on pressure contributions has shown to be applicable for ships sailing at low Froude numbers, it turns out not to be a good approximation to the locks [Rav10]. The vertical lock walls are represented via a pressure contributions that pushes the surface almost down to the seabed. Large gradients in the free surface elevation and potential will therefore occur regardless of the numerical resolution.

A numerical experiment was performed according to the guidelines in [VDM12], with a 12,000 TEU container carrier entering the lock at very low Froude numbers. The numerical resolution is $N_x \times N_y \times N_z = 513 \times 193 \times 17$, all variables are normalized with the ship length $Lpp = 348m$, such that the domain size is $L_x = 5$, $L_y = 1$, and $L_z = 0.048$. The hydrodynamic ship model results in a reasonable wave field generated by the carrier in the entrance part of the lock, illustrated in Figure 7.3. The steep gradients that are present at the interface between the locks and the free surface generate spurious waves as the carrier enters the lock. These artificial waves dominate the relatively small waves generated by the ship, as clearly visible in Figure 7.4. Though filtering methods could be applied to remove some of the spurious waves in the vicinity of the locks, it would unintentionally also affect the waves generated by the ship when the ship approaches the lock walls. The present hydrodynamic ship model therefore proved not to be suitable for representation of near-vertical geometries such as the lock in a moving frame of reference. Since the present model is to be implemented in a full mission marine simulator where such geometries are likely to exist, alternative methods have been investigated.

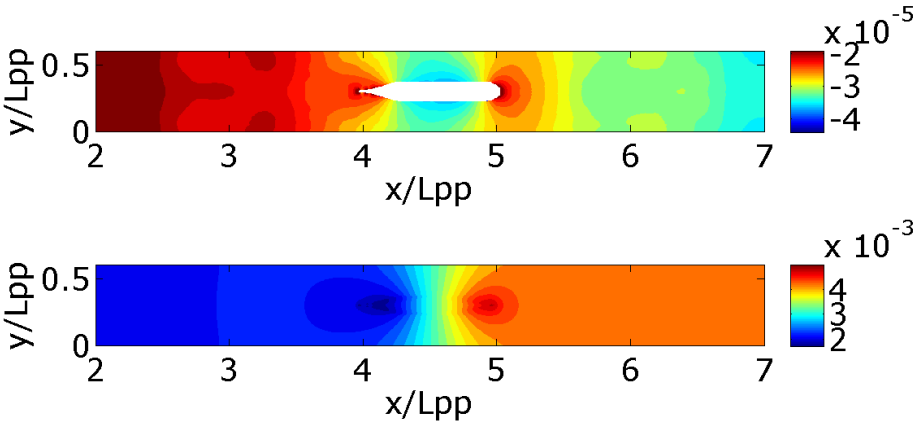


Figure 7.3: Free surface elevation and potential amplitudes before the ship enterns the lock, shortly after the startup phase. Numbers are normaliced by the ship length $Lpp = 348m$.

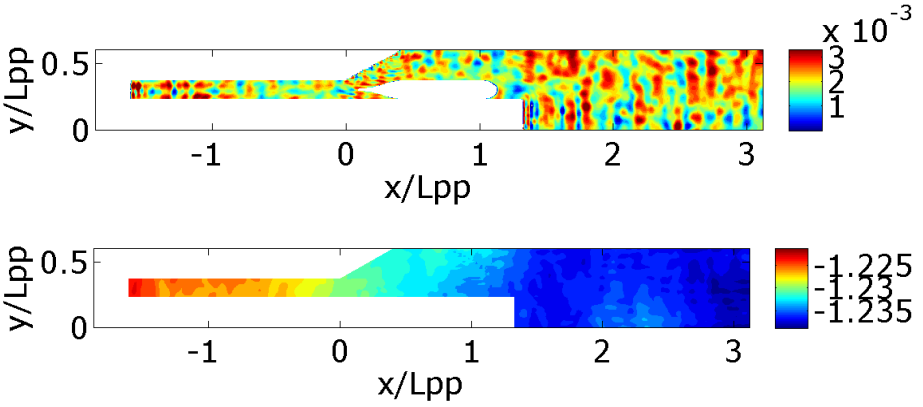


Figure 7.4: Free surface elevation and potential amplitudes when the ship enters the lock. Numbers are normaliced by the ship length $Lpp = 348m$.

7.3 Ship-wave interaction based immersed boundaries

The previous hydrodynamic ship model based on pressure distributions is not applicable for scenery that we want to consider. The lock entrance example clearly demonstrated weaknesses in the numerical modeling of deep and dis-

continuous geometries. Also the single value representation of ship hulls is a simplification that puts significant restrictions on the application range.

In the search for alternatives we explored a technique based on immersed boundary conditions, because these methods potentially have a flexible and accurate representation of geometries based on computational inexpensive techniques. The idea for an immersed boundary was originally introduced by Peskin, used for blood flow simulations in the heart using the Navier-Stokes equations [Pes02]. It has also been applied in the context of turbulent modeling of floating bodies interaction with water waves [YSW+07, YS09]. It is however, to the authors knowledge, the first time with this joint work, that it is applied to with the purpose of real-time simulation of ship generated water waves.

The principle of the immersed boundary method is to represent geometries via finite difference or finite volume boundary approximations directly at the boundary of the geometry. Immersed boundaries are therefore treated much like the original outer domain boundaries. For the representation of ship hulls we impose the inhomogeneous Neumann condition (7.6b) for the solution of the Laplace equation, by introducing fictitious ghost points inside the ship hull and then approximating the boundary conditions with stencils that take into account these ghost points. This is an attractive approach because the original fast GPU-based implementation of the Laplace operator can be re-used with small modification, both preserving performance and improving developer productivity.

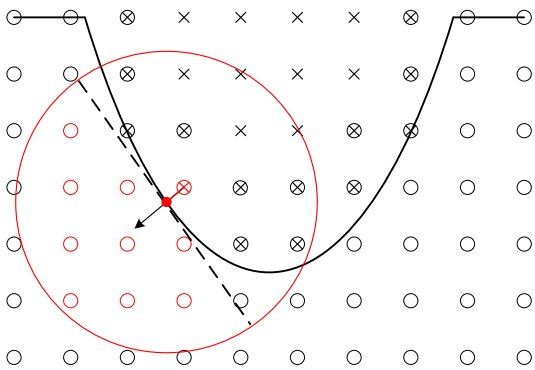


Figure 7.5: Discrete representation of a ship hull using immeresed boundaries. ○ is fluid grid points, ⊗ is body ghost points, ● is a body point, × is inactive interior body points.

One challenge for the immersed boundary method lies in the setup phase, where we need to properly identify the internal ghost points and compute the corre-

sponding finite difference coefficients satisfying (7.6b). A pre-processing phase is introduced where ghost points inside the body, which belong to a finite difference stencil, are identified and the corresponding body points are found via normal projections to the body surface, see Figure 7.5. The stencil coefficients are pre-computed via Taylor expansions from the body point to each fluid point at the outer tangential plane within a given Euclidean distance plus the ghost point, as illustrated in red. Taylor expansions up to second order are used to reduce the size of the resulting system of equations, but can be generalized to higher orders. The Taylor expansion from the body point to each of the fluid points in three spatial dimensions is

$$f(x_1+\Delta_1, x_2+\Delta_2, x_3+\Delta_3) \approx \sum_{i=0}^{2}\sum_{j=0}^{2}\sum_{k=0}^{2} \frac{\Delta_1^i \Delta_2^j \Delta_3^k}{i!j!k!} f^{(i,j,k)}(x_1, x_2, x_3).$$

The expansions with respect to each point can be arranged in an overdetermined linear system of equations (assuming there is more fluid points than expansion terms),

$$\mathcal{A}\mathbf{f} = \mathbf{b}, \quad \mathcal{A} \in \mathbb{R}^{\alpha\times\beta}, \mathbf{f} \in \mathbb{R}^{\beta}, \mathbf{b} \in \mathbb{R}^{\alpha}, \tag{7.8}$$

where $\mathcal{A}$ is assembled from the Taylor expansion coefficients, $\mathbf{f}$ represents each derivative $f^{(i,j,k)}$ at the body point, and $\mathbf{b}$ holds the values of each fluid and ghost points. β is equal to the number of expansion terms up to second order (27 in three dimensions) and α is the number of points considered. $\mathcal{A}$ is assembled using a coordinate system with a normal basis spanned by the tangential plane and the body normal. The first derivative, normal to the body surface, is then directly represented in one entry of $\mathbf{f}$. The system can be solved with a weighted least squares method,

$$\mathcal{A}^T\mathcal{W}\mathcal{A}\mathbf{f} = \mathcal{A}^T\mathcal{W}\mathbf{b}, \quad W \in \mathbb{R}^{\beta\times\alpha}, \tag{7.9}$$

where $\mathcal{W}$ is a diagonal matrix containing the weights. The coefficients corresponding to the surface normal derivative are now located in the second row of $[\mathcal{A}^T\mathcal{W}\mathcal{A}]^{-1}\mathcal{A}^T\mathcal{W}$. These coefficients can be pre-computed for each body ghost point and then later be used for satisfying the inhomogeneous Neumann boundary condition, by isolating the value for the ghost point using,

$$\sum_{p=1}^{\alpha} c_m \phi_{i(p),j(p),k(p)} \approx \mathbf{n}\cdot\mathbf{u_B}. \tag{7.10}$$

This calculation is performed in parallel on the GPU for each ghost point every time the boundary condition needs to be satisfied. Promising results have been obtained with the present immersed boundary method. Research is still active, and investigation for e.g., optimal search radii size and least square weights are ongoing. A parameter analysis of the weight matrix $\mathcal{W}$ is part of ongoing work.

The immersed boundary method has additional advantageous properties; The method is not restricted to either floating bodies or floor mounted and surface piercing geometries, which means that we can easily represent submerged bodies or complex ship hulls with bulbous bows. The method also seems promising with respect to performance, since most of the computational work is part of the pre-processing phase and because the original fast Laplace operator works with no changes. Only ghost points close to the body surface need to be updated, which is a relatively small task due to the volume to surface ratio.

7.4 Current status and future work

The project on real-time simulation of ship-ship interaction continues at FORCE Technology after the completion of the present work. The development of the numerically and computationally efficient hydrodynamic model is now being rigorously tested and validated. A snapshot of waves generated by a container carrier, based on the current immersed boundary implementation, is illustrated in Figure 7.6. The next step is to merge the computational model into the full mission marine simulator software, which will increase the requirements for reliability and robustness due to human interaction and complex simulation environments. Also a thorough performance optimization analysis will be carried out to pursue real-time performance at reasonable resolutions, to match the requirements of the marine simulator and to clearly set the standards within interactive ship-ship simulations. Performance results obtained with spatial domain decomposition in Section 4.4.6 indicated that there is no attainable speedup for the problem sizes considered for the hydrodynamic ship-wave model. However, a parameter study of the Parareal algorithm and the present model can possibly lead to speedups on heterogeneous systems where multiple GPUs are available. This will be investigated if single-GPU optimizations do not lead to sufficient speedup.

This collaboration is a good example of the possibilities that become available after development of a generic software framework. The GPUlab library, and the free surface water wave tool in particular, have been a perfect starting point for the collaboration on which the ship hydrodynamic research has been able to build directly on. The hardware abstraction provided by the library has enabled a lighter coding effort, that still enables execution in parallel on many-core heterogeneous computer systems.

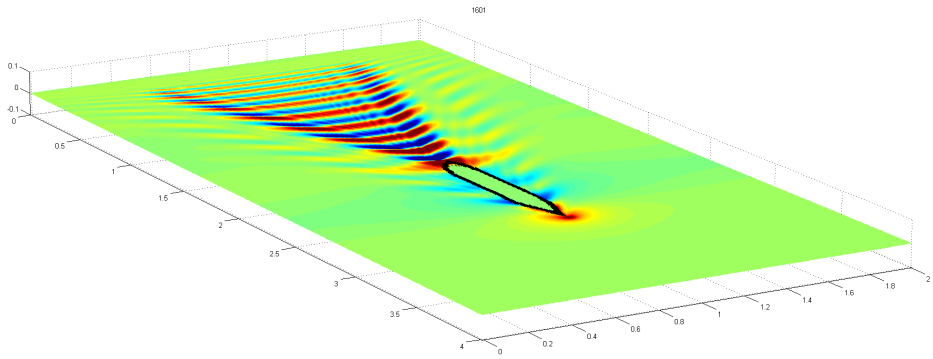


Figure 7.6: Free surface wave generation based on immersed boundary conditions. Waves are generated by a KCS container ship with Froude number $Fr = 0.2$.

7.5 Conclusion and outlook

Massively parallel heterogeneous systems continue to enter the consumer market, and there has been no identification that this trend will stop in years to come. Though these massively parallel many-core architectures have proven to be computationally efficient, with high performance throughput for a vast amount of applications[BG09, Mic09, GWS+09, GÏ0, EKMG11], they still pose significant challenges for software developers[EBW11], and they are still to become standard within the industry.

The present work can be considered as a reaction to the recent development of massively parallel architectures where we attempt to address some of the questions and challenges that follow from the new generation of hardware. One of the main challenges is that these heterogeneous systems require software vendors to adjust to new programming models and optimization strategies. We have presented our ideas for a generic GPU-based library for fast proto-typing of PDE solvers. A high-level interface with simple implementation concepts have been presented with the objective of enhancing developer productivity and to ensure performance portability and scalability.

Based on the proof-of-concept nonlinear free surface water wave model, we have presented details of a domain decomposition technique in three spatial dimensions for distributed parallel computations on both desktop and cluster platforms. The spatial domain decomposition technique preserves algorithmic efficiency and further improves performance of the single-block version of the solver, cf. recent studies [EKBL09, EKMG11, GEKM11]. If the performance speedups that we have reported for the single-block solver in Section 3.4 of up to 100 compared to the single-threaded CPU solver, are combined with either the results reported with spatial domain decomposition or Parareal, then we achieve speedups that significantly beat Moore's law. We mean by this that the performance improvements we have achieved within 3-4 years of development, cannot be explained only by Moore's law, but is a product of Moore's law and improved algorithmic design choices and efficient implementations.

The numerical model is implemented using the GPUlab library based on Nvidia's CUDA programming model and is executed on a recent generation of programmable GPUs. In performance tests, we have demonstrated good weak scalability in absolute as well as measures for relative speedups and efficiency in comparison to the single-GPU implementation. Scaling results based on the multi-block solver have detailed how the free surface model is capable of solving systems of equations with more than one billion degrees of freedom for the first time. These impressive numbers can be obtained with a reasonable sized cluster equipped with 16 or more GPUs. Even at these large-scale problems

we outlined that simulations are in fact possible within reasonable turn-around-times. Also, we highlight that multi-GPU implementations can be a means for further acceleration of run-times in comparison with single-GPU computations. A study of the multigrid coarsening strategy has led to the conclusion that few multigrid levels are sufficient for fast convergence of the Laplace problem. These results imply that a significant reduction in communication can be obtained to maximize performance throughput.

The library has already successfully been used for development of a fast tool intended for scientific applications within maritime engineering, cf. Chapter 7. We intend to further extend the library, as we explore new techniques, suitable for parallelization on heterogeneous systems, that fit the scope of our applications. The library and the free surface water wave solver, in their present stages, are ideal for further collaboration on PDE applications that may benefit from parallelization on heterogeneous systems.

The GPUlab library has been developed and designed exclusively by the author himself. A major effort has been put into the development of all the generic design and implementation of iterative solvers, time integrators, the matrix-free stencil operators etc.. Also, the implementation and validation of the free surface wave solver on heterogeneous hardware have taken a significant amount of time. This confirms our motivation for the project itself, that software development for massively parallel processors is very time consuming, and the need for well designed, portable, and efficient software libraries is key to future developments. In particular, the process of debugging parallel programs running on heterogeneous hardware is a troublesome procedure.

7.6 Future work

Our work has tried to address a number challenges that arise in heterogeneous computing and software design. However, along the way there have been a number of unanswered questions and new challenges that arise. We suggest that the following topics can be used for directing future research towards supplementing or improving our work.

Large-scale engineering case The combination of an efficient multi-block and boundary-fitted free surface wave solver opens up new opportunities for fully three-dimensional large-scale applications of practical interest in maritime engineering. With interconnected and boundary-fitted domains it would be possible to construct large and complex harbor facilities. In

future work we would like to demonstrate practical examples based on the present free surface solver.

Parareal fault resilience Parareal possesses several interesting features, fault resilience being a highly relevant feature of execution on massively parallel HPC clusters. A numerical experiment that first of all confirms this property along with a deeper analysis of how well and fast Parareal is able to fully recover from an erroneous process would be of interest. A study of how stable Parareal will be if multiple processors are lost may also be of interest.

Autotuning There is a number of kernel configuration parameters that are based on basic knowledge of good programming practice. However, these configurations are not guaranteed to be optimal on any heterogeneous system. An autotuning module that can be executed on specific heterogeneous systems can be used to reveal if there are alternative configurations that would result in improved performance. Such a module would be a valuable tool to maximize performance portable settings.

OpenCL In this work we have exclusively considered the CUDA for C programming model. Though OpenCL traditionally has not been able to offer the same degree of documentation, it is not designed specifically for GPU programming, but is designed to execute on heterogeneous system in general. An interesting analysis of CUDA vs. OpenCL or especially OpenCL on alternative systems, e.g., multi-core CPUs or the Intel MIC, would be of interest. The CUDA programming model has proven very attractive for development of Nvidia GPU-based applications. However, we believe that the restriction to Nvidia GPUs will eventually cause CUDA to play a minor role in future HPC environments.

Green computing Massively parallel processors are able to offer an effective performance per watt ratio, and therefore they are expected to play an important role in future HPC systems where energy consumption is a primary challenge. A better understanding of the power consumption of the GPU and the system as a whole in relation to performance throughput is a relevant concern and will be important for future hardware design.

Appendix A

The GPUlab library

The GPUlab library is a generic C++/CUDA/MPI library designed for fast prototyping of large-scale PDE solvers – without compromising performance.

The library hides CUDA-specific implementation details behind a generic interface that will allow developers to assemble PDE solvers at a much higher abstraction level. The library is designed to be efficient, portable and to enhance developer productivity. The library is component-based, and it provides a good starting point, from which custom designed components can be implemented, to assemble advanced PDE solvers. Thus, it is expected that the users are able to implement custom components that may contain CUDA specific implementations. The library will provide the basis for many of these custom components. The following sections will present a brief introduction to the programming guidelines and principles. Examples will demonstrate the use of specific components.

A.1 Programming guidelines

The GPUlab library is a header-only library, which means that everything is contained in header files. Therefore the library should not be pre-compiled and

there is no linking stage.

For the GPUlab library to work properly in multi-GPU settings, it needs to setup a private MPI communicator and assign each GPU to individual processes. Therefore a program should at least follow this template:

```
#include <gpulab/gpulab.h>

int main(int argc, char** argv)
{
    using namespace gpulab;

    // Initialize GPUlab
    if(gpulab::init(argc, argv))
    {
        // DO STUFF HERE
    }
    // Finalize GPUlab
    gpulab::finalize();
    return 0;
}
```

There is always one optional input parameter that can be passed to the program, it is the name of a configuration file. If not specified, the default value `config.ini` is used.

A.1.1 Templates

The library heavily depends on template arguments, which allow a flexible and user-controllable environment. We therefore recommend to use type definitions at the very beginning of the program to control the assembling of the PDE solvers. The program might start with:

```
// Basic type definitions
typedef float                                                   value_type;
typedef gpulab::vector<value_type,gpulab::device_memory>        vector_type;
typedef gpulab::vector<value_type,gpulab::host_memory>          host_vector_type;
typedef myLaplaceMatrix<device_vector_type>                     matrix_type;

typedef typename grid_type::property_type                       property_type;
typedef typename grid_type::dim_size_type                       dim_size_type;
typedef typename grid_type::dim_value_type                      dim_value_type;
```

This will allow you to control e.g., the working precision or the matrix implementation only one place in the code.

A.1.2 Dispatching

Due to the template-based design it is not directly possible to determine the type of objects, but sometimes it is convenient to treat host and device vectors differently. To do this one can create a overloaded function with dispatching:

```
typedef gpulab::vector<double,gpulab::device_memory>    d_vector_type;
typedef gpulab::vector<double,gpulab::device_memory>    h_vector_type;

int main(int argc, char** argv)
{
    if(gpulab::init(argc, argv))
    {
        d_vector_type x(10);    // Device vector
        h_vector_type y(10);    // Host vector

        foo(x);     // Will be dispatched to device code
        foo(y);     // Will be dispatched to host code
    }
    gpulab::finalize();
    return 0;
}

template<class V>
void foo(V &a)
{
    // Dispatch to the right function
    foo(a,typename V::memory_space());
}

template<class V>
void foo(V &a, gpulab::device_memory)
{
    // a is a device vector
}

template<class V>
void foo(V &a, gpulab::host_memory)
{
    // a is a host vector
}
```

A.1.3 Vectors and device pointers

The vector class is derived from the Thrust[BH11] vector objects and therefore offers the same container iterators and operators, e.g.:

```
typedef gpulab::vector<double,gpulab::device_memory>    vector_type;

vector_type a(20);                    // vector of size 20
thrust::sequence(a.begin(),a.end()); // Fill a with 1,2,...,19
```

The vector class is also a wrapper for a pointer that points to either host or device memory. If a device pointer is required to send to a kernel function, then

do:

```
typedef gpulab::vector<double,gpulab::device_memory>    vector_type;

vector_type a(10);          // vector of size 10
double* ptr = RAW_PTR(a);   // pointer to device memory
```

A.1.4 Configuration files

The GPUlab is initialized with a (possibly empty) configuration file, named `config.ini` that should be in the same directory as the executable. The configuration file is a simple text file. Every line in the file defines a key and a value. In order to retrieve information from the configuration file use:

```
double tol;
int N;
GPULAB_CONFIG_GET("N",&N,100);           // Get N from config, default value
    = 100
GPULAB_CONFIG_GET("tolerance",&tol,0.0); // Get tolerance from config,
    default value = 0
```

If the key is not found in the configuration database, the default value will be used. The GPUlab library will automatically try to convert the value onto the same type as provided.

it is also possible to insert entries into the configuration database. They only exist during the lifetime if the program and are not stored in the text file.

```
double tol = 0.1;
int N = 30;
GPULAB_CONFIG_SET("N",N);           // Set N to 30
GPULAB_CONFIG_SET("tolerance",tol); // Set tolerance to 0.1
```

A.1.5 Logging

The GPUlab library will automatically log some information to the screen during a run. The level at which information will be displayed can be controlled via the configuration file and can be any of:

```
log_level DEBUG
log_level INFO
log_level WARNING
log_level ERROR
```

From anywhere in your code you can log information by doing:

```
int N = 100;
GPULAB_LOG_DBG("This is debugging\n");
GPULAB_LOG_INF("For your information, N = %i \n", N);
GPULAB_LOG_WRN("This is a warning\n");
GPULAB_LOG_ERR("This is an error\n");
```

A.1.6 Input/Output

There is build in routines for reading and writing vectors to binary format.

```
#include <gpulab/io/print.h>
#include <gpulab/io/read.h>

using namespace gpulab;
vector<double,device_memory> a(100);
vector<double,device_memory> b;
io::print(a,io::TO_BINARY_FILE,1,"a.bin");  // Print to file a.bin
io::read("b.bin",io::BINARY,1);             // Read from file b.bin
```

A.1.7 Grids

The grid class is an extension to the vector class and it holds extra dimensional information. Here is an example of how to to create a two-dimensional grid with Dirichlet boundary conditions in the x-direction and Neumann boundary conditions in the y-direction:

```
#include <gpulab/grid.h>
typedef gpulab::grid<double,gpulab::device_memory>  grid_type;
typedef typename grid_type::property_type           property_type;
typedef typename grid_type::dim_size_type           dim_size_type;
typedef typename grid_type::dim_value_type          dim_value_type;

dim_size_type dim(100,50);           // Grid size 100 x 50
dim_value_type p0 ( 0, 0);           // Physical x0, y0
dim_value_type p1 ( 1, 1);           // Physical x1, y1
dim_size_type  g0 ( 0, 2);           // Ghost layers x0, y0
dim_size_type  g1 ( 0, 2);           // Ghost layers x1, y1
dim_size_type  o0 ( 0, 0);           // Offset x0, y0
dim_size_type  o1 ( 0, 0);           // Offset x1, y1
dim_size_type  bc0( BC_DIR, BC_NEU); // Boundary conditions
dim_size_type  bc1( BC_DIR, BC_NEU); // Boundary conditions

property_type props2d(dim,p0,p1,g0,g1,o0,o1,bc0,bc1);

grid_type U(props2d);
grid_type* W = U.duplicate();
```

A.1.8 Matlab supporting file formats

To support post- and pre-processing in Matlab we have also created functions to read, write and reshape vectors and grids in binary files. Simply use:

```
v1 = 1:100;
print2gpu('v.bin',v1,'precision','double')
v2 = readgpu('v.bin','precision','double')
```

A.2 Configuring a free surface water wave application

A.2.1 Configuration file

The configuration of a free surface application can be controlled with the GPUlab input configuration file. Most of the configuration parameters have default values, but in order to control the wave characteristics one can specify:

```
# WAVE CHARACTERISTICS
L 1.0
Lx 4.0
Ly 1.0
h 1.0
k 6.283185
H 0.042436
c 1.261316
T 0.792823
percent 30.0
linear 0
periodic 0
```

For the discretization settings the following parameters can be specified:

```
# DISCRETIZATION PARAMETERS
alpha 3
Nx 257
Ny 129
Nz 9
Cr 0.6
tend 20
```

And finally some of the configuration parameters that controls the multigrid preconditioned defect correction solver:

```
# LAPLACE SOLVER PARAMETERS
iter 20
v1 2
```

```
v2 2
vc 4
K 4
rtol 0.0001
atol 0.00001
mgcycle V
```

A.2.2 The Matlab GUI

Manually configuring the wave characteristics can be troublesome as they have to match each other. We have therefore creates a simple Matlab GUI that will help create the configuration files, see Figure A.1

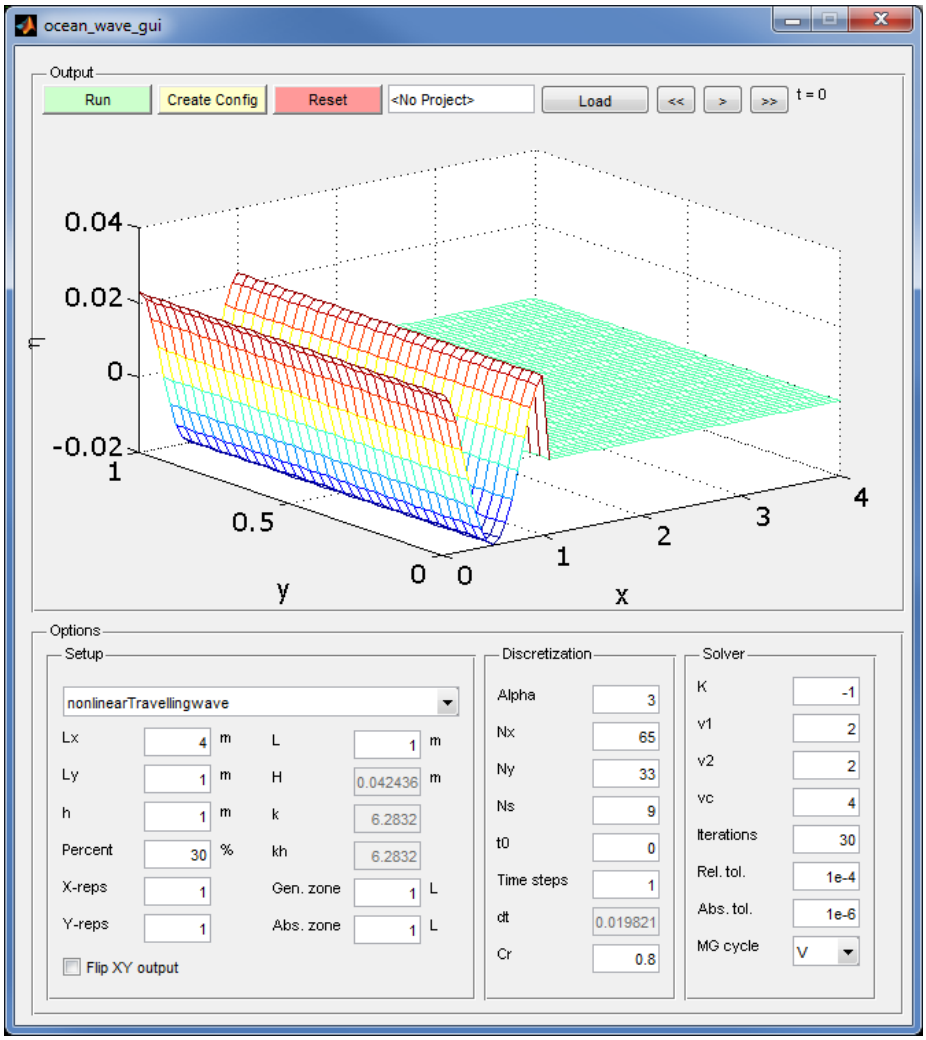


Figure A.1: The Matlab GUI for creating configuration files to the free surface water wave solver.

Bibliography

[ABC+06] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[ABD+13] A. M. Aji, P. Balaji, J. Dinan, W.-C. Feng, and R. Thakur. Synchronization and ordering semantics in hybrid MPI+GPU programming. In *3rd International Workshop on Accelerators and Hybrid Exascale Systems (AsHES)*, 2013.

[ALB98] E. Acklam, H. P. Langtangen, and A. M. Bruaset. Parallelization of explicit finite difference schemes via domain decomposition, 1998.

[AMW84] M. Abott, A. McCowan, and I. Warren. Accuracy of short-wave numerical models. *ASCE Journal of Hydraulic Engineering*, 110(10):1287–1301, 1984.

[APS78] M. Abott, H. Petersens, and O. Skovgaard. On the numerical modelling of short waves in shallow water. *Journal of Hydraulic Research*, 16(3):173–203, 1978.

[Aub11] E. Aubanel. Scheduling of tasks in the parareal algorithm. *Parallel Computing*, 37:172–182, 2011.

[Bai91] D. H. Bailey. Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomputing Review*, pages 54–55, 1991.

[Bai92] D. H. Bailey. Misleading performance reporting in the supercomputing field. Technical report, Numerical Aerodynamic Simulation (NAS) Systems Division, NASA Ames Research, 1992.

[Bai09] D. H. Bailey. Misleading performance claims in parallel computations. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 528–533. ACM, 2009.

[BBB+11] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc developers manual. Technical report, Argonne National Laboratory, 2011.

[BBB+13] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.4, Argonne National Laboratory, 2013.

[BBC+94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, 1994.

[BBD+09] M. Baboulin, A. Buttarib, J. Dongarra, J. Kurzak, J. Langouc, J. Langou, P. Luszczek, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180:2526–2533, 2009.

[BBM+02] L. Baffico, S. Bernard, Y. Maday, G. Turinici, and G. Zérah. Parallel in time molecular dynamics simulations. *Physical Review E.*, 66(057701), 2002.

[BFH+04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[BG09] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. ACM, 2009.

[BH11] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. *in GPU Computing Gems, Jade Edition, Edited by Wen-mei W. Hwu*, 2:359–371, 2011.

[BHM00] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial*. Society of Industrial and Applied Mathematics, SIAM, 2000.

[BMK+10] D. L. Brown, P. Messina, D. Keyes, J. M. , R. Lucas, J. Shalf, P. Beckman, R. B. , A. Geist, J. Vetter, B. L. Chamberlain, E. Lusk, J. Bell, M. S. Shephard, M. Anitescu, D. Estep, D. Estep, A. Pinar, and M. A. Heroux. Scientific grand challenges, crosscutting technologies for computing at the exascale. Technical report, U.S. Department of Energy, 2010.

[BN04] S. Beji and K. Nadaoka. Fully dispersive nonlinear water wave model in curvilinear coordinates. *Journal of Computational Physics*, 198(2):645–658, 2004.

[Bra77] A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, 31(138):333–390, 1977.

[Bur95] K. Burrage. *Parallel and Sequential Methods for Ordinary Differential Equations*. Monographs on Numerical Analysis. Clarendon Press, 1995.

[BZ89] A. Bellen and M. Zennaro. Parallel algorithms for initial-value problems for difference and differential equations. *Journal of Computational and Applied Mathematics*, 25(3):341–350, 1989.

[BZ07] H. B. Bingham and H. Zhang. On the accuracy of finite-difference solutions for nonlinear water waves. *Journal of Engineering Mathematics*, 58:211–228, 2007.

[CM93] A. Chorin and J. Marsden. *A Mathematical Introduction to Fluid Mechanics*. Texts in Applied Mathematics. Springer, 1993.

[Coo12] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Applications of GPU Computing Series. Elsevier Science, 2012.

[CPL05] X. Cai, G. Pedersen, and H. Langtangen. A parallel multi-subdomain strategy for solving boussinesq water wave equations. *Advances in Water Resources*, 28:215–233, 2005.

[DB06] F. Dias and T. J. Bridges. The numerical computation of freely propagating time-dependent irrotational water waves. *Fluid Dynamics Research*, 38(12):803 – 830, 2006. Free-Surface and Interfacial Waves.

[DBEKF10] G. Ducrozet, H. Bingham, A. Engsig-Karup, and P. Ferrant. High-order finite difference solution for 3d nonlinear wave-structure interaction. *Journal of Hydrodynamics*, 22(5):225–230, 2010.

[DBM+11] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. Van Der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

[DKM94] R. A. Dalrymple, J. T. Kirby, and P. Martin. Spectral methods for forward-propagating water waves in conformally-mapped channels. *Applied Ocean Research*, 16(5):249 – 266, 1994.

[DMV+08] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE Press, 2008.

[EBW11] D. Eschweiler, D. Becker, and F. Wolf. Patterns of inefficient performance behavior in GPU applications. In *Proceedings of the 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, PDP '11, pages 262–266. IEEE Computer Society, 2011.

[EGH03] R. Eymard, T. Gallouët, and R. Herbin. *Handbook of Numerical Analysis*, chapter Finite Volume Methods, pages 713–1020. 2003.

[EK06] A. P. Engsig-Karup. *Unstructured Nodal DG-FEM Solution of High-Order Boussinesq-type Equations*. Technical University of Denmark, Department of Mechanical Engineering, 2006.

[EK14] A. P. Engsig-Karup. Analysis of efficient preconditioned defect correction methods for nonlinear water waves. *To appear: International Journal for Numerical Methods in Fluids*, 2014.

[EKBL09] A. Engsig-Karup, H. Bingham, and O. Lindberg. An efficient flexible-order model for 3D nonlinear water waves. *Journal of Computational Physics*, 228:2100–2118, 2009.

[EKGNL13] A. P. Engsig-Karup, S. L. Glimberg, A. S. Nielsen, and O. Lindberg. *Designing Scientific Applications on GPUs*, chapter Fast hydrodynamics on heterogenous many-core hardware. Chapman & Hall/CRC Numerical Analysis and Scientific Computing Series, 2013.

[EKMG11] A. P. Engsig-Karup, M. G. Madsen, and S. L. Glimberg. A massively parallel GPU-accelerated model for analysis of fully nonlinear free surface waves. *International Journal for Numerical Methods in Fluids*, 70(1):20–36, 2011.

[Far11] R. Farber. *CUDA Application Design and Development.* Applications of GPU computing. Morgan Kaufmann, 2011.

[FC] W. Feng and K. W. Cameron. Top 500 green supercomputer sites, www.green500.org.

[FP96] J. Ferziger and M. Perić. *Computational methods for fluid dynamics.* Numerical methods: Research and development. Springer-Verlag, 1996.

[FR82] J. D. Fenton and M. M. Rienecker. A fourier method for solving nonlinear water-wave problems: application to solitary-wave interactions. *Journal of Fluid Mechanics*, 118:411–443, 1982.

[GÏ0] D. Göddeke. *Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU Clusters.* PhD thesis, Der Fakultät für Mathematik der Technischen Universität Dortmund, 2010.

[GEKM11] S. L. Glimberg, A. P. Engsig-Karup, and M. G. Madsen. A fast GPU-accelerated mixed-precision strategy for fully nonlinear water wave computations. *Proceedings of European Numerical Mathematics and Advanced Applications (ENUMATH)*, pages 645–652, 2011.

[GEKND13] S. L. Glimberg, A. P. Engsig-Karup, A. S. Nielsen, and B. Dammann. *Designing Scientific Applications on GPUs*, chapter Development of software components for heterogeneous many-core architectures. Chapman & Hall/CRC Numerical Analysis and Scientific Computing Series, 2013.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.

[GLS99] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. MIT Press, 1999.

[GLT99] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, 1999.

[Gmb] P. GmbH. Top 500 supercomputer sites, www.top500.org.

[GPL04] S. Glimsdal, G. Pedersen, and H. P. Langtangen. An investigation of domain decomposition methods for one-dimensional dispersive long wave equations. *Advances in Water Resources*, 27(11):1111–1133, 2004.

[GS10] D. Göddeke and R. Strzodka. Cyclic reduction tridiagonal solvers on GPUs applied to mixed precision multigrid. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints):1–13, 2010.

[GV07] M. Gander and S. Vandewalle. Analysis of the parareal time-parallel time-integration method. *SIAM Journal of Scientific Computing*, 29(2):556–578, 2007.

[GWS+09] 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. *International Journal of Computational Science and Engineering*, 2009.

[HC05] X. Hu and C. T. Chan. Refraction of water waves by periodic cylinder arrays. *Physical Review Letters*, 95:154501, 2005.

[HS11] T. Hoefler and M. Snir. Writing parallel libraries with MPI - common practice, issues, and extensions. In Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6960 of *Lecture Notes in Computer Science*, pages 345–355. Springer Berlin / Heidelberg, 2011.

[Hwu11] W. Hwu. *GPU Computing Gems Jade Edition*. Applications of GPU Computing Series. Elsevier Science, 2011.

[HX96] J. Häuser and Y. Xia. *Modern Introduction to Grid Generation*. Department of Parallel Computing Center of Logistics and Expert Systems, 1996.

[IN90] A. Iserles and S. P. Norsett. On the Theory of Parallel Runge–Kutta Methods. *IMA Journal of Numerical Analysis*, 10(4):463–488, 1990.

[ITRon] ITRS. International Technology Roadmap for Semiconductors, Assembly and Packaging, 2011 Edition.

[JDB+12] J. Jenkins, J. Dinan, P. Balaji, N. F. Samatova, and R. Thakur. Enabling fast, noncontiguous GPU data movement in hybrid MPI+GPU environments. *2012 IEEE International Conference on Cluster Computing*, 0:468–476, 2012.

[JS05] C. Jiang and M. Snir. Automatic tuning matrix multiplication performance on graphics hardware. In *IEEE PACT*, pages 185–196. IEEE Computer Society, 2005.

[KDW+06] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, MSPC '06, pages 51–60. ACM, 2006.

[Kea11] D. Keyes and V. T. et al. Task force on software for science and engineering. Technical report, National Science Foundation. Advisory Committee for CyberInfrastructure, 2011.

[Kel95] C. Kelley. *Iterative Methods for Linear and Nonlinear Equations.* Society for Industrial and Applied Mathematics, 1995.

[Key11] D. E. Keyes. Exaflop/s: The why and the how. *Journal of Comptes Rendus Mecanique*, 339:70–77, 2011.

[KM92] T. Korson and J. D. McGregor. Technical criteria for the specification and evaluation of object-oriented libraries. *Journal of Software Engineering*, 7(2):85–94, 1992.

[KmWH10] D. B. Kirk and W. mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series).* Morgan Kaufmann, 1st edition, 2010.

[Kop09] D. Kopriva. *Implementing Spectral Methods for Partial Differential Equations: Algorithms for Scientists and Engineers.* Mathematics and Statistics. Springer Science+Business Media B.V., 2009.

[Kri90] D. Kriebel. Nonlinear wave interaction with a vertical circular cylinder. part i: Diffraction theory. *Ocean Engineering*, 17(4):345 – 377, 1990.

[Kri92] D. Kriebel. Nonlinear wave interaction with a vertical circular cylinder. part ii: Wave run-up. *Ocean Engineering*, 19(1):75 – 99, 1992.

[LD83] J. Larsen and H. Dancy. Open boundaries in short wave simulations - a new approach. *Coastal Engineering*, 7:285–297, 1983.

[LeV07] R. J. LeVeque. *Finite difference methods for ordinary and partial differential equations - steady-state and time-dependent problems.* SIAM, 2007.

[LF97] B. Li and C. A. Fleming. A three dimensional multigrid model for fully nonlinear water waves. *CE*, 30:235–258, 1997.

[LF01] B. Li and C. A. Fleming. Three-dimensional model of navier-stokes equations for water waves. *Waterway, Port, Coastal, and Ocean Engineering*, pages 16–25, 2001.

[LGB+12] O. Lindberg, S. L. Glimberg, H. B. Bingham, A. P. Engsig-Karup, and P. J. Schjeldahl. Towards real time simulation of ship-ship interaction - part ii: double body flow linearization and GPU implementation. In *Proceedings of The 28th International Workshop on Water Waves and Floating Bodies (IWWWFB)*, 2012.

[LGB+13] O. Lindberg, S. L. Glimberg, H. B. Bingham, A. P. Engsig-Karup, and P. J. Schjeldahl. Real-time simulation of ship-structure and ship-ship interaction. In *3rd International Conference on Ship Manoeuvring in Shallow and Confined Water*, 2013.

[Lin08] P. Lin. *Numerical Modeling of Water Waves.* Taylor & Francis, 2008.

[Lis99] V. D. Liseikin. *Grid Generation Methods.* Scientific Computation. Springer-Verlag, 1999.

[LMT01] J.-L. Lions, Y. Maday, and G. Turinici. Résolution d'edp par un schéma en temps pararéel. *C.R. Acad Sci. Paris Sér. I math*, 332:661–668, 2001.

[LZ01] Y. S. Li and J. M. Zhan. Boussinesq-type model with boundary-fitted coordinate system. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, 127(3):152–160, 2001.

[Mad08] Y. Maday. The parareal in time algorithm. Technical Report R08030, Universite Pierré et Marie Curie, 2008.

[MF54] R. MacCamy and R. Fuchs. *Wave Forces on Piles: A Diffraction Theory.* Technical memorandum. U.S. Beach Erosion Board, 1954.

[Mic09] P. Micikevicius. 3d finite difference computation on GPUs using CUDA. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. ACM, 2009.

[Moo65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[MSK10] V. Minden, B. Smith, and M. Knepley. Preliminary implementation of petsc using GPUs. *Proceedings of the 2010 International Workshop of GPU Solutions to Multiscale Problems in Science and Engineering*, 2010.

[Nie12] A. S. Nielsen. Feasibility study of the parareal algorithm. Master thesis, Technical University of Denmark, Department of Informatics and Mathematical Modeling, 2012.

[Nvi12a] Nvidia. Nvidia GPUDirect technology and cluster computing. online matrial. 2012.

[Nvi12b] Nvidia. CUDA C Best Practices Guide. online material, 2012.

[Nvi13] Nvidia. CUDA C Programming Guide. online material. 2013.

[Per67] D. Peregrine. Long waves on a beach. *Journal of Fluid Mechanics*, 27(4):815–827, 1967.

[Pes02] C. S. Peskin. The immersed boundary method. *Acta Numerica*, 11:479–517, 2002.

[PK87] J. Pos and F. Kilner. Breakwater gap wave diffraction: an experimental and numerical study. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, 113(1):1–21, 1987.

[Rav10] H. C. Raven. Validation of an approach to analyse and understand ship wave making. 15:331–344, 2010.

[Saa03] Y. Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, 2003.

[SBG96] B. F. Smith, P. E. Bjørstad, and W. D. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, 1996.

[SDB94] A. Skjellum, N. E. Doss, and P. V. Bangaloret. Writing libraries in MPI. Technical report, Department of Computer Science and NSF Engineering Research Center for Computational Fiels Simulation. Mississippi State University, 1994.

[SDK+01] F. Shi, R. A. Dalrymple, J. T. Kirby, Q. Chen, and A. Kennedy. A fully nonlinear boussinesq model in generalized curvilinear coordinates. *Coastal Engineering*, 42:337–358, 2001.

[SJ76] I. A. Svendsen and I. G. Jonsson. *Hydrodynamics of coastal regions*, volume 3. Den private inginoerfond, Technical University of Denmark, 1976.

[SK10] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[SS95] F. Shi and W. Sun. A variable boundary model of storm surge flooding in generalized curvilinear grids. *International Journal for Numerical Methods in Fluids*, 21(8):641–651, 1995.

[TOS+01] U. Trottenberg, C. W. Oosterlee, A. Schuller, contributions by A. Brandt, P. Oswald, and K. Stuben. *Multigrid*. Academic Press, 2001.

[VDM12] M. Vantorre, G. Delefortrie, and F. Mostaert. Behaviour of ships approaching and leaving locks: Open model test data for validation purposes. Technical report, WL2012R815_08e. Flanders Hydraulics Research and Ghent University - Division of Maritime Technology: Antwerp, Belgium, 2012.

[VJ02] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 1st edition, 2002.

[Wil81] C. J. Willmott. On the validation of models. *Physical Geography*, 2(2):184–194, 1981.

[WoEU71] R. W. Whalin, U. S. A. C. of Engineers, and W. E. S. (U.S.). *The Limit of Applicability of Linear Wave Refraction Theory in a Convergence Zone*. Research report. Waterways Experiment Station, 1971.

[WPL+11a] H. Wang, S. Potluri, M. Luo, A. K. Singh, X. Ouyang, S. Sur, and D. K. Panda. Optimized non-contiguous MPI datatype communication for GPU clusters: Design, implementation and evaluation with MVAPICH2. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, pages 308–316. IEEE Computer Society, 2011.

[WPL+11b] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda. MVAPICH2-GPU: optimized GPU to GPU communication for infiniband clusters. *Computer Science - Research and Development*, 26(3-4):257–266, 2011.

[wZsZ10] H. wei Zhou and H. sheng Zhang. A numerical model of nonlinear wave propagation in curvilinear coordinates. *China Ocean Engineering*, 24(4):597–610, 2010.

[Yeu82] R. W. Yeung. Numerical methods in free-surface flows. In *Annual review of fluid mechanics, Vol. 14*, pages 395–442. Annual Reviews, 1982.

[YS09] J. Yang and F. Stern. Sharp interface immersed-boundary/level-set method for wave-body interactions. *Journal of Computational Physics*, 228:6590–6616, 2009.

[YSW+07] J. Yang, N. Sakamoto, Z. Wang, P. Carrica, and F. Stern. Two phase level-set/immersed-boundary cartesian grid method for ship hydrodynamics. *Ninth International Conference on Numerical Ship Hydrodynamics*, 2007.

[zFlZbLwY12] K. zhao FANG, Z. li ZOU, Z. bo LIU, and J. wei YIN. Boussinesq modelling of nearshore waves under body fitted coordinate. *Journal of Hydrodynamics, Ser. B*, 24(2):235 – 243, 2012.

[ZZY05] H. Zhang, L. Zhu, and Y. You. A numerical model for wave propagation in curvilinear coordinates. *Coastal Engineering*, 52:513–533, 2005.