Czech Technical University in Prague
Faculty of Nuclear Sciences and Physical Engineering

Dissertation

Data Structures and Parallel Algorithms for Numerical Solvers in Computational Fluid Dynamics

Prague 2023 Jakub Klinkovský

This thesis is submitted to the Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague, in partial fulfilment of the requirements for the degree of Doctor of Philosophy (Ph.D.) in Mathematical Engineering.

Bibliografický záznam

Název práce	Datové struktury a paralelní algoritmy pro numerické řešiče ve výpočetní dynamice tekutin
Autor	Ing. Jakub Klinkovský
Školitel	doc. Ing. Tomáš Oberhuber, Ph.D.
Školitel specialista	doc. Ing. Radek Fučík, Ph.D.
Pracoviště	České vysoké učení technické v Praze, Fakulta jaderná a fyzikálně inženýrská, Katedra matematiky
Studijní program	Aplikace přírodních věd
Studijní obor	Matematické inženýrství
Akademický rok	2022–2023
Počet stran	184
Klíčová slova	Paralelní výpočty, distribuované výpočty, nestrukturovaná síť, hybridní metoda smíšených konečných prvků, mřížková Boltzmannova metoda, validace

Bibliographic Entry

Dissertation title	Data Structures and Parallel Algorithms for Numerical Solvers in Computational Fluid Dynamics
Author	Ing. Jakub Klinkovský
Supervisor	doc. Ing. Tomáš Oberhuber, Ph.D.
Co-supervisor	doc. Ing. Radek Fučík, Ph.D.
Affiliation	Czech Technical University in Prague, Faculty of Nuclear Sciences and Physical Engineering, Department of Mathematics
Degree programme	Application of Natural Sciences
Field of study	Mathematical Engineering
Academic year	2022–2023
Number of pages	184
Keywords	Parallel computing, distributed computing, unstructured mesh, mixed-hybrid finite element method, lattice Boltzmann method, validation

Abstrakt

Práce uvádí výsledky multidisciplinárního výzkumu, které kombinují znalosti z oblastí matematického modelování, vědeckých výpočtů a experimentální fyziky.

První část práce obsahuje přehled technik pro programování moderních paralelních výpočetních systémů a představuje dvě efektivní a konfigurovatelné datové struktury pro reprezentaci dat pomocí vícerozměrných polí, respektive konformních nestrukturovaných sítí. Obě datové struktury jsou implementovány v knihovně TNL a mají přímou podporu pro výpočty na grafických akcelerátorech a také pro distribuované výpočty s MPI.

Druhá část práce se věnuje numerickým metodám ve výpočetní dynamice tekutin a jejim aplikacím. Nejprve je popsán obecný řešič založený na hybridní metodě smíšených konečných prvků (MHFEM), který je dále verifikován pomocí testovací úlohy se známým analytickým řešením. Následně je popsána mřížková Boltzmannova metoda (LBM) pro simulaci proudění tekutin. Práce vychází z výpočetního kódu vyvíjeného na Katedře matematiky, FJFI ČVUT v Praze, který je zde použit s důrazem na škálovatelnou implementaci pro superpočítače využívající grafické akcelerátory. Následně je představena výpočetní metoda využívající kombinaci metod LBM a MHFEM. Vlastnosti spojeného numerického schématu jsou zkoumány na jednoduchém modelu obsahujícím Navierovy–Stokesovy rovnice a lineární advekčně–difúzní rovnici. Nakonec je odvozen matematický model šíření vodní páry v turbulentní mezní vrstvě vzduchu nad nerovným povrchem. Numerické výsledky dosažené pomocí popsaného přístupu založeného na kombinaci LBM a MHFEM jsou validovány pomocí kvalitativního i kvantitativního srovnání s experimentálními daty naměřenými ve větrném tunelu ve třech konfiguracích s různými režimy proudění. Obsažené výsledky mohou sloužit jako první krok pro vývoj efektivního a flexibilního řešiče pro pokročilé aplikace.

Abstract

The thesis presents a multidisciplinary work that combines knowledge from the fields of mathematical modeling, computational science and experimental physics.

The first part of the thesis provides a review of programming techniques for modern parallel architectures and presents two efficient and configurable data structures for the representation of multidimensional arrays and conforming unstructured meshes, respectively. Both data structures are implemented in the Template Numerical Library (TNL) and natively support GPU-accelerated computing as well as distributed computing with MPI.

The second part deals with numerical methods in computational fluid dynamics and their applications. First, a general solver based on the mixed-hybrid finite element method (MHFEM) is developed and verified using a benchmark problem with a known analytical solution. Next, the lattice Boltzmann method (LBM) for the simulation of fluid flow is described. This work builds on the code developed at the Department of Mathematics, FNSPE CTU in Prague, and focuses on the scalable implementation for GPU-accelerated supercomputers. Then, a novel coupled computational approach based on the combination of LBM and MHFEM is presented. The properties of the coupled numerical scheme are investigated using a model problem based on the Navier–Stokes equations coupled with a linear advection–diffusion equation. Finally, a mathematical model for water vapor transport in turbulent air flow above a disturbed soil surface is developed and solved numerically using the coupled LBM-MHFEM solver. The results are validated by qualitative as well as quantitative comparison to experimental data measured in a climate-controlled wind tunnel in three configurations with different flow regimes. The presented work can serve as the first step towards the development of an efficient and flexible multiphysics solver.

Acknowledgements

It is my pleasure to thank all giants on whose shoulders this thesis stands and everyone who supported me during my Ph.D. studies. First of all, I thank my supervisors, Tomáš Oberhuber and Radek Fučík, for their expert guidance that helped to shape my research work. I also thank Tissa H. Illangasekare and Andrew C. Trautz for their invaluable advice, ideas and repeated opportunities to visit their institutions. Finally, I thank all members of the Mathematical Modeling Group formed around Michal Beneš for providing me with a comfortable working environment with a friendly atmosphere.

Our work on the research topics related to this thesis was supported by the following projects:

- *Application of advanced supercomputing methods in mathematical modeling of natural processes*, project no. SGS17/194/OHK4/3T/14 of the Grant Agency of the Czech Technical University.
- *Investigation of shallow subsurface flow with phase transitions*, project no. 17-06759S of the Czech Science Foundation.
- *Large structures in boundary layers over complex surfaces under high Reynolds numbers*, project no. 18-09539S of the Czech Science Foundation.
- *Analysis of flow character and prediction of evolution in endovascular treated arteries by magnetic resonance imaging coupled with mathematical modeling*, project no. NV19-08-00071 of the Ministry of Health of the Czech Republic.
- *Computational Models and Experimental Investigation of Fluid Dynamics, Mass Transfer and Transport, and Phase Transitions in Porous Media for Environmental Applications*, project Inter-Excelence MSMT no. LTAUSA19021 of Ministry of Education, Youth and Sports of the Czech Republic.
- *Research centre for informatics*, project no. CZ.02.1.01/0.0/0.0/16_019/0000765, Operational Programme of Research, Development and Education, Ministry of Education, Youth and Sports of the Czech Republic.
- *Development and application of advanced mathematical modeling methods for natural and industrial processes using high-performance computing*, project no. SGS20/184/OHK4/3T/14 of the Grant Agency of the Czech Technical University.
- *Multiphase flow, transport, and structural changes related to water freezing and thawing in the subsurface*, project no. 21-09093S of the Czech Science Foundation.
- *e-INFRA CZ* project ID:90140 of Ministry of Education, Youth and Sports of the Czech Republic.

This thesis was created with the help of many open-source software projects, namely LaTeX, Asymptote, Python, and ParaView.

Author's Declaration

I confirm having prepared the thesis on my own and having listed all used sources of information in the bibliography.

Prague, March 31, 2023 Jakub Klinkovský

Contents

Introduction

Mathematical modeling of fluid dynamics has many ecological, medical and industrial applications and it is one of the central research topics investigated at the Department of Mathematics, FNSPE CTU in Prague in collaboration with prominent domestic as well as foreign institutions. In order to accurately model complex natural processes governing the behavior of fluids, it is often necessary to employ advanced numerical methods capable of treating multiscale and multiphysics cases. Multiscale modeling refers to techniques that resolve fundamental physical processes at many different temporal and/or spatial scales. On the other hand, multiphysics models comprise several parts describing specific aspects of a large system, such as thermal distribution in a flowing fluid. Both factors bring additional challenges to the development of mathematical models as well as numerical methods that can be applied.

Accurate numerical simulations in high resolution are possible only on large computational clusters or supercomputers. However, using the facilities for high-performance computing efficiently is non-trivial, as it requires careful management of data in the computer memory and appropriate division of the computations between all available processing units. Especially when designing algorithms for systems with GPU accelerators, which provide significantly more processing units as well as memory levels compared to traditional computational systems, specifics of the hardware architectures have to be considered in the software design. Due to the diversity of parallel computing platforms, it is desirable for scientists to use established high-level libraries that provide a portable and easy to use interface for common operations. However, it may be difficult to combine different packages and libraries, or even to gain sufficient overview of the available options.

Since the field of computational fluid dynamics includes many substantially different applications, the most important requirements imposed on the building blocks of numerical solvers are configurability and composability. The former allows to adapt a piece of software, such as a data structure, for a specific application. The latter allows to combine multiple existing pieces together to resolve a new use case. With these two design aspects, low-level components can be used to develop tools and solvers, which are also configurable and composable on a higher level. On the highest level, a multiphysics solver typically comprises a hierarchy of components that are coupled together and configured for the needs of a specific problem.

This thesis pursues topics from two levels of interest. At one level, it deals with the development of fundamental building blocks, such as efficient and reusable data structures and parallel algorithms. Specifically, the data structures described in this thesis allow to organize structured as well as unstructured data in numerical simulations according to the requirements for efficient utilization of modern supercomputers. At the other level, these building blocks are used as fundamental ingredients for the development of advanced numerical solvers in computational fluid dynamics. The solvers described in this thesis are based on two main numerical methods, namely the mixed-hybrid finite element method and the lattice Boltzmann method. Both methods are thoroughly tested separately from each other and then a coupled computational approach based on both methods is introduced. Finally, the resulting coupled solver is applied in practice for the simulation of water vapor transport in turbulent air flow.

State of the art

Contemporary supercomputers are based on several different hardware platforms and each has an associated parallel programming framework providing native performance. For example, CUDA [M19] targets NVIDIA GPU accelerators, ROCm [M2] targets AMD GPU accelerators, and TBB [O13] targets multi-core processors. In order to simplify parallel programming and provide performance portability, high-level libraries such as Thrust [O26], Kokkos [A170], or TNL [A142] are being developed. Additionally, established scientific libraries such as Trilinos [O37] or PETSc [O2] are being updated in order to support GPU accelerators via suitable interfaces. However, this often requires a complete redesign of an algorithm due to hardware platforms having different requirements on optimal data layout.

Practically all numerical solvers in computational fluid dynamics utilize methods of linear algebra, which provides a well-established framework for representing scientific problems. Many data structures and algorithms for GPU accelerators are available in common libraries such as cuBLAS [M19] or cuSPARSE [M19]. An important problem in linear algebra is the solution of sparse linear systems, which is covered later in this thesis in Chapter 3.

Unlike linear algebra, other fields such as stencil computations on grids are not as established and may not even have a fixed universal system of notation. Hence, there is no obvious way to formulate problems and the development of reusable software components is inherently difficult. Overall, there is a lack of software libraries providing robust and multi-platform data structures for the representation of unstructured meshes. Moreover, multiple types of meshes are used in practice, e.g., tetrahedral, hexahedral, and polyhedral. While tetrahedral meshes are the easiest to use in numerical methods such as finite elements, hexahedral and especially polyhedral meshes are advantageous in complex cases with high amount of data per cell. Compared to other types, polyhedral meshes need smaller number of cells to cover a given domain and numerical methods such as finite volumes can be designed carefully with the same level of accuracy that can be achieved on tetrahedral meshes [A178].

Numerous computational tools based on numerical methods such as finite volumes or finite elements are available for solving partial differential equations originating from mathematical modeling of various biological, environmental, or industrial problems. In particular, software projects such as deal.II [A17], DUNE [C10], OpenFOAM [C26], TOUGH2 [M24], MFiX [M27], ANSYS Fluent [M3] or COMSOL Multiphysics [M8] are suitable for simulations in the field of computational fluid dynamics. All the aforementioned projects provide some parallel computing capabilities. While efficient implementations of the finite volume and finite element methods for GPU accelerators are available [A20, A42–A44, A67, A189], the aforementioned projects provide only limited or no support for GPU-accelerated computations. Additionally, the lattice Boltzmann method (LBM) has become popular for turbulent flow simulations [A84, A106, A118, A147, A181, A188]. Unlike traditional numerical approaches such as finite volume or finite element methods, the parallelization of the LBM algorithm is simpler and most computational software that employs LBM also supports computations on GPU accelerators.

While many of the aforementioned projects are open-source and thus can be freely modified, it is difficult to incorporate novel approaches and methods into extensive software packages, especially for external users. Hence, significant stream of innovation originates from small separate projects that gradually either evolve into larger projects, or get incorporated into existing software. Many such research projects were started separately at the Department of Mathematics, FNSPE CTU in Prague, including an in-house code implementing the lattice Boltzmann method, and the author's previous work [B20], an implementation of the mixed-hybrid finite element method for multiphase compositional flow in porous media. The work described in this thesis integrates, extends and generalizes several such components.

Research goals

Based on the previous section, the following goals were identified for our research. To the best of our knowledge, they represent unique ideas that push forward the frontiers of the state of the art.

1. To develop the Template Numerical Library (TNL) [A142], an open-source software library of high-performance algorithms and efficient data structures that follows modern software design patterns and simplifies the development of CFD solvers.

   Why develop yet another numerical library? The short answer is: Why not? There are many competing libraries with similar features even in established fields such as linear algebra, where one might hope for the existence of a universal project collecting contributions from all over the world. However, competition inspires innovation and maintaining a separate project provides its developers with an easy way to try and share ideas with others.

   Note that this goal is never-ending, it lies in continuous effort rather than reaching a milestone.

2. To develop an efficient data structure for the representation of conforming unstructured meshes with full support of modern distributed computing platforms based on GPU accelerators.

3. To extend the author's previously developed solver based on the mixed-hybrid finite element method with distributed computing capabilities.

4. To develop a scalable solver based on the lattice Boltzmann method for GPU-based supercomputers.

5. To develop a mathematical model of vapor transport by air flow above a soil surface, implement a high-performance solver for the model, and validate it using experimentally measured data.

Achieved results

The thesis presents the following novel results:

1. **Data structure for conforming unstructured meshes supporting distributed computing on GPU accelerators.** The data structure is implemented in the TNL library [A142] and several benchmarks were performed to evaluate its efficiency, showing that for triangular and tetrahedral meshes it outperforms the MOAB library [M28] by at least an order of magnitude. We have published our results in [A110]. Since the publication of the paper, the data structure has been extended for polygonal and polyhedral meshes and these modifications are included in the thesis.

2. **General solver based on the mixed-hybrid finite element method (MHFEM) with distributed computing capabilities.** This extends the author's previous work [B20] and serves as an important building block for other results in this thesis. The solver uses the aforementioned distributed data structure for unstructured meshes and can utilize the Hypre library [C20] for efficient solution of sparse linear systems via wrappers implemented in the TNL library. Our results related to the development of the MHFEM solver are included in the publications [A72, A110].

3. **Scalable implementation of the lattice Boltzmann method (LBM) for supercomputers based on GPU accelerators.** The implementation is based on a distributed multidimensional array data structure and an MPI synchronizer for distributed data, which are implemented in the TNL library. Strong scaling as well as weak scaling studies were performed on the Karolina supercomputer [O15]. The lattice Boltzmann method code is developed by the research group

at the Department of Mathematics, FNSPE CTU in Prague and the author's contributions are included in the publications [A23, A68–A70].

4. **Coupled computational approach based on the combination of lattice Boltzmann method and mixed-hybrid finite element method.** We consider a model based on the Navier–Stokes equations (solved by LBM) coupled with a linear advection–diffusion equation (discretized using MHFEM) and analyze properties of the numerical scheme and performance of its implementation. A simple benchmark problem with analytical solution is constructed in order to investigate accuracy of the scheme for conservative and non-conservative form of the transport equation. The solver benefits from native implementation of both LBM and MHFEM for GPU accelerators, which allows for efficient coupling between the methods. Our results are included in the publication [A111].

5. **Mathematical model for vapor transport in air flow and its validation using experimental data.** The aforementioned coupled computational approach has been used to model transport of water vapor in the turbulent boundary layer above a disturbed soil surface. While the porous medium below the surface is not simulated in this work, the interaction between soil and atmosphere such as evaporation is modeled using boundary conditions. The model is compared both qualitatively and quantitatively to experimental data measured in three configurations resulting in different flow regimes. Our results are included in the publication [A111].

Outlook

During the work on this thesis, we identified several interesting problems and directions for future research. Here, we summarize a few of them.

The coupled LBM-MHFEM solver that was validated for vapor transport in air is the first step towards the development of a flexible multiphysics solver. In the future, it can be extended to include coupling with porous medium, thermodynamic effects, or component transport. The primary use case for such solver is to investigate land-atmospheric interactions with soil surface disturbances (e.g., vegetation, micro-topography) and macro-porous disturbances in the subsurface. The resulting solver would facilitate the symbiosis between modeling and experimental approaches for the investigation of environmental processes.

The numerical models used in this work have several problems that hinder their application in large-scale and complex scenarios. Notably, our MHFEM implementation is limited to fixed meshes that cannot adapt to the simulated fields, and the LBM implementation assumes a uniform lattice. The development of schemes for adaptive meshes and non-uniform lattice discretizations may be necessary for successful use of these methods to solve real-world problems.

The performance and efficiency of the developed solvers was tested on the largest computing systems that were available to the author at that time. However, many supercomputers in the world are much larger and the scaling of the presented solvers on such systems should be investigated. Furthermore, hardware as well as software platforms powering contemporary supercomputers are continuously evolving. The TNL library supports parallelization via OpenMP and CUDA for multi-core CPUs and NVIDIA GPUs, but the implementation of new backends (e.g., using HIP and SYCL) is needed to provide support for other hardware platforms.

Finally, the TNL library has proved to be an effective tool for the development of scalable and flexible numerical solvers. Its development requires continuous effort in order to keep up with the world. The most important general directions of future development, as viewed by the author, are interoperability with other libraries, improving the modular structure of the project, and using formal *concepts* in the C++20 standard to improve type-checking, documentation, and compiler diagnostics.

Chapter 1

Programming Techniques for Modern Parallel Architectures

The world entered the exascale era of computing in 2022 when the Frontier supercomputer [O40] was installed with a performance of 1.102 exaFLOPS measured in the LINPACK benchmarks suite [O32, O33]. This thesis originated in the pre-exascale era of computing and does not actually target the world's fastest supercomputers. The fastest computing system available to the author during his work on the thesis was the Karolina supercomputer [O15], which was ranked 85th place in the TOP500 list in November 2022 [O34] as the fastest supercomputer in the Czech Republic, with a performance of 6.75 petaFLOPS measured in the LINPACK benchmarks suite [O34].

Since 2007 [C24, M18], graphical processing units (GPUs) have been used more and more as general-purpose computing accelerators. Over time, they evolved into powerful and efficient massively parallel devices that drive the computational performance of contemporary supercomputers thanks to their energy- and cost-efficiency [C4, A34, C14, A138, C35, C38]. As of November 2022, 7 out of 10 most powerful supercomputers according to the TOP500 list are based on GPU accelerators [O35]. Parallel compute accelerators such as GPUs are based on a conceptually different hardware architecture compared to traditional processors based on the x86/x86-64 architectures. Much has been written about characteristics and evolution of these platforms [B1, A34, C24]. This thesis does not aim to repeat the summary, nevertheless, we need to highlight the main GPU hardware features:

- high number of simpler compute units (cores),
- smaller caches and orientation to data-parallel applications,
- groups of compute units and stacks of high-bandwidth global memory organized in scalable hierarchies.

Overall, GPU accelerators are advantageous for compute-bound as well as memory-bound data parallel applications.

To make the collection of individual accelerators and compute nodes scalable to the size of supercomputers, fast and scalable interconnections between the individual units are necessary. Technologies such as NVLink and NVSwitch [A123, M21] allow for high-throughput and low-latency communication between GPUs in a single node. Inter-node communication typically relies on switched fabric network interconnections. State–of–the–art solutions for high-performance computing provide transfer speeds up to 100 Gbit/s per link [O41], latency around 0.5 µs [O41], remote direct memory access (RDMA) capabilities to minimize CPU overhead [A123, C34], and acceleration of collective communication operations [C22, C36]. Compute nodes can be organized in various network topologies such as fat tree or dragonfly with multiple levels of network switches and links on higher levels can be aggregated in order to increase the overall bandwidth of the network.

This thesis deals with the development of numerical solvers for parallel platforms, especially GPUs that are indispensable for high-performance computing on modern systems. Even though the implemented methods were not tested on the world's largest supercomputers, we believe it should be possible to upscale most of the developed algorithms and data structures to larger computing systems thanks to similarity between GPU architectures.

The main programming language used in this thesis is C++, which is a state–of–the–art programming language for modern, high-performance applications. In the following sections, we review the parallel programming frameworks and high-level libraries designed to simplify parallel programming in C++. The last section describes the Template Numerical Library where the author is a main developer and which collects the author's original work related to this chapter.

1.1 Parallel programming frameworks

This section provides a brief summary of parallel programming frameworks most commonly used in high-performance computing. To compare the individual programming styles, we show examples of a parallel *axpy* operation, i.e., the computation of $\boldsymbol{y} := \alpha\boldsymbol{x}+\boldsymbol{y}$ for given vectors $\boldsymbol{x}$, $\boldsymbol{y}$ and scalar multiplier α. In the C language, a sequential *axpy* operation can be implemented as shown below, where `index_t` and `value_t` are user-defined aliases for the indexing type (e.g., `int`) and value type (e.g., `float`), respectively.

```
void sequential_axpy(index_t size, value_t alpha, value_t* x, value_t* y)
{
    for (index_t i = 0; i < size; i++)
        y[i] = alpha * x[i] + y[i];
}
```

Note that the list of frameworks discussed in the following subsections is not exhaustive. Other high-performance computing frameworks related to the C and C++ languages include Boost.Thread [O42], HPX [A105, O16], RaftLib [C11], and Unified Parallel C [M7].

1.1.1 POSIX threads (multi-core CPUs)

POSIX threads (pthreads) [M26] is a parallel execution model for multi-core processors and shared memory systems. It is an application programming interface (API) standardized in 1996 with implementations available for many Unix-like systems as well as Microsoft Windows. The API provides complete functionality for thread management, synchronization, mutexes and semaphores, condition variables, and spinlocks. The API consists of a set of types, functions and constants defined in the C programming language, which can be used from practically any other higher-level language. Details on programming with POSIX threads can be found in [B9].

A parallel *axpy* operation can be implemented using POSIX threads as follows. First, we must define a structure to store the parameters that will be passed to the thread function. For our example, we need to pass the range of elements to be processed by the thread, which is represented by the `begin` and `end` indices, the scalar multiplier `alpha`, and pointers to the vectors `x` and `y`:

```
typedef struct
{
    index_t begin;
    index_t end;
    value_t alpha;
    value_t* x;
    value_t* y;
} arg_struct;
```

The function `thread_axpy` that will be executed by each thread must have the signature `void* thread_axpy(void* arg)`, where `arg` is an opaque pointer to the function arguments. The thread function converts the pointer to `arg_struct*`, computes the sequential *axpy* operation on the given range of elements, and calls `pthread_exit` at the end:

```
void* thread_axpy(void* arg)
{
    // get the parameters
    arg_struct* param = arg;

    // sequential axpy on the given range of elements
    for (index_t i = param->begin; i < param->end; i++)
        param->y[i] = param->alpha * param->x[i] + param->y[i];

    pthread_exit(NULL);
}
```

The final piece of the implementation is the partitioning of the work among threads, initialization of the argument structures for each thread, and launching the thread functions. In the example below, we set the number of threads to the number of CPU cores in the system and partition the array size uniformly among the threads. Note that the storage containing arguments for a thread must not be changed during its execution, so we create an array of these structures similarly to the array of threads. The threads are created with their compute function and arguments in a loop using the `pthread_create` function and the `pthread_join` function is used in a separate loop to wait for all threads to finish their work.

```
void parallel_axpy(index_t size, value_t alpha, value_t* x, value_t* y)
{
    // set number of threads (equal to the number of cores in the system)
    int num_threads = sysconf(_SC_NPROCESSORS_ONLN);
    // calculate the block size
    index_t block_size = (size + num_threads - 1) / num_threads;

    // reserve space for thread objects
    pthread_t threads[num_threads];
    // reserve space for thread args
    arg_struct thread_args[num_threads];

    for (int i = 0; i < num_threads; i++) {
        // initialize parameters for the i-th thread
        thread_args[i].begin = i * block_size;
        thread_args[i].end = (i + 1) * block_size;
        if (thread_args[i].end > size)
            thread_args[i].end = size;
        thread_args[i].alpha = alpha;
        thread_args[i].x = x;
        thread_args[i].y = y;
        // create the i-th thread
        pthread_create(&threads[i], NULL, thread_axpy, (void*) &thread_args[i]);
    }

    // wait for all threads to finish their work
    for (int i = 0; i < num_threads; i++)
        pthread_join(threads[i], NULL);
}
```

1.1.2 Threading Building Blocks

Threading Building Blocks (TBB) [O13] is a C++ template library developed by Intel for parallel programming on multi-core processors. It was originally developed as a separate library and later became part of the oneAPI collection under the name oneTBB. It is a high-level library, but still allows the programmer strong control over the parallel execution details affecting the performance. TBB was created in 2006 and its design patterns influenced the inclusion of parallel programming support in modern C++ standards. TBB provides functionality for thread management and task scheduling, scalable memory allocation, parallel algorithms, and concurrent containers. Details on programming with TBB can be found in [B32].

TBB provides a high-level function `tbb::parallel_for` that allows to implement a parallel *axpy* operation analogously to the POSIX threads approach using just a few lines of code:

```
void parallel_axpy(index_t size, value_t alpha, value_t* x, value_t* y)
{
    tbb::parallel_for(
        // global range of elements
        tbb::blocked_range<std::size_t>(0, size),
        // compute function executed by threads
        [=](const tbb::blocked_range<std::size_t>& r) {
            // process the local range `r` sequentially
            for (std::size_t i = r.begin(); i < r.end(); ++i)
                y[i] = alpha * x[i] + y[i];
        }
    );
}
```

The first parameter of the `tbb::parallel_for` function represents the global range of elements that is split into subranges that are processed by individual threads. The second parameter of the `tbb::parallel_for` function defines the compute function that is executed by a particular thread for its subrange of elements that is denoted by the parameter `r` of the inner function. Here, the compute function is implemented using a *lambda expression*, a feature introduced in the C++11 language standard. Since the compute function operates only with scalar values and pointers, all variables can be captured by value and thus the capture list of the lambda expression contains just = for simplicity.

1.1.3 OpenMP and OpenACC

OpenMP [A51] is a multi-platform parallelization API for shared-memory systems. It is based on compiler directives and provides library routines to query and modify the execution context at run-time. Furthermore, the parallel execution can be affected by environment variables.

The *axpy* operation is a very simple example that can be parallelized with OpenMP by adding a single line with a `#pragma` directive before the loop in the sequential version:

```
void parallel_axpy(index_t size, value_t alpha, value_t* x, value_t* y)
{
    #pragma omp parallel for
    for (index_t i = 0; i < size; i++)
        y[i] = alpha * x[i] + y[i];
}
```

In this example, the compiler preprocessor interprets the `#pragma omp parallel for` directive and modifies the code to add parallel execution of the loop following the directive. At run-time, a number of threads is spawned, the iteration range is partitioned into chunks of contiguous elements and each chunk is processed by a thread. When all work is finished, the threads are joined and the sequential processing of code after the loop continues.

OpenMP provides many other clauses that may be used in the `#pragma omp` directives in more complex cases. It is obvious that OpenMP greatly simplifies parallelization in straightforward cases where the provided `#pragma` directives are sufficiently expressive. However, the reliance on compiler directives removes some level of control from the programmer, which may lead to workflow and performance problems. Most notably, OpenMP directives may conflict with features introduced in modern C++ standards [O11]. Also due to *ghost code* added by the compiler in place of `#pragma` directives, debugging synchronization bugs, race conditions and performance regressions may be more difficult compared to other frameworks. Furthermore, OpenMP does not guarantee interoperability with other multi-threading libraries. Hence, combining OpenMP with TBB, POSIX threads, or other library results in undefined behavior, which leads to non-portable code in the best case.

OpenACC [M29] is a standard similar to OpenMP, which is designed to simplify parallel programming for heterogeneous systems consisting of multi-core processors and accelerators. The programming style of OpenACC has basically the same advantages and disadvantages as OpenMP. The *axpy* operation can also be parallelized in OpenACC by adding a single line with a `#pragma` directive:

```
1 void parallel_axpy(index_t size, value_t alpha, value_t* x, value_t* y)
2 {
3     #pragma acc parallel loop
4     for (index_t i = 0; i < size; i++)
5         y[i] = alpha * x[i] + y[i];
6 }
```

Depending on the compiler options, the parallel loop may be executed either on CPU or offloaded to an accelerator.

Several years after OpenACC was introduced, the OpenMP standard version 4.0 from 2013 added support for offloading parallel execution to accelerators. However, the need for handling data transfers further complicates the API and as of 2023, the support for offloading in OpenMP compilers is still very limited or not available at all. Overall, high-level parallelization approaches such as OpenMP or OpenACC cannot compete with native frameworks such as CUDA in terms of performance [C9].

1.1.4 OpenCL

OpenCL [M10] is a parallel programming framework for heterogeneous platforms consisting of CPUs, GPUs, and other hardware accelerators. It is based on the C and C++ languages, but it does not follow a single-source approach and the *kernel functions* (i.e., code that is executed on the accelerators) must be compiled separately from the main program and loaded at run-time. Unlike OpenMP and OpenACC, OpenCL is a low-level language standard that provides higher performance and wider range of implementations for different hardware platforms.

It is hard to demonstrate a simple parallel program implemented in OpenCL. As the kernel function for *axpy* must be compiled separately, we embed its source code in a string in the main program for simplicity. For a more flexible approach in larger projects, loading of external files could be implemented. The function is annotated with the `__kernel` specifier and the pointers to the global memory of the accelerator with `__global`:

```
1 const char* kernel_source =
2     "__kernel void axpy(int size,                                    \n"
3     "                   float alpha,                                 \n"
4     "                   __global float* x,                           \n"
5     "                   __global float* y)                           \n"
6     "{                                                               \n"
7     "   int i = get_global_id(0);                                    \n"
8     "   if (i < size)                                                \n"
```

```
    "       y[i] = alpha * x[i] + y[i];                                    \n"
    "}                                                                     \n";
```

Since the main program involves calling many OpenCL runtime functions that may result in errors, we first define an auxiliary function for checking the state of an error variable:

```
void checkErr(cl_int err, const char* name)
{
    if (err != CL_SUCCESS) {
        fprintf(stderr, "ERROR: %s returned code %d\n", name, err);
        exit(EXIT_FAILURE);
    }
}
```

The implementation of the parallel *axpy* operation follows by getting the OpenCL platform ID, obtaining a device ID, and creating an OpenCL compute context and command queue:

```
void
parallel_axpy(index_t size, value_t alpha, value_t* x, value_t* y)
{
    // error code returned from api calls
    int err;

    // get the platform ID
    cl_platform_id platform;
    err = clGetPlatformIDs(1, &platform, NULL);
    checkErr(err, "clGetPlatformIDs");

    // connect to a compute device
    cl_device_id device_id;
    err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
    checkErr(err, "clGetDeviceIDs");

    // create a compute context
    cl_context context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
    checkErr(err, "clCreateContext");

    // create a command queue
    cl_command_queue command_queue =
    clCreateCommandQueueWithProperties(context, device_id, NULL, &err);
    checkErr(err, "clCreateCommandQueueWithProperties");
```

At this point, the separate source code stored in the `kernel_source` string can be loaded into the OpenCL runtime and compiled. The desired kernel function (named `axpy`) is extracted into the `kernel` object:

```
    // create the compute program from the source string
    cl_program program = clCreateProgramWithSource(context, 1, &kernel_source, NULL, &err);
    checkErr(err, "clCreateProgramWithSource");

    // build the program executable
    err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
    checkErr(err, "clBuildProgram");

    // create the compute kernel in the program we wish to run
    cl_kernel kernel = clCreateKernel(program, "axpy", &err);
    checkErr(err, "clCreateKernel");
```

Before the kernel can be launched, the input data must be prepared. In our case, we need to allocate arrays in the device memory and copy the data from the `x` and `y` arrays in the host memory. The

allocation is a blocking operation on the context and the copy operations are enqueued to the command queue:

```
36      // allocate the x and y arrays in device memory
37      cl_mem device_x;
38      cl_mem device_y;
39      device_x = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(value_t) * size, NULL, NULL);
40      device_y = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(value_t) * size, NULL, NULL);
41      if (!device_x || !device_y) {
42          printf("ERROR: failed to allocate device memory\n");
43          exit(EXIT_FAILURE);
44      }
45
46      // copy the x and y arrays from host memory to device memory
47      err = clEnqueueWriteBuffer(command_queue, device_x, CL_TRUE, 0, sizeof(value_t) * size, x, 0, NULL,
        ↪  NULL);
48      checkErr(err, "clEnqueueWriteBuffer");
49      err = clEnqueueWriteBuffer(command_queue, device_y, CL_TRUE, 0, sizeof(value_t) * size, y, 0, NULL,
        ↪  NULL);
50      checkErr(err, "clEnqueueWriteBuffer");
```

When the input data is prepared, arguments of the kernel function can be set and the kernel can be enqueued to the command queue. Note that in this example we assume that `index_t` is an alias for `int` and `value_t` is an alias for `float` in order to match the types used in the kernel function.

```
51      // set the compute kernel arguments
52      err = 0;
53      err |= clSetKernelArg(kernel, 0, sizeof(index_t), &size);
54      err |= clSetKernelArg(kernel, 1, sizeof(value_t), &alpha);
55      err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &device_x);
56      err |= clSetKernelArg(kernel, 3, sizeof(cl_mem), &device_y);
57      checkErr(err, "clEnqueueWriteBuffer");
58
59      // execute the kernel over the entire range of our 1d input data set
60      // using the maximum number of work group items for this device
61      size_t global_size = size;
62      err = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global_size, NULL, 0, NULL, NULL);
63      checkErr(err, "clEnqueueNDRangeKernel");
```

Finally, when the execution of the kernel is finished, the result can be copied from the device memory to the host memory:

```
64      // wait for the command queue to get serviced before reading back results
65      clFinish(command_queue);
66
67      // copy the y array from device memory to host memory
68      err = clEnqueueReadBuffer(command_queue, device_y, CL_TRUE, 0, sizeof(value_t) * global_size, y, 0,
        ↪  NULL, NULL);
69      checkErr(err, "clEnqueueReadBuffer");
```

For correctness, allocated resources are released after successful completion:

```
70        // release allocated resources
71      clReleaseMemObject(device_x);
72      clReleaseMemObject(device_y);
73      clReleaseKernel(kernel);
74      clReleaseProgram(program);
75      clReleaseCommandQueue(command_queue);
76      clReleaseContext(context);
77  }
```

1.1.5 CUDA

CUDA [M19] is a parallel programming framework and API for general-purpose computing on NVIDIA GPU accelerators. It is based on the C++ programming language with certain extensions and limitations. CUDA is also accessible from many other programming languages via either direct bindings or high-level interface libraries. The CUDA toolkit provides a compiler toolchain for the CUDA language, developer tools facilitating debugging and optimization, a rich set of accelerated scientific libraries, and an extensive documentation with examples and tutorials. Together with OpenCL, CUDA has pushed the frontiers of high-performance computing on general-purpose GPU accelerators and inspired the creation of other parallel programming frameworks, such as ROCm/HIP and SYCL.

Unlike OpenCL, CUDA provides a single-source embedded domain-specific language (eDSL), called CUDA Runtime API, where the code for the execution on host and device can be defined in the same source file. This greatly improves the programming productivity and provides better integration between the host and device sides in terms of type checking and supported features (e.g., templates).

The kernel function for the *axpy* operation, which is executed by threads on the GPU, can be implemented similarly to the preceding OpenCL example. The main difference is that the function can be defined directly in the source code rather than a string:

```
__global__ void kernel_axpy(index_t size, value_t alpha, value_t* x, value_t* y)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < size)
        y[i] = alpha * x[i] + y[i];
}
```

As in the OpenCL example, we define an auxiliary function for convenient error checking:

```
void checkErr(cudaError_t err, const char* name)
{
    if (err != cudaSuccess) {
        fprintf(stderr, "ERROR: %s returned code %d\n", name, err);
        exit(EXIT_FAILURE);
    }
}
```

The ideas behind the parallel *axpy* function are also similar to the OpenCL example, but the implementation is overall much simpler as the programmer only needs to deal with data management and kernel launch. First, we need to allocate arrays in the device memory and copy the data from the `x` and `y` arrays in the host memory:

```
void parallel_axpy(index_t size, value_t alpha, value_t* x, value_t* y)
{
    cudaError_t err;

    // allocate the x and y arrays in device memory
    value_t* device_x, device_y;
    err = cudaMalloc((void**) &device_x, sizeof(value_t) * size);
    checkErr(err, "cudaMalloc");
    err = cudaMalloc((void**) &device_y, sizeof(value_t) * size);
    checkErr(err, "cudaMalloc");

    // copy the x and y arrays from host memory to device memory
    err = cudaMemcpy(device_x, x, sizeof(value_t) * size, cudaMemcpyHostToDevice);
    checkErr(err, "cudaMemcpy");
    err = cudaMemcpy(device_y, y, sizeof(value_t) * size, cudaMemcpyHostToDevice);
    checkErr(err, "cudaMemcpy");
```

When the input data is prepared in the device memory, the kernel can be launched with thread block and grid sizes appropriate for the size of input dataset. Without explaining the thread hierarchy in the CUDA programming model, these values can be used for the `axpy` operation with 1D data:

```
18      dim3 blockSize, gridSize;
19      blockSize.x = 256;
20      gridSize.x = (size + blockSize.x - 1) / blockSize.x;
21      kernel_axpy<<<gridSize, blockSize>>>(size, alpha, device_x, device_y);
```

Finally, we wait for the kernel execution to finish, copy the result from device memory to host memory, and deallocate the temporary arrays:

```
22      // wait for the kernel to finish before reading back results
23      cudaDeviceSynchronize();
24
25      // copy the y array from device memory to host memory
26      err = cudaMemcpy(y, device_y, sizeof(value_t) * size, cudaMemcpyDeviceToHost);
27      checkErr(err, "cudaMemcpy");
28
29      // release allocated resources
30      cudaFree(device_x);
31      cudaFree(device_y);
32  }
```

1.1.6 ROCm and HIP

ROCm [M2] is a software platform for parallel programming of AMD general-purpose GPU accelerators. It serves as the driver for code execution of several frameworks, including OpenMP, OpenCL, and HIP. HIP [M1] is a portable interface designed as a generalization of the CUDA Runtime API, providing a similar programming model for AMD GPUs. HIP can be installed with either AMD ROCm or NVIDIA CUDA as the target platform, but the general interface has some limitations due to hardware differences and the latter is not fully supported by AMD.

The parallel *axpy* operation can be implemented almost identically to the preceding CUDA example: it is sufficient to replace all occurrences of `cuda` in the source code with `hip` (e.g., `hipMalloc` instead of `cudaMalloc` and so on). Hence, we omit the example for the HIP platform in this work.

1.1.7 SYCL

SYCL [M11] is a single-source embedded domain-specific language (eDSL) based on pure C++17 providing an open standard for heterogeneous computing. It evolved from the OpenCL standard and has been generalized to be able to target other systems. Multiple implementations of the SYCL standard exist providing support for many hardware architectures. The most mature implementation is Intel DPC++ [O12] from the oneAPI collection, which targets primarily Intel CPUs, GPUs, and FPGAs.

SYCL allows for a programming style similar to CUDA and HIP, where data movement and dependencies between individual operations are managed explicitly by the programmer. Additionally, SYCL provides the concepts of *buffers* and *accessor* that allow automatic asynchronous scheduling of data movement based on the data accesses in individual device kernels and host functions.

In the following, we show an implementation of the parallel *axpy* operation with explicit data movement. First, a type must be declared at namespace scope to provide the name of the device kernel:

```
1  struct kernel_axpy;
```

The parallel *axpy* function can be implemented following the same general structure as in the preceding OpenCL and CUDA examples:

```
void parallel_axpy(index_t size, value_t alpha, value_t* x, value_t* y)
{
    // create a queue for a default-selected device
    auto queue = sycl::queue{sycl::default_selector{}};

    // allocate the x and y arrays in device memory
    value_t* device_x = sycl::malloc_device<value_t>(size, queue);
    value_t* device_y = sycl::malloc_device<value_t>(size, queue);
    if (!device_x || !device_y) {
        std::cerr << "ERROR: could not allocate device arrays" << std::endl;
        std::terminate();
    }

    // copy the x and y arrays from host memory to device memory
    queue.memcpy(device_x, x, sizeof(value_t) * size).wait();
    queue.memcpy(device_y, y, sizeof(value_t) * size).wait();

    // execute a device kernel
    queue.parallel_for<kernel_axpy>(
        sycl::range(size),
        [=](sycl::item<1> item) {
            auto i = item.get_id();
            device_y[i] = alpha * device_x[i] + device_y[i];
        }
    ).wait();

    // copy the y array from device memory to host memory
    queue.memcpy(y, device_y, sizeof(value_t) * size).wait();

    // release allocated resources
    sycl::free(device_x, queue);
    sycl::free(device_y, queue);
}
```

Notice that SYCL makes use of an explicit *queue* object where individual asynchronous operations are enqueued, whereas CUDA uses the concept of *streams* and provides a default stream that can be used in simple cases. SYCL also provides several interfaces to submit parallel tasks, the `parallel_for` interface (similar to TBB) was used above.

1.1.8 Message Passing Interface

The Message Passing Interface (MPI) [M15] is a de facto standard interface for parallel programming on distributed platforms. It is available natively for C and Fortran and interfaces exist for many other programming languages, including object-oriented libraries developed in high-level languages. MPI follows the *single program, multiple data* (SPMD) programming style, where multiple instances of the same program are executed at the same time and operate on different pieces of data from the global dataset. The interface deals with communication among the individual instances, hereafter referred to as MPI processes or ranks, via message passing. In general, MPI ranks are executed on a set of independent computing systems, hereafter referred to as nodes, which are interconnected by a network. When the ranks are executed on a single node with a shared memory, the MPI library can choose to optimize out the need for explicit message passing, otherwise messages need to be assembled by the sending process, transferred via the network layer, and disassembled by the receiving process. Thus, network communication may incur a considerable overhead to the algorithm that is not present when it is parallelized in a non-distributed manner. However, dealing with network communication is necessary to utilize compute resources provided by modern supercomputers efficiently and various techniques for the development of scalable algorithms exist.

It is not the purpose of this thesis to explain all concepts used in the MPI standard. Many books are dedicated to explaining the basics of MPI programming as well as advanced features and optimizations, for example [B17, B18]. Here we describe only the most important concept, *communicators*. A communicator is an object that provides a context for communication among a group of MPI processes. Each communicator is characterized by its ID/tag that differentiates messages from different communicators, and a group of processes that provides a rank-naming mechanism based on numeric IDs that are used for addressing of the messages. The most common communicator is denoted by the constant `MPI_COMM_WORLD` and comprises all ranks participating in the invocation of an MPI program. Communicators are represented by the type `MPI_Comm` and MPI provides several functions to create a custom communicator, for example:

- `MPI_Comm_dup` creates a duplicate of given communicator using the same process group and its rank-naming.
- `MPI_Comm_split` creates communicators by partitioning the process group associated to an existing communicator into several disjoint subgroups. Each process falls into exactly one subgroup based on the value of the `color` parameter of the `MPI_Comm_split` function, each subgroup contains all processes that specified the same `color`.
- `MPI_Comm_split_type` creates communicators similarly to `MPI_Comm_split`, but uses different criteria for partitioning. The `split_type` parameter is used instead of `color` and all ranks must supply the same value of this parameter. For example, using the type `MPI_COMM_TYPE_SHARED` splits the group into subgroups of processes that can communicate via shared memory (i.e., they are running on the same node of a distributed computing system).
- `MPI_Comm_create` creates a new communicator by specifying an arbitrary group of processes represented by the `MPI_Group` type. MPI provides several functions for manipulating process groups, for example `MPI_Group_difference`, `MPI_Group_intersection`, and `MPI_Group_union`.

1.2 High-level libraries with backend systems

This section provides an overview of libraries that provide high-level abstractions and general interfaces for common aspects of parallel computing. They are designed to enhance programmer's productivity while enabling performance portability between different hardware platforms. While an implementation of an algorithm using a standard high-level interface may not be the most optimal one, it is often good enough and it can be further optimized later when the need for improvement is identified. Note that the list is not exhaustive; only projects featuring the most successful concepts are introduced.

1.2.1 Thrust and rocThrust

Thrust [M23, O26] is a C++ library of parallel algorithms providing backends for parallel execution on CPU (via OpenMP or TBB) and GPU (via CUDA). Thrust is included as a header-only library in the CUDA toolkit. Together with TBB, Thrust inspired the introduction of parallel algorithms to the C++ Standard Library. Thrust provides a collection of algorithms such as parallel iteration, reduce, scan, sort, and other common patterns that can be expressed in terms of these primitives. The high-level interfaces of the algorithms are built on top of *iterators* [O4] that can be composed together to express complex algorithms in a concise and readable manner.

The parallel *axpy* operation for the host backend (i.e., execution on CPU) can be implemented in a straightforward way using the `thrust::transform` function:

```
void parallel_axpy_host(index_t size, value_t alpha, value_t* x, value_t* y)
{
    thrust::transform(
        thrust::host,                           // execution backend
        x,                                      // beginning of the first input sequence
        x + size,                               // end of the first input sequence
        y,                                      // beginning of the second input sequence
        y,                                      // beginning of the output sequence
        [alpha](value_t x_val, value_t y_val)   // transformation function applied to
        { return alpha * x_val + y_val; }       // pairs of input elements
    );
}
```

The individual parameters of the function are described in the comments in the code block. Note that raw pointers are used as forward random access iterators and the transformation function is implemented in terms of a *lambda expression* that captures the `alpha` variable and operates on a pair of elements from the two input iterators. The `transform` function writes the result of the lambda expression to the output iterator.

To execute the *axpy* operation on GPU using the device backend, the input data must be first transferred to the device memory and the result must be copied back to the host memory after the `thrust::transform` function is executed. This can be achieved with the `thrust::device_vector` container and the `thrust::copy` algorithm that automatically detects the appropriate memory space:

```
void parallel_axpy_device(index_t size, value_t alpha, value_t* x, value_t* y)
{
    // allocate arrays on the device
    thrust::device_vector<value_t> x_device(size);
    thrust::device_vector<value_t> y_device(size);

    // copy the arrays from host to device
    thrust::copy(x, x + size, x_device.begin());
    thrust::copy(y, y + size, y_device.begin());

    // perform the axpy operation
    thrust::transform(
        thrust::device,                         // execution backend
        x_device.begin(),                       // beginning of the first input sequence
        x_device.end(),                         // end of the first input sequence
        y_device.begin(),                       // beginning of the second input sequence
        y_device.begin(),                       // beginning of the output sequence
        [alpha] __device__ (value_t x_val, value_t y_val) // transformation function applied to
        { return alpha * x_val + y_val; }               // pairs of input elements
    );

    // copy the result from device to host
    thrust::copy(y_device.begin(), y_device.end(), y);
}
```

Note that the transformation function has been annotated as `__device__` so that it can be executed on the GPU. Thus, the example relies on the `--extended-lambda` and `--expt-relaxed-constexpr` flags to be passed to the `nvcc` compiler. It is possible to avoid these flags by implementing the transformation via a functor instead of a lambda expression, but that solution is considerably more verbose.

rocThrust [O1] is a port of the Thrust library to the ROCm/HIP platform. rocThrust provides the same interface as Thrust, so the example above works without any change even on AMD GPUs. The only part that needs to be changed is the compiler: `hipcc` can be used instead of `nvcc` and it does not need any special flags.

Note that Thrust and rocThrust provide a *zip*-iterator that can be used to iterate over more than two input iterators at the same time. This can be used, for example, to sum more than two vectors.

1.2.2 Thread support in STL

The C++17 standard defines an interface for parallel algorithms in the Standard Template Library (STL) based on *iterators* [O4], which is inspired by the TBB and Thrust libraries. The standard extends the former collection of sequential algorithms defined in the `algorithm` header file. The new interface is provided in a separate header file `execution`, however, not all STL implementations provide it. The `libstdc++` implementation from the GNU Compiler Collection (GCC) since version 9 provides an implementation of the standardized parallel algorithms using the TBB library for parallel execution.

The parallel *axpy* operation can be implemented in STL very similarly to the Thrust example, using the `std::transform` function:

```
void parallel_axpy(index_t size, value_t alpha, value_t* x, value_t* y)
{
    std::transform(
        std::execution::par_unseq,              // execution policy
        x,                                      // beginning of the first input sequence
        x + size,                               // end of the first input sequence
        y,                                      // beginning of the second input sequence
        y,                                      // beginning of the output sequence
        [alpha](value_t x_val, value_t y_val)   // transformation function applied to
        { return alpha * x_val + y_val; }       // pairs of input elements
    );
}
```

In this example, we used the `std::execution::par_unseq` execution policy that allows parallelization and vectorization of the inner function. Other execution policies defined by the C++17 standard are `std::execution::par` that allows parallelization but not vectorization within threads, and `std::execution::seq` that implies sequential execution in the calling thread. C++20 also defines `std::execution::unseq` that allows vectorization, but not parallelization. Furthermore, implementations of the standard library are allowed to provide additional execution policies, for example to target parallel execution on GPU accelerators.

Note that as of C++20, STL does not provide an interface to iterate over more than two input iterators at the same time. The concept of a *zip*-iterator is planned for the C++23 standard, which will allow to compose an arbitrary number of sequences next to each other in a single iterator. However, the interface used in TBB, where the inner function operates with a range of indices rather than iterators, is apparently simpler and more appropriate for this case.

1.2.3 Kokkos

Kokkos [A170] is a collection of modules (subprojects) that facilitate portable performance programming in C++. The fundamental module, Kokkos Core [A171], defines the programming model based on parallel execution and memory abstractions. It has influenced other frameworks (e.g., SYCL) as well as the development of C++ standards (e.g., `std::mdspan` in the future C++23 standard). Derived modules provide additional higher-level functionality, such as math kernels for dense and sparse linear algebra and graph operations. Many projects with broader scope use Kokkos for parallelization, for example Trilinos [O37].

Kokkos Core can be used either from a system-wide installation or included, configured and built within a project. As of version 3.7.1, Kokkos Core provides backends for parallel execution using C++ Threads, OpenMP, CUDA, HIP, SYCL, and HPX. At most one GPU device backend and one parallel CPU backend can be enabled at a time during Kokkos Core installation. Additionally, Kokkos Core provides a serial backend that can be used as fallback.

To use Kokkos Core in a program, its backend system must be initialized before first use and finalized after last use. This is done typically in the `main` function, for example:

```
int main(int argc, char** argv)
{
    Kokkos::initialize(argc, argv);

    // program execution...

    Kokkos::finalize();
    return EXIT_SUCCESS;
}
```

The core functionality of Kokkos Core is based on an abstract machine model consisting of several components related to data management and code execution:

- *memory space* defines *where* data resides (e.g., host memory, device global memory, registers),
- *memory layout* defines *how* the data is *stored* (e.g., row-major or column-major),
- *memory traits* define *how* the data is *accessed* by an algorithm (e.g., read-only or atomic writes),
- *execution space* defines *where* a function is executed (e.g., CPU or GPU),
- *execution policy* defines *how* a function is executed (e.g., a one-dimensional or multi-dimensional range of fully independent threads or hierarchical groups of collaborative threads),
- *execution pattern* defines the type of *operation* that is subject of parallel dispatch (e.g., `parallel_for`, `parallel_reduce`, `parallel_scan`, or single tasks with dependencies).

The strong data abstraction model is a prominent feature of Kokkos that allows for transparent memory layout changes (i.e., without rewriting algorithms) to satisfy different data access pattern on different hardware. Kokkos Core provides hierarchical parallelism with *leagues* of *teams* of threads, which was inspired by the CUDA programming model and can be mapped to any other parallel backend. At each level of the thread hierarchy, an execution pattern such as `parallel_for` or `parallel_reduce` can be used. Furthermore, Kokkos Core provides high-level algorithms such as parallel STL algorithms, and data structures such as high-performance unordered map. Further details on the Kokkos Core programming model and provided functionality can be found in its documentation [O17].

The parallel *axpy* function can be implemented using Kokkos Core as follows. Since the input arrays are specified using raw pointers as in previous examples, we first create Kokkos views in order to encode the memory space in the data structures:

```
void parallel_axpy(index_t size, value_t alpha, value_t* x_raw, value_t* y_raw)
{
    // create host views for the input arrays
    using host_view = Kokkos::View<value_t*, Kokkos::HostSpace>;
    host_view x_host(x_raw, size);
    host_view y_host(y_raw, size);
```

Next, we define space where the *axpy* operation will be executed and a view type for data residing in its preferred memory space:

```
    // define the execution space and accessible view type
    using exe_t = Kokkos::DefaultExecutionSpace;
    using view_t = Kokkos::View<value_t*, typename exe_t::memory_space>;
```

Before we execute the parallel algorithm, we must ensure that the data resides in an accessible memory space. Recall that Kokkos Core may be configured such that `DefaultExecutionSpace` refers either to a host backend or a device backend. In the former case, no further memory operations are necessary, since the `x_host` and `y_host` views created above are accessible from host backends. In the latter case, however, it is necessary to allocate separate arrays in the memory space of the default execution space and perform a deep copy of the data from the host views. Both of these cases can be handled using the `Kokkos::create_mirror_view_and_copy` function:

```
    // mirror the views in the memory space of exe_t
    view_t x = Kokkos::create_mirror_view_and_copy(exe_t(), x_host);
    view_t y = Kokkos::create_mirror_view_and_copy(exe_t(), y_host);
```

When the data is prepared, the execution policy can be defined and the *axpy* kernel can be executed:

```
    // execute the kernel on the device
    using RangePolicy = Kokkos::RangePolicy<exe_t>;
    Kokkos::parallel_for(
        "axpy",                         // name of the kernel
        RangePolicy(0, size),           // iteration range
        KOKKOS_LAMBDA (index_t i) {     // kernel function
            y[i] = alpha * x[i] + y[i];
        }
    );
```

Finally, we must remember to copy the result back to the host memory in case the execution happened in the device space:

```
    // copy the y array from device memory to host memory
    Kokkos::deep_copy(y_host, y);
}
```

Note that the `Kokkos::deep_copy` function has no effect when the `y_host` and `y` views are aliases to the same array in the host memory space.

1.3 Template Numerical Library

In this section, we describe Template Numerical Library, an open-source project that is developed mainly by the author and his supervisor. The content is based on the paper [A142] with updates due to recent changes in the library. The text provides a high-level view on the project, detailed description of the features and usage examples can be found in its documentation [O31].

1.3.1 Introduction

Template Numerical Library [A142] is a project that aims to simplify the development of high-performance numerical solvers and methods, especially in the field of computational fluid dynamics. It is an open-source project developed in C++ and relies on modern features of the language, currently the C++17 standard. In order to simplify installation and inclusion in other projects, TNL is designed as a header-only library.

The name refers to C++ template meta-programming techniques that are used throughout the library and often preferred over corresponding run-time approaches involving virtual functions that have limited support in the CUDA framework [M19]. In general, this approach also leads to a more

efficient code [B12, O24] at the cost of increased work at compile-time. Moreover, it encourages detection of programming errors at compile-time. However, considering the development of modern C++ standards, using the word "template" as part of the name does not differentiate it from other libraries, since any successful C++ library must use templates.

Of the high-level libraries described in the previous section, Kokkos is the most general parallel programming and performance portability library. TNL tries to be the next one and achieves it by targeting a broader scope of problems. TNL is not only a parallel programming library, but provides also algorithms and data structures for dense and sparse linear algebra, structured grids and unstructured meshes, and other building blocks for advanced numerical solvers. Furthermore, TNL has wrappers to simplify using low-level MPI functions in C++ and provides distributed data structures with synchronizers based on MPI.

TNL does not follow a typical design for heterogeneous platforms where most computations are performed by the CPU and specific tasks are offloaded to an accelerator, or where the work is divided between accelerators and the CPU. Instead, TNL is designed for numerical solvers where most computations are performed either entirely on CPU or a GPU accelerator. This is achieved by a `Device` template parameter that allows to switch between computations on CPU, GPU, and potentially other platforms. Of course, it is still necessary to rely on CPU for sequential tasks such as initialization or data output, but the majority of time-consuming computations can be parametrized by `Device`. This approach requires all data used in the computations to be allocated in the memory of the accelerator in order to minimize data movement between the system memory and accelerators, otherwise computations would be limited by communication over the PCIe bus. Data structures implemented in TNL provide an easy way to manage data allocated in different address spaces.

Finally, an important aspect of the TNL design is that it simplifies the development of portable parallel algorithms, a goal that is shared with other high-level libraries such as Thrust or Kokkos. This means that algorithms or whole numerical solvers can be developed and debugged on CPUs, for which many development tools are available, and then it can be switched to a different platform by changing a single parameter. Based on our experience, many algorithms developed this way work on GPUs with no or minor modifications due to different behavior of the parallel platforms.

1.3.2 Parallel programming components

TNL is based on a backend system for modern parallel architectures such as multi-core CPUs and GPU accelerators, resulting in a high-level interface providing a simple and portable way of programming these platforms. The current backends are based on OpenMP [A51] for CPU parallelization and CUDA [M19] for GPU parallelization. Adding support for backends based on HIP [M1] and SYCL [M11] is planned for the future.

Similar to Kokkos, the parallel programming interface in TNL is based on abstractions for various memory and execution operations. The concept of *allocators* takes care of memory allocation, which is the fundamental memory operation and corresponds roughly to the *memory space* concept in Kokkos. TNL provides the following allocators in the `TNL::Allocators` namespace with an interface compatible with STL: `Host` (alias for `std::allocator`), `Cuda` (allocations in the global memory of a GPU using `cudaMalloc`), `CudaHost` (allocations in the host memory using `cudaMallocHost`), and `CudaManaged` (allocations in the unified memory space using `cudaMallocManaged`). Next, TNL provides interface for additional memory operations such as data copies (including source and target with mismatched address spaces), comparison, construction and destruction of elements (objects) in allocated memory.

The main abstraction for code execution in TNL is realized by the `Device` template parameter of various functions and classes. A *device* determines where and how the code is executed. TNL currently provides the following devices in the `TNL::Devices` namespace: `Sequential` (sequential execution either on CPU or within a GPU thread), `Host` (parallel execution on CPU via OpenMP), and `Cuda` (parallel

execution on a GPU). Note that the device concept is not fixed and the implementation of additional backends may require significant refactoring or even introduction of more refined concepts.

Furthermore, TNL provides fundamental parallel algorithms for common operations: `parallelFor`, `reduce`, `scan`, and `sort`. These correspond to the same functionality present in other high-level libraries and some parallel programming frameworks, such as Kokkos, SYCL, and TBB. The algorithms in TNL use indices and lambda expressions for data access, not iterators like Thrust and STL. Some algorithms like `parallelFor` expose an interface for fine-tuning execution details, such as specifying a CUDA stream or dynamic shared memory size for CUDA kernels. The parameters are specified via an instance of the `LaunchConfiguration` structure, which roughly corresponds to the *execution policy* concept in Kokkos.

Finally, TNL provides components for distributed computing based on MPI [M15]. These are based on low-level wrappers for MPI functions with C interface, which simplify their usage in C++ and provide basic portability layer (i.e., the wrappers provide common behavior for builds without MPI, but switching to a different distributed computing framework is not supported). Higher-level utilities are provided for common operations, e.g., `reduce` function template for scalar values, `send` and `recv` function templates for the `Array` data structure, etc. On the highest level, distributed data structures are implemented in order to provide an object-oriented view on distributed computing.

1.3.3 Data structures

TNL provides many common dynamic data structures, such as `Array` and `Vector`, `DenseMatrix` and `SparseMatrix`, or `NDArray` (see Section 2.1 for details). Additional variants are implemented for each of the aforementioned data structures:

- *Views*: Each dynamic data structure has an associated view data structure, which provides an interface with non-owning and shallow-copy semantics. As a consequence, views cannot be resized, can be rebound to other data (including external data structures), can be passed by value to device functions such as CUDA kernels, and can be captured by value in lambda expressions. Hence, they simplify data access in code for GPU accelerators.
- *Distributed data structures*: For each aforementioned data structure there is a distributed variant (e.g., `DistributedArray` etc.) that implements an interface for object-oriented distributed computing via MPI. Each distributed data structure uses the original dynamic data structure to manage the local data and provides access to global and local views of the data.

Besides dynamic data structures, TNL provides `StaticArray`, `StaticVector` and `StaticNDArray` that manage data in fixed-size objects created in the *stack segment* of memory. They are usable in device code and even constant expressions (e.g., `constexpr` functions).

Additionally, TNL provides advanced data structures specific to numerical applications, such as grids and meshes. The data structure for unstructured meshes is described in detail in Section 2.2. Similarly to the *memory layout* concept in Kokkos, most of the TNL data structures can be configured with a template parameter to adjust the layout of the data in memory (e.g., row-major or column-major matrices). TNL currently does not have an analogy to the *memory traits* concept in Kokkos.

1.3.4 Algorithms

TNL provides high-level functions for common operations based on the fundamental parallel algorithms, for example, `compare`, `contains`, `containsOnlyValue`, or vector operations based on expression templates [C30, A175]. More advanced algorithms include sparse matrix–vector multiplication and other matrix operations, iterative solvers and preconditioners for systems of linear algebraic equations with details described in Section 3.3.3, and solvers for systems of ordinary differential equations. Note that the list is not exhaustive and other algorithms may be implemented in the future.

Finally, we show an example implementing the *axpy* operation in TNL. As in the preceding sections, the input data is allocated in the host memory. A parallel version for the `Host` device can be implemented using the `parallelFor` function and a simple lambda expression as follows:

```
void host_axpy(index_t size, value_t alpha, value_t* x, value_t* y)
{
    // define the device type and execute the kernel on the host
    using device_t = TNL::Devices::Host;
    TNL::Algorithms::parallelFor<device_t>(
        0, size,            // iteration range
        [=] (index_t i) {   // kernel function
            y[i] = alpha * x[i] + y[i];
        }
    );
}
```

Alternatively, expression templates implemented in TNL can be used to evaluate the vector operation. This leads to even clearer code as the meta-code associated with the `parallelFor` function is avoided. We only need to create a `VectorView` object for each vector in the expression:

```
void host_axpy(index_t size, value_t alpha, value_t* x_raw, value_t* y_raw)
{
    // create host views for the input arrays
    using host_view = TNL::Containers::VectorView<value_t, TNL::Devices::Host>;
    host_view x(x_raw, size);
    host_view y(y_raw, size);

    // compute the vector expression
    y = alpha * x + y;
}
```

Both variants of the `host_axpy` function can be adapted for computations on a GPU accelerator. First, we create `ArrayView` objects associated to the `Host` device for the input arrays, then allocate `Array` objects associated to the `Cuda` device, and copy the data from the host memory to the device memory:

```
void device_axpy(index_t size, value_t alpha, value_t* x_raw, value_t* y_raw)
{
    // define the device type
    using device_t = TNL::Devices::Cuda;

    // create host views for the input arrays
    using host_view = TNL::Containers::ArrayView<value_t, TNL::Devices::Host>;
    host_view x_host(x_raw, size);
    host_view y_host(y_raw, size);

    // allocate arrays in the device memory and initialize them with the host data
    using array_t = TNL::Containers::Array<value_t, device_t>;
    array_t x_device, y_device;
    x_device = x_host;
    y_device = y_host;
```

In order to access the data in `x_device` and `y_device` from a lambda expression in the `parallelFor` function, we must create views for the data, which can be captured by value in the lambda expression. Note that the lambda expression must be marked with the `__cuda_callable__` macro, which is an annotation for functions that should be executable by the CPU as well as GPU. Additionally, the lambda expression must have the `mutable` specifier so that the captured variables are modifiable:

```
    // obtain views accessible on the device
    auto x = x_device.getView();
    auto y = y_device.getView();

    // execute the kernel on the device
    TNL::Algorithms::parallelFor<device_t>(
        0, size,                                        // iteration range
        [=] __cuda_callable__ (index_t i) mutable {     // kernel function
            y[i] = alpha * x[i] + y[i];
        }
    );
```

Finally, the result must be copied back to the host memory:

```
    // copy the y array from device memory to host memory
    y_host = y_device;
}
```

Programming the variant using expression templates on GPU is left as an exercise to the reader.

The functions `host_axpy` and `device_axpy` can be combined in a general `parallel_axpy` function as follows. The `__CUDACC__` macro is defined by the `nvcc` and `clang++` compilers if and only if a CUDA source code is being compiled. Hence, the code might be used in common header files for both C++ and CUDA source files.

```
void parallel_axpy(index_t size, value_t alpha, value_t* x, value_t* y)
{
#ifdef __CUDACC__
    device_axpy(size, alpha, x, y);
#else
    host_axpy(size, alpha, x, y);
#endif
}
```

1.3.5 Future work

As is evident from the preceding description, TNL is an actively developed project with many features in development. The most fundamental features planned for the future are backends based on modern frameworks such as HIP [M1] and SYCL [M11]. Their development might also require refactoring of existing facilities, namely the concept behind the `Device` template parameter should be revisited with respect to the concepts of *execution policies* that are used in other high-level libraries such as Kokkos [A170, A171], Thrust/STL [O26], or Ginkgo [A6]. Furthermore, aspects such as documentation, unit tests, and benchmarks are continuously being improved.

Finally, the transition from a monolithic package to a more modular structure is being considered for the TNL project. Introducing modules would allow clear separation of responsibilities for each part of the project, clear definition of dependencies on external projects and between the modules, and better overview of their completion status. Consequently, TNL developers would have to track the stability of interfaces more carefully, for example using versioned releases that would also improve the usability of TNL in downstream projects. This progress already started when the Python bindings for TNL were moved to a separate repository named PyTNL [O30]. Candidates for new modules are features related to image processing and advanced numerical methods such as FEM, MHFEM, or FVM.

Chapter 2

Data Structures

Efficient data structures play an important role in high-performance computing, because they affect where each data is stored in the computer memory and how quickly it can be accessed. Hence, data structures and algorithms often have to be designed collectively in order to utilize the most efficient access pattern on given hardware architecture.

In this chapter, we present several data structures implemented in the Template Numerical Library that was introduced in Section 1.3, which are flexible in the sense that they can be configured for a specific algorithm or hardware architecture.

2.1 Multidimensional arrays

Many algorithms in scientific computing work with coefficients indexed by three, four or even more indices. Multidimensional arrays are therefore a natural data structure for organizing and storing such values in the computer memory. Since the C++ language before its C++23 standard supports only one-dimensional arrays natively, multidimensional arrays have to be implemented explicitly (e.g., in a library) based on a mapping of multidimensional data to an internal one-dimensional array.

An interesting problem is how to choose the mapping from the multidimensional index space into the one-dimensional index space. Even for two-dimensional arrays (i.e., matrices) it can be argued whether they should be stored in the *row-major* format, where rows are stored as 1D arrays, or in the *column-major* format, where columns are stored as 1D arrays. The optimal choice depends on the operations that we want to do with the data, as well as on the hardware architecture that will process the data. For example, the row-major format is suitable for algorithms processing a matrix row by row on the CPU, while for GPU algorithms also processing a matrix row by row the column-major format is more appropriate. For three- and more-dimensional arrays there are even more combinations of possible array orderings.

For these reasons, we developed a data structure which allows to configure the indexing of multi-dimensional data and thus optimize the data structure for given algorithm and hardware architecture. The data structure was first proposed in [B20] and its implementation based on the C++14 standard was later integrated into the TNL library and further developed therein. The TNL project's documentation [O31] provides an overview of the public interface and examples showing how the data structure can be used. In the following subsections, we describe the *distributed* version of the data structure, which is especially useful for the algorithms described in Chapters 4 and 5.

2.1.1 Distributed multidimensional array

In the *distributed* configuration, a global multidimensional array is decomposed into several subarrays and each MPI rank typically stores the data associated to one subarray. Since a multidimensional

Figure 2.1: Typical two-dimensional decomposition of a two-dimensional array into 9 subarrays assigned to ranks 0-8.

array stores structured data, we will consider only *structured conforming* decompositions, where the array is split by hyperplanes perpendicular to one of the axes. Figure 2.1 shows a typical two-dimensional decomposition of a two-dimensional array into 9 subarrays. Note that the implementation also allows multiple blocks to be assigned to the same rank, which can be seen as a generalization of the requirement for conforming decompositions. This feature is useful for optimizing the decomposition under various constraints, which will be described later in the following chapters.

In order to represent a distributed multidimensional array, each MPI rank needs to have the following variables:

- `localArray` – an instance of a multidimensional array which contains data and indexing attributes of the subarray assigned to the rank.
- `communicator` – the MPI communicator comprising all ranks that have a subarray of the global multidimensional array.
- `globalSizes` – a multi-index determining the sizes of the global multidimensional array in terms of the number of elements in each dimension.
- `localBegins` and `localEnds` – two multi-indices determining the beginning and ending position of the local subarray in the global array. According to the convention used in TNL, the local subarray spans the multidimensional interval [`localBegins`, `localEnds`).

Although these attributes of a distributed multidimensional array can be created separately and managed manually, TNL provides a convenient data structure `TNL::Containers::DistributedNDArray` that provides a high-level interface to manage these attributes. The TNL project's documentation [O31] provides an overview of the public interface and examples showing how the data structure can be used.

2.1.2 Operations

Algorithms involving a distributed multidimensional array may perform different operations on the data structure. Common operations may be classified as follows:

- *Local operations* are the simplest type of operations that can be performed on each MPI rank independently of the other ranks. An example is the `setValue` member function of `DistributedNDArray` which sets all elements of the array to a constant value.

- *Collective operations* involve some kind of communication among all ranks in the MPI communicator. An example is the `operator==` function for the comparison between two instances of `DistributedNDArray`, which involves a local operation (comparison of two local arrays) followed by a collective `AND`-reduction of the local results among all ranks.

- *Partially-collective operations* involve communication not on the global communicator, but among smaller groups of MPI ranks. It is often convenient to create MPI sub-communicators using the `MPI_Comm_split` function (see Section 1.1.8) to simplify the communication pattern definition in the program. For example, computing the sums of array elements per row involves the communication only among ranks containing the same rows. Another example are stencil computations on regular grids, which typically involve data exchange among neighboring ranks.

Additionally, complex algorithms may involve data exchange prior to performing the local operation (such as the input vector re-distribution in the distributed matrix–vector multiplication). General description of such algorithms is out of scope of this work. In the following subsection, we focus on data exchange in stencil computations, which are common in numerical analysis.

2.1.3 Overlaps and data synchronization

Stencils in numerical analysis are geometric arrangements of nodes on a structured grid that express which data is accessed from a grid node of interest and used in the numerical computation. The application of stencils on the data stored in a non-distributed multidimensional array is straightforward as all data is readily available in the memory of one rank. However, in a distributed configuration, the multidimensional array needs to be extended with *overlaps* to facilitate access to data owned by neighboring ranks, and *data synchronization* to exchange data in the overlapping regions of the array when the stencil is applied iteratively.

Figure 2.2 shows a typical two-dimensional decomposition of a two-dimensional array into 9 subarrays with overlaps highlighted using dotted lines. The width of the overlaps in the TNL implementation is configurable separately for each dimension of the array. I.e., the overlaps can be described by a multi-index `overlaps` where each component is non-negative and determines the number of data elements along the corresponding axis shared between neighboring ranks. Each rank includes the overlapping regions in its local array which is allocated larger, i.e., it spans the multidimensional interval $[\texttt{localBegins} - \texttt{overlaps}, \texttt{localEnds} + \texttt{overlaps})$. Note that negative indices may be used to access the local data in the overlapping region below `localBegins` and they are mapped properly to the one-dimensional storage index. Also note that the array always includes the so called *periodic overlaps* in the allocation, i.e., overlapping regions around the whole array. This allows to easily implement the periodic boundary conditions in numerical schemes as it ensures that the stencil can be applied on all local nodes.

The data synchronization for the overlaps in distributed multidimensional arrays in TNL is implemented in the class `TNL::Containers::DistributedNDArraySynchronizer`. Before the algorithm can be started, each MPI rank must configure the synchronizer:

- The rank IDs of the *neighbors* relevant for the stencil must be set. In the example shown in Figure 2.2, rank 4 would set ranks 1, 3, 5, and 7 as its neighbors for the five-point stencil, and ranks 0, 1, 2, 3, 5, 6, 7, and 8 for the nine-point stencil. Note that for simplicity, the rank numbering in Figure 2.2 is structured, but the synchronizer supports an arbitrary unstructured numbering.

- The *directions* in which the data can be transferred must be set depending on the stencil. In the simplest case, all values of the array can be transferred in all directions, but some applications (such as the lattice Boltzmann method described in Chapter 5) may use separate arrays for different synchronization directions (e.g., left-to-right or right-to-left).

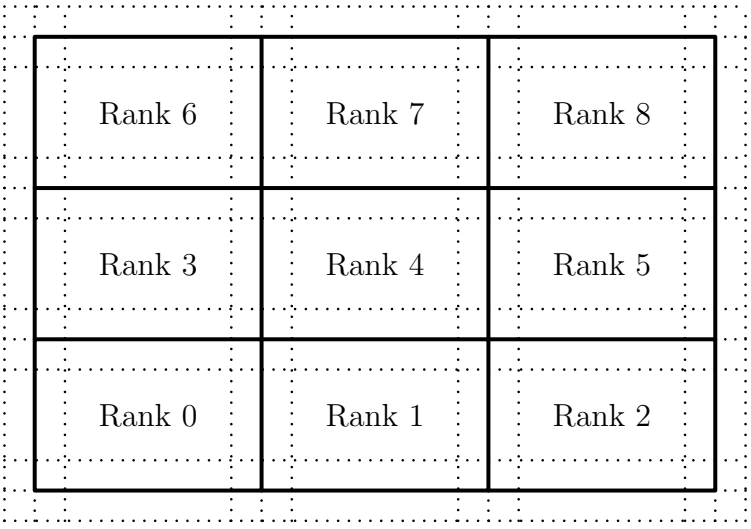


Figure 2.2: Typical two-dimensional decomposition of a two-dimensional array into 9 subarrays with overlaps between neighbors (highlighted using dotted lines).

- The *tags* for MPI messages can be set to avoid conflicts when multiple arrays distributed among the same ranks are synchronized at the same time.

The synchronization procedure consists of the steps summarized in Algorithm 1. Note that some steps in the algorithm may sometimes be unnecessary. For example, the allocated buffers can be reused when the synchronizer is used repeatedly on arrays of the same size.

Algorithm 1 (Distributed multidimensional array synchronization)

1. Allocate all send and receive buffers.
2. Copy data from the local array to the send buffers.
3. Start MPI communications with all neighbors via `MPI_Isend` and `MPI_Irecv`.
4. Return a vector of MPI requests (the non-blocking procedure is suspended after this step).
5. Wait for all MPI requests to complete.
6. Copy data from the receive buffers to the local array.

The synchronization algorithm can be executed in several modes. The asynchronous or non-blocking mode allows to interleave the synchronization with some other, unrelated work by suspending the execution when all non-blocking MPI requests have been created and deferring the remaining steps for later. Assuming that the MPI implementation can proceed with the communication in the background, this approach can greatly improve the efficiency of the distributed algorithm. When multiple arrays are to be synchronized at the same time among the ranks, it is also desirable to interleave the individual steps of Algorithm 1 via *pipelining* for all arrays.

In general, to avoid a large number of small MPI requests, the data to be sent must be copied into a contiguous buffer and the received data must be copied from a contiguous buffer into the local array. On the other hand, if the data to be sent or received is already stored in a contiguous block of the array, these copies are useless and therefore should be omitted. Such cases are automatically detected in the `DistributedNDArraySynchronizer` class and buffers are replaced with views into the local array itself. Hence, users may configure the layout of the array appropriately for the synchronization directions in their application in order to improve the efficiency of the synchronization algorithm.

2.2 Unstructured meshes

The content of this section is based on the paper [A110] that presents a data structure for *conforming unstructured homogeneous* meshes implemented in the TNL library that was introduced in Section 1.3. The initial implementation was designed by Vítězslav Žabka and later it was generalized and improved by the author. Since the publication of the paper in the journal, we have extended the data structure for *polygonal* and *polyhedral* meshes. The implementation of this extension in TNL was conducted by Ján Bobot as part of his Master's Degree project [B5] under the supervision of the author. The extensions for polygonal and polyhedral meshes are incorporated into the text in this section.

The section is organized as follows. First, we introduce the topic of unstructured meshes and the context of our work in Section 2.2.1 and review the basic terminology and mesh representations in Sections 2.2.2 and 2.2.3. Then, in Section 2.2.4, we present the main ideas behind the implementation of an efficient data structure for the representation of conforming unstructured meshes on CPU and GPU-based systems. In Section 2.2.5, we describe the implementation details and Section 2.2.6 is dedicated to the extension of the primary data structure for distributed meshes. Results of the benchmark problems are presented in Sections 2.2.7 to 2.2.10.

2.2.1 Introduction

Meshes are fundamental data structures for many numerical methods applied to problems in natural sciences, engineering, and other fields. In order to approximate problems with complex geometries with sufficient accuracy, unstructured meshes often have to be considered instead of structured meshes. Hence, an efficient way to generate, represent and manipulate an unstructured mesh in computer memory is needed in order to facilitate highly performant computations of large problems. There are many software tools and libraries which provide data structures and interfaces to represent and manipulate unstructured meshes for scientific computations. Firstly, mesh generators such as CUBIT [M5], Gmsh [A85], or MMG [A53], produce the resulting mesh from their internal representation in some specific file format. On the other end of common scientific simulations workflow, post-processing and visualization libraries, such as VTK [B27] or VCGlib [O5], interpret the data associated with a mesh, which may be stored in different data formats than initially generated. The conversion between various mesh file formats can be performed by many tools, including ParaView [B2], VisIt [B11], and `meshio` [O28]. The widest range of approaches to the unstructured mesh representation is needed for numerical simulation solvers that stand between mesh generators and visualization toolkits. Several open-source C++ libraries, including OpenMesh [C12], ViennaGrid [M25], PUMI [A101], DUNE [A18, A19], and MOAB [M28], provide generic data structures for the representation of unstructured meshes. The latter two libraries are especially notable for following or defining a very general specification that can be used as an interface between multiple compatible applications. Numerical simulation frameworks, such as deal.II [A17], OpenFoam [C26], or libMesh [A109], provide data structures integrated with other parts of the respective library.

Although the aforementioned libraries are very general in the sense that they facilitate code reuse and are not bound to a specific application, each library has its distinct feature set. For example, some libraries focus on adaptive mesh refinement of conforming simplex meshes, while others focus on the handling of non-conforming and hierarchical meshes. From the performance point of view, the development has focused on parallel computations on distributed memory systems and scaling to thousands of nodes that are available on current supercomputers. Another important feature is the support for efficient computations on the GPU accelerators which have already been used successfully for numerical simulations on unstructured meshes [C9, A90, A148, A163, C42]. GMlib [O20] provides unstructured mesh data structures for programmers using the OpenCL language, but in general, native programming languages such as CUDA for NVIDIA GPUs are usually preferred in high-performance computing due to higher flexibility and performance [C9]. However, most work has focused on the

algorithmic aspects of GPU computing rather than providing a multi-purpose data structure and the code of these works is not easily reusable, either because it is not open-source, or because it is too tightly coupled with a specific application. Another problem is that extending existing software for GPUs while preserving efficiency is practically impossible due to different memory layout requirements.

Hence, we believe that the mesh representation has to be revisited with respect to the hardware-specific memory layout in order to design an efficient data structure for scientific computations on unstructured meshes. In this section, we pursue the following goals:

1. to design a general multi-purpose data structure for representing conforming unstructured meshes that can be used efficiently on a wide range of hardware architectures, including traditional multi-core CPUs, GPUs, and distributed CPU or GPU-based clusters,

2. to provide a flexible, comprehensible and reusable implementation of the data structure as an open-source software, and

3. to demonstrate its efficiency on GPUs compared to sequential or parallel CPU implementations.

The meshes considered in [A110] are unstructured conforming homogeneous meshes as specified in Definitions 1 to 3 in Section 2.2.2, which is a sufficiently general class for many applications. Additionally, this thesis includes modifications of the data structure necessary to support a larger class of conforming polygonal and polyhedral meshes (which are not homogeneous). We also describe an extension of the base data structure for *distributed* meshes using the domain decomposition approach. On the other hand, as a first step towards an efficient implementation, we disregard some features, such as adaptive refinement, which might be challenging to implement efficiently on GPUs.

The proposed data structure can represent essentially arbitrary *shapes* or *topologies* of mesh entities, including a general polygonal or polyhedral entity. Each topology is represented by its own type in the C++ language. The library currently features definitions of the following topologies: vertex, edge, triangle, quadrangle, tetrahedron, hexahedron, arbitrarily dimensional simplex, pyramid, wedge, general polygon and general polyhedron. Furthermore, the data structure is highly configurable via templates of the C++ language, allowing to avoid the storage of unnecessary dynamic data. Similarly to [C42], the internal memory layout is based on state–of–the–art sparse matrix formats [C31, C32, A143] for the storage of incidence matrices. This approach reduces complexity of the implementation thanks to code reuse and allows us to select a sparse matrix format that is optimized for given hardware architecture.

The efficiency of the implemented data structure is verified using several benchmark problems. We developed two benchmark problems based on simple parallel algorithms to evaluate different configurations of the data structure, to compare it with MOAB [M28], and to demonstrate the efficiency of CUDA-based GPU parallelization compared to OpenMP-based CPU parallelization. Additionally, the data structure has been successfully used as a fundamental building block in advanced numerical solvers based on the method described in Chapter 4.

2.2.2 Terminology

In this section, we review the basic concepts related to unstructured meshes which will be used throughout this thesis. For a mathematically rigorous definition of a mesh see, e.g., [A19] and the references therein.

A *mesh* is a collection of geometric objects with a simple shape called *mesh entities*. For example, a D-dimensional mesh can be a partitioning of a region in $\mathbb{R}^D$ or a D-manifold in $\mathbb{R}^N$, such as the partitioning of a polyhedral region into a set of tetrahedra which are composed of triangles, edges and vertices. Non-vertex mesh entities are composed of other mesh entities which have lower dimension. The lower-dimensional entities that frame an entity of higher dimension are called *subentities* of the

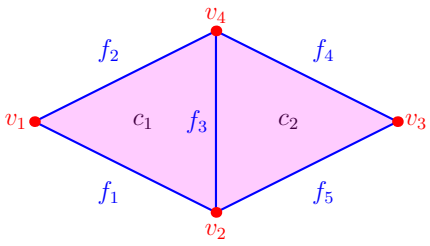


Figure 2.3: An example of a mesh consisting of cells c_1, c_2, faces f_1, f_2, f_3, f_4, f_5, and vertices v_1, v_2, v_3, v_4.

entity and the higher-dimensional entity is called a *superentity* of the comprising entities. For example, a triangle is composed of three vertices which are the 0-dimensional subentities of the triangle and the triangle is a 2-dimensional superentity of these vertices. Note that in this description, vertices are the only entities which hold their spatial coordinates in $\mathbb{R}^N$ and other entities are realized by means of their subvertices. This is advantageous for working with *dynamic meshes*, where the spatial coordinates may change during the computation.

When we denote a mesh as $\mathcal{M}$ in the abstract language, it should be understood that $\mathcal{M}$ is a set of all mesh entities which form the partitioning and all of their subentities, but concrete representations of the mesh (e.g., in the computer memory or in a data file) are allowed to omit certain non-essential entities. The D-dimensional entities of a D-dimensional mesh are called *cells*, the $(D-1)$-dimensional entities are called *faces*, the 1-dimensional entities are called *edges* and the 0-dimensional entities are called vertices.

Mathematically, the relations between mesh entities of different dimensions can be represented by an *incidence matrix*. Let $\mathcal{E}_{d_1} = \{E_1, \ldots, E_m\}$ and $\mathcal{E}_{d_2} = \{F_1, \ldots, F_n\}$ denote the sets of d_1-dimensional and d_2-dimensional mesh entities, respectively. The entities E_i and F_j are called *incident*, if E_i is a subentity of F_j (if $d_1 < d_2$) or E_i is a superentity of F_j (if $d_1 > d_2$). The incidence matrix I_{d_1,d_2} of classes $\mathcal{E}_{d_1}$ and $\mathcal{E}_{d_2}$ is an $m \times n$ binary matrix such, that $[I_{d_1,d_2}]_{i,j} = 1$ if and only if E_i and F_j are incident. For example, a simple mesh consisting of two triangles is shown in Figure 2.3 and its incidence matrices are shown in Figure 2.4. It can be observed that $I_{d_2,d_1} = I_{d_1,d_2}^T$.

Additional connectivity properties can be defined for unstructured meshes. For example, the dual graph $\mathcal{D}$ can be used to describe the adjacency of cells in a mesh $\mathcal{M}$. The vertices of $\mathcal{D}$ correspond to the mesh cells $\mathcal{E}_D \subset \mathcal{M}$ and edges of the graph $\mathcal{D}$ represent links between adjacent (or neighboring) mesh cells. An example of a small mesh and its dual graph is shown in Figure 2.5. The dual graph is usually constructed such that two D-dimensional cells are adjacent, if and only if they have a common $(D-1)$-dimensional face, i.e., they share at least D different subvertices. A more general definition that can be used, e.g., in the METIS library [A107], is based on specifying the minimum number of common subvertices n_{common}, which two cells need to share to be considered adjacent. For example, specifying $n_{common} = 1$ leads to a dual graph where two cells are adjacent if they share at least one subvertex.

The dual graph has many interesting properties. For example, it can be shown that elements of the matrix $S_D = I_{D,0}I_{0,D}$ are equal to the counts of common subvertices shared by each two cells. Then, the binary matrix A_D defined such that $[A_D]_{i,j} = 1$ if and only if $i \neq j$ and $[S_D]_{i,j} \geq n_{common}$ is the adjacency matrix of the dual graph with the parameter n_{common}. See Figure 2.5 and also [C42, Appendix A], for a similar example.

The next three definitions specify the type of meshes that will be dealt with in the following sections of this thesis.

Definition 1 (conforming mesh)**.** *Let $\mathcal{M}$ be a mesh. If for all mesh entities $E_1, E_2 \in \mathcal{M}$ the intersection of their closures $\overline{E}_1 \cap \overline{E}_2$ is either an empty set or a mesh entity, then $\mathcal{M}$ is called a conforming mesh.*

$$I_{0,1} = \begin{array}{c|ccccc} & f_1 & f_2 & f_3 & f_4 & f_5 \\ \hline v_1 & 1 & 1 & & & \\ v_2 & 1 & & 1 & & 1 \\ v_3 & & & & 1 & 1 \\ v_4 & & 1 & 1 & 1 & \end{array}$$

$$I_{1,0} = \begin{array}{c|cccc} & v_1 & v_2 & v_3 & v_4 \\ \hline f_1 & 1 & 1 & & \\ f_2 & 1 & & & 1 \\ f_3 & & 1 & & 1 \\ f_4 & & & 1 & 1 \\ f_5 & & 1 & 1 & \end{array}$$

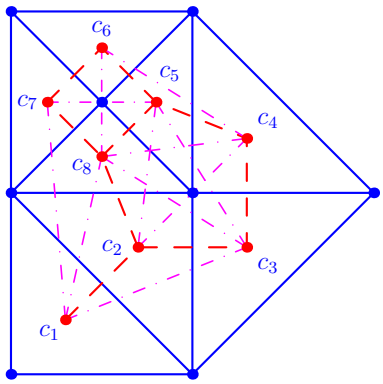
$$I_{0,2} = \begin{array}{c|cc} & c_1 & c_2 \\ \hline v_1 & 1 & \\ v_2 & 1 & 1 \\ v_3 & & 1 \\ v_4 & 1 & 1 \end{array}$$

$$I_{2,0} = \begin{array}{c|cccc} & v_1 & v_2 & v_3 & v_4 \\ \hline c_1 & 1 & 1 & & 1 \\ c_2 & & 1 & 1 & 1 \end{array}$$

$$I_{1,2} = \begin{array}{c|cc} & c_1 & c_2 \\ \hline f_1 & 1 & \\ f_2 & 1 & \\ f_3 & 1 & 1 \\ f_4 & & 1 \\ f_5 & & 1 \end{array}$$

$$I_{2,1} = \begin{array}{c|ccccc} & f_1 & f_2 & f_3 & f_4 & f_5 \\ \hline c_1 & 1 & 1 & 1 & & \\ c_2 & & & 1 & 1 & 1 \end{array}$$

Figure 2.4: Incidence matrices representing the relationships between entities in the mesh shown in Figure 2.3.



$$S_2 = \begin{array}{c|cccccccc} & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 \\ \hline c_1 & 3 & 2 & 1 & 0 & 0 & 0 & 1 & 1 \\ c_2 & 2 & 3 & 2 & 1 & 1 & 0 & 1 & 2 \\ c_3 & 1 & 2 & 3 & 2 & 1 & 0 & 0 & 1 \\ c_4 & 0 & 1 & 2 & 3 & 2 & 1 & 0 & 1 \\ c_5 & 0 & 1 & 1 & 2 & 3 & 2 & 1 & 2 \\ c_6 & 0 & 0 & 0 & 1 & 2 & 3 & 2 & 1 \\ c_7 & 1 & 1 & 0 & 0 & 1 & 2 & 3 & 2 \\ c_8 & 1 & 2 & 1 & 1 & 2 & 1 & 2 & 3 \end{array}$$

Figure 2.5: An example of a 2-dimensional mesh (blue vertices and edges), its dual graph with the parameter $n_{common} = 2$ (red vertices, red dashed edges), and its dual graph with the parameter $n_{common} = 1$ (red vertices, red dashed and pink dot-dashed edges). The matrix $S_2 = I_{2,0}I_{0,2}$ shows the counts of shared subvertices between each pair of the cells $c_1, \ldots, c_8$.

Definition 1 implies that a conforming mesh does not contain so called *hanging entities* which would be subentities of only one of the neighboring cells.

Definition 2 (unstructured mesh). *A mesh is called an unstructured mesh, if each vertex of the mesh can be a vertex of non-constant number of cells.*

Definition 2 implies that the number of superentities of an entity in an unstructured mesh (i.e., the number of neighboring entities of a higher dimension than the entity itself) is not a priori constant. This is in contrast with the number of subentities of an entity which depends only on the shape of the entity and not on the neighborhood of the entity.

Definition 3 (homogeneous mesh)**.** *If all cells of a mesh have the same shape, then the mesh is said to be* weakly homogeneous. *If all mesh entities of the same dimension have the same shape, then the mesh is said to be* (strongly) homogeneous.

Definitions 1 to 3 determine the class of meshes considered in the paper [A110]. In this section, we extend the work to cover general polygonal, weakly homogeneous, and general polyhedral meshes.

2.2.3 Mesh representations

First, note that a weakly homogeneous mesh is not automatically strongly homogeneous. For example, elements such as wedge and pyramid have both quadrilateral and triangular faces. However, wedge and pyramidal meshes can be represented similarly to strongly homogeneous meshes by using a general mesh entity topology, polygon, for the representation of faces. This approach combines the use of a *dynamic topology*, where the number of subentities may vary for each polygon, with a *static topology* for cells which all have the same shape and constant numbers of all subentities.

A D-dimensional mesh composed of cells with static topologies can be fully specified by supplying a list of subvertices for each cell of the mesh. In case of a simplex mesh, the lists of subvertices can be unordered and the relations between cells and its subvertices can be fully represented by the incidence matrix $I_{D,0}$. In case of a non-simplex mesh, the lists of subvertices have to be ordered and thus the binary incidence matrix $I_{D,0}$ represents only the connectivity, but not topological information. The authors of [C42] define a non-binary *mesh matrix* which has the same pattern as the incidence matrix and the non-zero entries specify local indices of the corresponding subentities. However, the local ordering of subentities can be encoded in the representation of the corresponding binary incidence matrix, avoiding the need to explicitly store the non-zero entries. When a sparse matrix format is used for its representation, the non-zero entries in each row can be placed according to the *local ordering* rather than the usual ordering given by the global subentity indices. Such an *enhanced* incidence matrix $I_{D,0}$ fully represents the connectivity as well as topological information about the cells. The relations between other mesh entities can be deduced from the incidence matrix $I_{D,0}$ based on the topology of a specific cell.

In contrast, general polyhedral meshes (with $D = 3$) do not have cells with a static topology. We know that each cell is a polyhedron that may have any number of subvertices, but the topology does not imply which vertices are connected with edges. A common way to specify a polyhedron is by supplying the list of its enclosing polygonal faces, which are themselves specified via ordered lists of subvertices. In the framework described in this section, two incidence matrices are needed to represent a polyhedral mesh: $I_{3,2}$ to specify polyhedra in terms of their polygonal faces, and $I_{2,0}$ to specify polygonal faces in terms of their subvertices.

Note that each two-dimensional mesh consisting of cells with static topologies can be represented as a polygonal mesh, and each three-dimensional mesh consisting of cells with static topologies can be represented as a polyhedral mesh. The purpose of representing a homogeneous mesh using static entity topologies is to allow the selection of algorithms optimized for a specific topology. Static topologies allow to define conventions for local indexing of subentities, which can be used for algorithmic optimizations that are not possible for a general polygonal or polyhedral mesh. For 3D meshes, the use of static topologies, where possible, compared to the polyhedral representation is also more efficient in terms of memory requirements or file size.

2.2.4 Design considerations

The data structure for the representation of a conforming unstructured mesh is intended to be used in advanced numerical methods in high-performance computing applications and therefore it has to be as efficient as possible. Unfortunately, it is practically impossible to design a single efficient representation

suitable for all numerical schemes. Hence, compromises had to be made and this section summarizes our design goals and techniques we found to achieve them.

This section covers only the important ideas behind the implementation of the `Mesh` class template in the TNL library. The general design decisions related to parallel programming are described in Section 1.3. The full public interface as well as examples showing how to use the data structure can be found in the TNL project's documentation [O31].

Static configuration

In accord with the programming paradigm used in TNL, we aim to satisfy the high configurability requirement with *static configuration*, i.e. configuration which is supplied and resolved at compile-time. An obvious disadvantage is that if a run-time selection of the configuration is needed, it is restricted to a finite set of configurations which were explicitly instantiated at compile-time. The user configuration is supplied as a template parameter to the main class template `Mesh` which will be described in the next section.

The purpose of the configuration is to allow users to fine-tune the mesh representation for a specific purpose by selecting only the necessary mesh features. A prime example is the elimination of mesh entities and incidence matrices which are not accessed by the application. Another example is the selection of internal data types for storing coordinates and indices, which can be used to further reduce the memory requirements and consequently improve the efficiency of computation.

A complete example of the default configuration from which the user configuration can be easily derived is given in Appendix A. The configuration allows to change the following parameters:

- The shape of cells which implies the shape of all other mesh entities. It is denoted as `CellTopology` in Appendix A. Examples of built-in entity shapes are shown in Appendix B.
- Dimension $N \geq D$ of the space the mesh is embedded in. Note that N is denoted as `SpaceDimension` in Appendix A and D is the dimension of the cells (denoted as `meshDimension`).
- Coordinate data type (e.g. `float`, `double`), global and local index types (e.g. `int` and `short int`). The types are denoted as `RealType`, `GlobalIndexType` and `LocalIndexType` in Appendix A.
- Pairs of dimensions d_1, d_2, where $d_1 > d_2$, for which d_1-dimensional entities store the links to their d_2-dimensional subentities. Each entity stored in the mesh must store the links to its subvertices which provide their geometric realization. See the function `subentityStorage` in Appendix A.
- Pairs of dimensions d_1, d_2, where $d_1 < d_2$, for which d_1-dimensional entities store the links to their neighboring d_2-dimensional superentities. See the function `superentityStorage` in Appendix A.
- Dimensions of entities for which the interior/boundary tags are stored. See the function `boundaryTagsStorage` in Appendix A.
- Storage of the dual graph and its parameter n_{common}. See the function `dualGraphStorage` and the attribute `dualGraphMinCommonVertices` in Appendix A.

Internal data structures

An important design choice concerns the internal data structures used for the storage of non-temporary data. As explained in Section 2.2.2, the entire mesh topology can be expressed using incidence matrices which can be stored in (binary) sparse matrix formats. Similarly, the dual graph can be represented by its adjacency matrix A_D which can also be stored in a sparse matrix format. Additional data, such as vertex coordinates and entity tags, can be stored in plain arrays indexed by the appropriate mesh

entity indices. The use of arrays and suitable sparse matrix formats not only reduces complexity of the implementation, but also provides performance benefits compared to object-oriented representation of mesh entities, because random accesses to the data elements can be cached efficiently and the coalesced memory access pattern [M19] can be utilized on GPUs. On the other hand, an implementation of operations such as inserting new entities into an existing mesh representation would be complex and it could cause performance degradation due to reallocation of the arrays and sparse matrices. Such operations are therefore not supported.

Note that mesh entities themselves do not have to be stored explicitly in computer memory. Instead, they can be *instantiated* on demand based on the entity dimension and index, since all topological information as well as additional data such as vertex coordinates can be fetched from the global mesh object's internal data structures. In our implementation, `MeshEntity` is a generic class template for a d-dimensional mesh entity, which comprises just a pointer to the global mesh and an entity index, while the entity dimension d is encoded in the class template parameters. Various member functions supply the dimension and index of the entity when accessing global data structures via the mesh pointer.

The incidence matrices representing the connectivity between mesh entities are usually sparse, so we consider sparse matrix formats to organize the data in memory. Since the incidence matrix is a binary matrix, the conventional formats are modified by omitting the storage for the values of the matrix elements and storing only the column indices of non-zero elements. As explained in Section 2.2.3, the non-zero entries of the incidence matrix are ordered according to local indices rather than global indices.

A comprehensive evaluation of many sparse matrix storage formats has been recently presented in [A119]. We have considered the following commonly used formats for the incidence matrices used in the mesh:

1. The *Compressed Sparse Row* (CSR) format [C13, B26] is easy to implement and it does not have any overhead due to padding. The step size between two consecutive elements in the same row is always 1.

2. The *Ellpack* format [M12, B26] adds padding zeros at the end of all rows shorter than the longest row to obtain a two-dimensional array which can be stored either in the row-major or the column-major orientation. Hence, the step size between two consecutive elements in the same row is either 1 or m, where m is the number of matrix rows. The overhead due to padding can be considerable if there are a few very long rows. The row-major or column-major orientation can be chosen depending on the hardware architecture. For example, the column-major orientation is advantageous for GPUs due to higher memory throughput of the coalesced access pattern [M19] compared to the strided pattern implied by the row-major orientation.

3. The *Sliced Ellpack* format [C31, A143] is a block modification of the Ellpack format. The storage arrays are split into blocks containing a constant number of rows stored in the column-major (or row-major) orientation. This provides a compile-time constant step size between the consecutive elements in a row and the column-major variant still allows coalesced accesses assuming that the block size is a multiple of the CUDA warp size which is 32 on current hardware [M19]. Furthermore, it is not necessary to add padding entries to align with the globally longest rows, but the longest row in each block can be considered to reduce memory overhead.

The most important criteria for our implementation are storage overhead and the coalesced access pattern for GPUs. Therefore, we use sparse matrices based on the Sliced Ellpack format with row-major orientation for CPU computations and column-major orientation for GPU computations.

The aforementioned sparse matrix formats do not explicitly store the number of non-zero elements per row because it is not necessary in common algorithms such as the matrix–vector multiplication. However, mesh algorithms might require a direct knowledge of the number of subentities or superentities of a given entity. Therefore, each incidence matrix I_{d_1,d_2} and the adjacency matrix A_D of the dual

graph must be explicitly supplemented by an array storing the numbers of non-zero elements per row, unless the values can be determined a priori without additional storage. Recall that for an incidence matrix I_{d,d_1}, $d > d_1$, the elements stored in the j-th row correspond to the d_1-dimensional subentities of the j-th d-dimensional entity. If the d-dimensional entities have a static topology (e.g., a quadrangle or a simplex), the number of d_1-dimensional subentities (for any $d_1 < d$) is known a priori and can be provided as a compile-time constant, so the additional array for the numbers of non-zero elements per row is not necessary.

Internal storage layers

The data structure needs to contain several matrices and arrays for the mesh representation depending on the configuration. The internal data structures can be organized in several layers for each dimension of entities in the mesh. Individual storage layers can be combined using C++ features such as *recursive inheritance* and *partial class template specializations* in order to provide an efficient and generic D-dimensional data structure. Recursive inheritance is a C++ design pattern in which an instance of a class template inherits from another instance of the same class template but with different template arguments. Hence, in this case inheritance is not used to express a relationship between objects, but to generically include any finite number of differently typed attributes into a single object.

The exact hierarchy of class templates comprising the `Mesh` class template is quite complicated and technical. A brief description of the main layers in the hierarchy is included in Appendix C, which is based on the paper [A110]. An in-depth explanation and additional comments related to the implementation of the storage layers can be found in the thesis [B5].

Mapping user data to mesh entities

The mesh data structure is designed primarily for computational algorithms such as finite element or finite volume methods. An important part of computing on meshes is defining a mapping between user data and mesh entities. Unlike some other libraries such as OpenMesh [C12] where arbitrary user data can be stored directly in the mesh object, the data structure presented in this paper does not address the problem of mapping user data to the mesh. Mesh entities can be identified by their C++ type (i.e., topology) and global index, and adjacent entities are accessible using local indices. This information can be used to define a natural mapping between the mesh entities and an appropriate data container storing user data separately from the mesh. Depending on the specific application, a suitable container might be e.g. a (one-dimensional) array, a multi-dimensional array, an array of structures or a structure of arrays. As described in Section 1.3, TNL provides implementations of one-dimensional and general multi-dimensional arrays (`Array` and `NDArray`) including full GPU support. The storage layout of the latter is configurable by a permutation of indices, which allows to easily switch the memory layout (see Section 2.1 for details).

Orientations of mesh entities

For computational algorithms such as finite element or finite volume methods, it is often necessary to define *orientations* of mesh entities. Unfortunately, the term *orientation* is commonly used in two different contexts:

1. In case of the finite volume method, one is interested in the direction of the normal vectors uniquely assigned to each face of the mesh. Each face can have up to two adjacent cells and its normal vector can be oriented in two ways: either outward or inward with respect to the first adjacent cell (i.e., inward or outward, respectively, with respect to the other cell). This information is also sufficient for some low-order finite element discretizations, e.g. the lowest order Raviart–Thomas elements that we used in [A72].

The orientation of faces based on this context is stored implicitly in the presented data structure. Although it does not explicitly store the normal vectors for each face, their direction can be deduced based on the topology of the face and one of its adjacent cells. The core implementation detail is that column indices in each row of the incidence matrices are not sorted. Thus, for each mesh entity, we can define local indices of its subentities and superentities corresponding to the positions in the rows related to the entity. For example, the incidence matrix $I_{D-1,D}$ contains one or two entries per row which correspond to the cells adjacent to the face given by the particular row index. The first entry is assigned a local index 0 and its column index corresponds to the so called *owner cell* of the face, whereas the second entry (if it exists) is assigned a local index 1 and its column index is related to the so called *neighbor cell* of the face. The orientation of the face can be inferred from the topology of its owner cell (see Appendix B), because the owner cell is the one from which the face was initialized.

When appropriate, the orientation of faces needs to be extracted manually from the data structure using a suitable algorithm such as the following. For example, given a cell denoted by index K and its adjacent face denoted by index F, we are interested in whether the cell K is the owner or neighbor cell of the face F. First, we can use the templated function `getSuperentitiesCount` to obtain the number of cells adjacent to the face F. If we get one, it must be the cell K which is also the owner cell. Otherwise, we can pass the local index 0 to the templated function `getSuperentityIndex` to obtain the global index K_0 of the face's owner cell that can be compared to the index K. If K equals K_0, then K is the owner cell, otherwise it is the neighbor cell and the orientation of F needs to be determined based on K_0.

2. For high-order finite element methods, it is often necessary to define local coordinate systems on each cell such that degrees of freedom associated to faces and edges shared by adjacent cells can be located consistently when the face or edge is viewed from each of the adjacent cells. In this context, orientation of the entities relates to the local ordering of their subvertices with respect to their adjacent cells. This case is more general than the former since entities other than faces are considered (e.g. edges in 3D) and even the direction of normal vectors on faces can be based on consistent local ordering of face subvertices.

   However, the current implementation of the presented data structure does not provide any way to deal with general mesh entity orientations according to this context. Hence, implementing computational methods that need this information, such as high-order finite element methods, might require using additional data structures to store the necessary mesh entity orientations, or following a different approach that would avoid the need for additional storage. Various strategies on dealing with this general case can be found in [A1].

2.2.5 Implementation details

In this section, we describe additional implementation details related to mesh initialization, input from files and reordering of the mesh entities, which is a common optimization to improve data locality. The high-level documentation for programmers, including code examples, can be found in the TNL project's documentation [O31] in the section related to unstructured meshes.

Mesh initialization

Each mesh object, i.e., instance of the `Mesh<Config, Device>` type, has to be initialized prior to its first use. The former class template parameter, `Config`, refers to a specific configuration class (see Subsection "Static configuration" on Page 33) and the latter class template parameter, `Device`, denotes the *device* in which memory space the mesh will be allocated (see Section 1.3.2). In our implementation, the initialization process is sequential and done entirely on the CPU and then, the complete mesh can

be transferred to the GPU using an overloaded assignment operator. The initialization is done via a public member function `init` of the `Mesh` class template, which is usable only when `TNL::Devices::Host` is specified as the class template parameter `Device`.

The member function `init` takes three arguments: `points`, `faceSeeds`, and `cellSeeds`, which have the following meaning:

1. The `points` argument is an array of `PointType` objects. The i-th object in the array represents Cartesian coordinates of the i-th vertex in the Euclidean space.

2. The `faceSeeds` argument is used only for the initialization of polyhedral meshes, otherwise it is specified as an empty object. It is an instance of the `FaceSeedMatrix` type which comprises the incidence matrix $I_{2,0}$ (i.e., it represents subvertex indices of polygonal faces).

3. The `cellSeeds` argument is an instance of the `CellSeedMatrix` type and its meaning depends on whether a polyhedral or non-polyhedral mesh is being initialized. For a polyhedral mesh, it comprises the incidence matrix $I_{3,2}$ (i.e., it represents indices of polygonal faces of each polyhedron). On the other hand, cells of a non-polyhedral mesh can be defined directly in terms of their subvertices, so the `cellSeeds` argument comprises the incidence matrix $I_{D,0}$ in this case.

Note that the vertex indices in each row of the input incidence matrices $I_{2,0}$ and $I_{D,0}$ must follow a specific order based on the topology of the entity (see Appendix B). The remaining uninitialized incidence matrices are deduced from the specific ordering of vertex indices in the input incidence matrices.

The initialization procedure for a D-dimensional *non-polyhedral* mesh is carried out in the following phases. First, the physical coordinates of vertices are moved from the input `points` array into the mesh. Next, the `cellSeeds` argument initializes the incidence matrix $I_{D,0}$ and its transpose $I_{0,D}$ in the mesh. Then, for all $d = D - 1, D - 2, \ldots, 1$, the following steps are repeated:

1. Create intermediate seeds for the d-dimensional entities and store them in an intermediate *indexed set* container represented by `std::unordered_map<Seed, Index>` in our implementation. Note that to determine if two seeds represent the same entity, the subvertex indices must be sorted first, but most of the comparisons can be avoided by hashing. In order to avoid the sorting of subvertices, the hashes of the subvertex indices must be combined with a commutative operation (e.g., integer addition).

2. Initialize the incidence matrix $I_{d,0}$ (i.e., links from d-dimensional entities to their subvertices) and its transpose $I_{0,d}$ (i.e., links from vertices to the d-dimensional entities).

3. For all $s = d + 1, d + 2, \ldots, D$, initialize the incidence matrix $I_{s,d}$ (i.e., links from s-dimensional entities to their d-dimensional subentities) and its transpose $I_{d,s}$ (i.e., links from d-dimensional entities to their s-dimensional superentities).

4. Deallocate all intermediate data structures.

Finally, supplementary data members such as the dual graph or boundary entity tags are initialized. Note that if some matrices are disabled in the mesh configuration, the relevant initialization steps are omitted to reduce the memory and computational cost of the initialization.

The initialization procedure for a 3-dimensional polyhedral mesh is slightly different. First, the `faceSeeds` argument is used to initialize a wireframe of the mesh using the aforementioned procedure, where the incidence matrix $I_{2,0}$ is used as input instead of the $I_{D,0}$ matrix. Then, the `cellSeeds` argument initializes the incidence matrix $I_{3,2}$ and its transpose $I_{2,3}$. Finally, the remaining incidence matrices $I_{3,1}$, $I_{1,3}$, $I_{3,2}$, $I_{2,3}$, $I_{3,0}$, and $I_{0,3}$ are initialized, unless disabled in the mesh configuration.

The initialization process results either in a system error or in a populated mesh object containing all matrices and other attributes as specified in the mesh configuration. The correctness of the mesh representation depends on the input data in a sense that topological properties of the mesh entities are not checked by the initializer itself. For convenience, an intermediate class template called `MeshBuilder` with basic input data validation and text file readers for common formats are provided as well.

Input from files

In practice, scientific solvers usually initialize a mesh by reading an input file prepared by another specialized tool. The TNL library currently provides facilities for input as well as output in the legacy VTK, VTU, Netgen, and FPMA file formats. The parallel format PVTU is also supported for distributed meshes which will be described in Section 2.2.6.

The process of importing a mesh from a file into a `Mesh` object is divided into three phases. First, the features of the mesh are determined by examining the file. Next, the appropriate type template arguments of the `Mesh` class template are resolved at run-time. Finally, the mesh representation is imported from the file into a mesh object of the appropriate type.

The (compile-time) type of a mesh in a particular file can be resolved using a variadic class template called `MeshTypeResolver` from the TNL library. `MeshTypeResolver` performs the selection of template arguments for the default mesh configuration class template at run-time similarly to [C28]. The implementation of `MeshTypeResolver` is rather complicated, but its usage can be summarized with the following pseudo-code:

```
using Reader = [based on file type detection]
Reader reader(input_file);
using MeshType = [resolved by MeshTypeResolver from the reader]
MeshType mesh = reader.readMesh();
return functor(reader, mesh);
```

The user-supplied `functor` object is typically a C++ *lambda expression* which continues the computation with a fully initialized mesh.

Reordering of entities

The sparse matrix formats described in Subsection “Internal data structures” on Page 33 provide an efficient access to the subentity and superentity indices of a mesh entity, but the efficiency does not propagate to the access to other entities using the obtained indices. For example, if we access the incidence matrix $I_{2,1}$ to obtain the indices of edges incident to a two-dimensional mesh entity, and then use the obtained indices to query the incidence matrix $I_{1,0}$ to obtain the indices of edge subvertices, the efficiency of accessing the latter incidence matrix cannot be guaranteed and depends on the locality of obtained edge indices. A common optimization strategy for similar scenarios is to improve data locality by applying a suitable reordering algorithm [A163]. Common approaches typically involve an *in-order traversal* of a quadtree or octree which considers the geometric locality of mesh entities, or an application of a suitable space-filling curve such as the Hilbert curve [A98], or a graph ordering such as the Cuthill–McKee algorithm [C17] which is known to reduce the bandwidth of the graph’s adjacency matrix, or a graph colouring technique such as [A90].

The `Mesh` class template provides an interface for applying any custom ordering of mesh entities. The d-dimensional entities of a mesh can be reordered by calling the templated member function `reorderEntities` and specifying d as the template parameter:

```
1 template< int Dimension >
2 void reorderEntities( const GlobalIndexVector& perm, const GlobalIndexVector& iperm );
```

The `GlobalIndexVector` type stands for a vector of `GlobalIndex` elements allocated on the same device as the mesh. The vectors `perm` and `iperm` represent two mutually inverse permutations of indices from 0 to $N_d - 1$, where N_d is the number of d-dimensional entities in the mesh. The reordering is done according to the definition used in the METIS library [A107]. Let $\mathcal{M}$ be the original mesh and $\mathcal{M}'$ the permuted mesh. Then, the entity with an index i in $\mathcal{M}'$ is the entity with an index `perm[ i ]` in $\mathcal{M}$ and the entity with an index j in $\mathcal{M}$ is the entity with an index `iperm[ j ]` in $\mathcal{M}'$.

The reordering is performed by applying the permutations to all relevant incidence matrices and arrays. First, `perm` is used to permute the rows of all incidence matrices $I_{d,d'}$, where $d' \neq d$ stands for the dimension of subentities or superentities of d-dimensional entities. Next, `iperm` is used to permute the columns of all incidence matrices $I_{d',d}$, where $d' \neq d$ has the same meaning. Both `perm` and `iperm` are also applied the same way to permute the dual graph $I_{d,d}$. Finally, `perm` is used to permute data stored in plain arrays (e.g., vertex coordinates if $d = 0$) and extra properties such as boundary tags are re-initialized.

2.2.6 Distributed mesh

In this section, we describe an extension of the base data structure which allows to perform distributed computations on an unstructured conforming homogeneous mesh.

Distributed mesh generation

The obvious requirement for distributed computations on unstructured meshes is the generation of a distributed mesh, which is typically performed by partitioning an existing mesh into several subdomains following the domain decomposition approach. When the mesh is decomposed and the program is executed via MPI [M15], each subdomain can be assigned to a different MPI rank to provide parallelism to the computation.

The TNL library provides the tool `tnl-decompose-mesh` which performs the partitioning using the METIS library [A107]. Similarly to the `decomposePar` tool from the OpenFOAM framework [M9], `tnl-decompose-mesh` is run sequentially as a separate process before the main computation. It reads an input mesh representation from a file in the VTK, VTU, Netgen, or FPMA format, and writes the decomposed mesh into the parallel format PVTU which comprises several files per each subdomain.

In addition to mesh partitioning, `tnl-decompose-mesh` takes care of several features that are needed for our distributed mesh data structure. Firstly, it assigns global indices for vertices and cells such that the *owner* of each entity can be easily identified even for common entities on the interfaces between subdomains. Global indices are assigned separately in each dimension in such a way that for each two ranks i and j, $i < j$, all d-dimensional entities owned by rank i have a smaller global index than any d-dimensional entity owned by rank j. Next, it can generate *overlaps* (also called *ghost regions*) across subdomains. This results in several layers of vertices and cells close to the subdomain boundary being added to the local mesh (i.e., the mesh representing space discretization of the given subdomain) from other subdomains. This is needed for communication patterns between MPI ranks where data associated with mesh entities need to be exchanged. Finally, an appropriate renumbering of mesh entities on each subdomain is generated. In each dimension, mesh entities are ordered such that entities owned by the current MPI rank are numbered first, followed by the ghost entities owned by the first neighbor, then ghost entities owned by the second neighbor, and so on. This ordering simplifies the implementation and increases the efficiency of data synchronization on ghost regions, which will be described later.

Distributed mesh implementation

Since the distributed mesh stored in a PVTU file contains additional data compared to the original mesh (e.g., global entity indices), a supplementary data structure, where the PVTU file can be loaded, needs to be provided. The TNL library provides the `DistributedMesh<typename Mesh>` class template that contains the following parts:

- The local mesh of type `Mesh` representing a specific subdomain.
- The number of ghost region levels, stored as an integer (`int`).
- Arrays of global indices for each entity in the local mesh. In case of vertices and cells, these arrays are read from the PVTU file. For other dimensions they have to be generated if needed.
- Arrays of vertex and cell types (`vtkGhostType`) which are needed for data output in VTK file formats.

The local mesh contains all entities owned by the MPI rank which was assigned the corresponding subdomain, as well as ghost entities that are owned by other ranks. The type of each entity can be obtained by querying the array of entity tags (see Subsection "Internal storage layers" on Page 35). Specifically, each entity is either *local* or *ghost*, and from the geometric point of view, each entity is either *internal* or *boundary*.

Data synchronization on ghost regions

Computations following the domain decomposition approach typically involve synchronization to ensure data consistency on ghost regions. This is done by sending data computed by an MPI rank on its subdomain to the ghost regions of its neighbors, and from the opposite side, receiving data computed by neighbors into the ghost regions around the subdomain.

The data synchronization process is implemented in the `DistributedMeshSynchronizer` class template as a separate data structure that can be instantiated for a specific distributed mesh and dimension d of entities being synchronized. The synchronizer can be used to synchronize data arrays allocated anywhere, i.e., host–to–host, host–to–GPU, as well as GPU–to–GPU communication is supported. Note that thanks to the so-called CUDA-aware MPI [O18], we do not have to deal with handling custom buffers when receiving or sending data allocated on the GPU.

The synchronizer needs to be initialized before its use. Let P denote the number of MPI ranks among which the mesh is distributed. Then the initialization phase involves the following steps:

1. An unsymmetric $P \times P$ communication matrix G is created such that $G_{i,j}$ represents the number of d-dimensional ghost entities on rank i that are owned by rank j.
2. Each rank collects the indices of the first entity owned by each neighbor and stores them in an array of size P.
3. Each rank collects the indices of local entities, which are ghosts on a different subdomain. This can be represented by a binary sparse matrix with P rows.

During synchronization, each pair of ranks exchanges data according to the communication matrix. The data to be sent may be non-contiguous, so they have to be copied into a contiguous buffer prior to the transfer. On the other hand, the data to be received are contiguous thanks to the appropriate numbering of mesh entities performed by `tnl-decompose-mesh`, so buffering is not necessary in this case.

The aforementioned initialization process is sufficient for synchronizing data associated to entities that have their global indices assigned already in the PVTU file (i.e., vertices, cells, and also faces

in polyhedral meshes). On the other hand, remaining entities that do not exist in an input file (e.g., edges, or faces in non-polyhedral meshes) do not have global indices assigned yet. In this case, the remaining entities have to be initialized at run-time via MPI before the synchronizer can be used for these entities. The algorithm starts by each MPI rank determining and counting all *local* uninitialized entities, exchanging the counts with other MPI ranks, and assigning a global index to all local entities. All remaining entities whose global index could not be determined in the first phase are located either on the interface between subdomains or in the ghost region, and their global index must be received from the rank which owns the entity. It can be observed that the uninitialized entities can be unambiguously identified by their subvertices, which already have global indices assigned. Hence, neighboring ranks exchange the tuples of global subvertex indices, identify the requested entities in their local mesh and send back global indices to the neighbors. Finally, each rank reorders the newly initialized entities on its local mesh based on the global indices.

2.2.7 Benchmarking methodology

We used several benchmark problems to evaluate the data structure for the representation of unstructured meshes. In this subsection, we describe the methodology related to the execution of the benchmark problems and evaluation of the results.

All computations presented in this section were performed either on the 12-core CPU Intel Gold 6146 or on the GPU NVIDIA Tesla V100, whose characteristic properties are listed in Table 2.1. The code for the CPU was compiled with the `g++` compiler version 10.2.0 and the code for the GPU was compiled with the `nvcc` compiler version 11.1.105. Common flags for both compilers were `-std=c++14 -O3`. Additional optimization flags for the CPU code were `-march=native -mtune=native`, link-time optimizations were enabled with `-flto`, and OpenMP support was enabled with `-fopenmp`. Additional compiler flags for `nvcc` were `--expt-relaxed-constexpr --expt-extended-lambda` and the GPU architecture was set to `sm_70`.

We compared computational times CT for sequential (single-core) computations on the CPU, parallel multi-core computations on the CPU using OpenMP [A51], and parallel computations on one GPU accelerator using CUDA [M19] parallelization. Note that the distributed mesh is not tested in this section. To obtain statistically significant results, the computational time was taken as the average of 100 iterations of each computation. The secondary quantity of interest for the comparison of sequential computations and parallel computations using ℓ CPU cores is the speed-up Sp_ℓ, which is defined as the ratio between computational times using 1 and ℓ cores. For computations on the GPU, the quantity of interest is the GPU speed-up GSp_ℓ defined as the ratio between the computational times on CPU using ℓ cores and GPU. To compare the results with the hardware peak memory throughput, we also calculate an effective bandwidth EBW for each algorithm. The EBW is calculated by dividing the *useful data size*, which is problem-specific and specified later, by the achieved computational time.

Table 2.1: Hardware used for the computation of benchmarks [O14, M22]. The cluster is operated by the *Research Center for Informatics* (`http://rci.cvut.cz/`).

	CPU	GPU
Model	Intel Xeon Gold 6146	NVIDIA Tesla V100
Cores	12	5120
Core frequency	3.2-4.2 GHz	1.455 GHz
L2 cache size	12×1 MiB	6 MiB
L3 cache size	24.75 MiB	—
Amount of memory	384 GiB	32 GiB
Peak memory throughput	6×21 GB/s	900 GB/s

Table 2.2: Properties of the triangular ($2D_i^{\triangle}$) and tetrahedral ($3D_i^{\triangle}$) meshes used in the benchmarks: mesh size h (radius of the largest sphere circumscribed around a cell), number of vertices $\#\mathcal{V}$, number of faces $\#\mathcal{F}$, and number of cells $\#\mathcal{C}$.

Id.	h [m]	$\#\mathcal{V}$	$\#\mathcal{F}$	$\#\mathcal{C}$
$2D_1^{\triangle}$	6.71×10^{-2}	142	383	242
$2D_2^{\triangle}$	3.49×10^{-2}	513	1456	944
$2D_3^{\triangle}$	1.64×10^{-2}	1938	5651	3714
$2D_4^{\triangle}$	8.73×10^{-3}	7555	22342	14788
$2D_5^{\triangle}$	4.23×10^{-3}	29989	89324	59336
$3D_1^{\triangle}$	2.13×10^{-1}	395	2937	1312
$3D_2^{\triangle}$	1.27×10^{-1}	824	7773	3697
$3D_3^{\triangle}$	6.29×10^{-2}	5740	60839	29673
$3D_4^{\triangle}$	3.48×10^{-2}	43293	486875	240372
$3D_5^{\triangle}$	1.84×10^{-2}	336608	3903609	1939413

Table 2.3: Properties of the polygonal ($2D_i^{*}$) and polyhedral ($3D_i^{*}$) meshes used in the benchmarks: number of vertices $\#\mathcal{V}$, number of faces $\#\mathcal{F}$, number of cells $\#\mathcal{C}$, average number of subvertices per cell $\#\mathcal{VC}$, average number of vertices per face $\#\mathcal{VF}$, and average number of faces per cell $\#\mathcal{FC}$.

Id.	$\#\mathcal{V}$	$\#\mathcal{F}$	$\#\mathcal{C}$	$\#\mathcal{VC}$	$\#\mathcal{VF}$	$\#\mathcal{FC}$
$2D_1^{*}$	322	463	142	6.0	2.0	6.0
$2D_2^{*}$	1104	1616	513	6.0	2.0	6.0
$2D_3^{*}$	4034	5971	1938	6.0	2.0	6.0
$2D_4^{*}$	15428	22982	7555	6.0	2.0	6.0
$2D_5^{*}$	60616	90604	29989	6.0	2.0	6.0
$3D_1^{*}$	2018	2410	395	18.4	5.0	11.2
$3D_2^{*}$	4535	5356	824	20.9	5.1	12.4
$3D_3^{*}$	32811	38548	5740	22.3	5.1	13.1
$3D_4^{*}$	252930	296220	43293	23.1	5.1	13.5
$3D_5^{*}$	1989563	2326168	336608	23.5	5.1	13.7

The benchmarks presented in the following subsections were performed on a 2D or 3D computational domain consisting of a unit square or cube, respectively. The first computations were performed on series of 2D and 3D triangulations of the computational domain with properties summarized in Table 2.2. The triangular meshes were generated using the frontal algorithm in Gmsh [A85] and reordered using the reverse Cuthill–McKee algorithm [C17] to improve data locality. The tetrahedral meshes were generated by the COMSOL Multiphysics software [M8]. Additionally, polygonal and polyhedral meshes were created from the aforementioned triangular and tetrahedral meshes by creating a dual mesh from the input simplex mesh using the CinoLib [B23] library. The duality means that cells of the dual mesh correspond to vertices of the primal mesh and vertices of the dual mesh correspond to cells of the primal mesh. The properties of the dual meshes are summarized in Table 2.3.

2.2.8 Benchmark problems: algorithms and implementation

We developed two benchmark problems to compare the efficiency of the data structure under different configurations and against MOAB [M28]. The first benchmark is the calculation of the measures of each cell in the mesh, which involves launching as many independent parallel tasks as there are cells

in the mesh, calculating the measure of one cell per task, and writing the result into a global array. The calculation of the cell measure involves reading the subvertices of the cell and using their spatial coordinates in the formula for the cell measure. For example, the computational kernel for triangular meshes can be implemented using an *extended lambda expression* [M19] as follows:

```
auto kernel_measures = [] __host__ __device__
( GlobalIndex cell_idx, const Mesh& mesh, Real* output_array )
{
    const auto& cell = mesh.template getEntity< 2 >( cell_idx );
    const auto& v0 = mesh.getPoint( cell.template getSubentityIndex< 0 >( 0 ) );
    const auto& v1 = mesh.getPoint( cell.template getSubentityIndex< 0 >( 1 ) );
    const auto& v2 = mesh.getPoint( cell.template getSubentityIndex< 0 >( 2 ) );
    output_array[ cell_idx ] = getTriangleArea( v2 - v0, v1 - v0 );
};
```

Note that the lambda expression `kernel_measures` is passed to the TNL's `ParallelFor` function which executes it in parallel on the specified device.

The second benchmark is the calculation of the measure of the boundary on each cell in the mesh. Unlike the first benchmark, the independent parallel tasks are executed per face and the calculation involves the calculation of the face measure (i.e., length or area depending on the mesh dimension), reading indices of the cells incident to the face, and adding the face measure into the output array. Since there might be multiple parallel tasks writing their value to the same element of the output array at the same time, the addition must be performed atomically. The computational kernel can be implemented as follows:

```
1  auto kernel_boundary_measures = [] __host__ __device__
2  ( GlobalIndex face_idx, const Mesh& mesh, Real* output_array )
3  {
4      constexpr int D = Mesh::getMeshDimension();
5      const auto& face = mesh.template getEntity< D - 1 >( face_idx );
6      const auto face_measure = getEntityMeasure( mesh, face );
7      const auto cells_count = face.template getSuperentitiesCount< D >();
8      for( LocalIndex c = 0; c < cells_count; c++ ) {
9          const auto cell_idx = face.template getSuperentityIndex< D >( c );
10         TNL::Algorithms::AtomicOperations< Device >::add( output_array[ cell_idx ], face_measure );
11     }
12 };
```

The lambda expression `kernel_boundary_measures` is again passed to the TNL's `ParallelFor` function which executes it in parallel on the specified device. The function `getEntityMeasure` used at line 6 calculates the measure of the given entity in the given mesh, similarly to the explicit calculation for triangles in the previous kernel.

Note that the second benchmark presented in this section is different from the second benchmark used in the paper [A110]. The latter computes the boundary measure of a patch of cells around a vertex in the mesh and its efficient implementation relies on the local indexing convention for the opposite vertices and faces in simplex meshes. While the algorithm can be generalized for polygonal and polyhedral meshes, it would necessarily have a very low arithmetic intensity, because the number of arithmetic computations would be the same, but the amount of mesh data needed to be read would be significantly higher on polygonal and especially polyhedral meshes. For this reason, we used the aforementioned algorithm computing the boundary measures of each cell in the mesh, which has similar computational characteristics on all mesh types and on simplex meshes it is even comparable to the second benchmark presented in [A110].

To compare the data structure implemented in TNL with another library, we also implemented these two benchmarks using MOAB [M28]. However, there are some inevitable differences. All meshes in

MOAB are implicitly 3D, so there is no way to check if a triangle or a general polygon comes from a 2D or 3D space, other than checking the third component of vertex coordinates. The area of a triangle in 2D can be calculated using a 2×2 determinant, but formulas for the calculation of a triangle area in 3D involve the expensive square-root function. Similarly, the area of a polygon in 2D can be calculated using the surveyor's formula [A33] (also known as the shoelace formula), but the algorithm for the calculation of a polygon area in 3D is more complicated [B29]. In order to match the benchmark using the TNL data structure, we had to explicitly invoke the functions based on the specific 2D formulas in the first benchmark involving triangular and polygonal meshes in 2D. In the second benchmark, we used the general formulas for the calculation of areas of triangles and polygons in 3D.

The code of all benchmarks described in this subsection is available as open-source software in the following projects:

- `https://gitlab.com/tnl-project/tnl-benchmark-mesh` (the TNL implementation),
- `https://gitlab.com/tnl-project/moab-benchmark` (the MOAB implementation).

To compare the results with the hardware peak memory throughput, we calculate an effective bandwidth *EBW* for each algorithm. The *EBW* is calculated by dividing the *useful data size* for the given computation by the achieved computational time. For both benchmark algorithms, the useful data size consists of the size of all vertex coordinates, the size of all indices read by the algorithm, and the size of output data written by the algorithm to the memory. Note that this is a more accurate approach compared to [A110] where the output data size is not considered in *EBW* calculations. The number of write operations equals $\#\mathcal{C}$ in the first benchmark and approximately $2 \cdot \#\mathcal{F}$ in the second benchmark. The number of indices read by each algorithm depends on the mesh type.

In the first benchmark, the data size for simplex meshes is given by

$$\text{data size}_1^{\triangle} = \texttt{sizeof(Real)} \cdot (D\ \#\mathcal{V} + \#\mathcal{C}) + \texttt{sizeof(GlobalIndex)} \cdot (D+1) \cdot \#\mathcal{C},$$

the data size for polygonal meshes is given by

$$\begin{aligned}\text{data size}_1^{*} = {} & \texttt{sizeof(Real)} \cdot (D\ \#\mathcal{V} + \#\mathcal{C}) + \texttt{sizeof(GlobalIndex)} \cdot \#\mathcal{VC} \cdot \#\mathcal{C} \\ & + \texttt{sizeof(LocalIndex)} \cdot \#\mathcal{C},\end{aligned}$$

and the data size for polyhedral meshes is given by

$$\begin{aligned}\text{data size}_1^{**} = {} & \texttt{sizeof(Real)} \cdot (D\ \#\mathcal{V} + \#\mathcal{C}) + \texttt{sizeof(GlobalIndex)} \cdot (\#\mathcal{FC} \cdot \#\mathcal{C} + \#\mathcal{VF} \cdot \#\mathcal{F}) \\ & + \texttt{sizeof(LocalIndex)} \cdot (\#\mathcal{F} + \#\mathcal{C}),\end{aligned}$$

where the function `sizeof` returns the size of the given type in bytes, D stands for the mesh dimension, and the counts of entities $\#\mathcal{V}$, $\#\mathcal{F}$, $\#\mathcal{C}$ and the average counts of subentities $\#\mathcal{FC}$, $\#\mathcal{VC}$, $\#\mathcal{VF}$ are given in Tables 2.2 and 2.3.

In the second benchmark, the data size for simplex and polygonal meshes is given by

$$\begin{aligned}\text{data size}_2^{\triangle} = \text{data size}_2^{*} = {} & \texttt{sizeof(Real)} \cdot (D\ \#\mathcal{V} + 2\ \#\mathcal{F}) + \texttt{sizeof(GlobalIndex)} \cdot D \cdot \#\mathcal{F} \\ & + (2\ \texttt{sizeof(GlobalIndex)} + \texttt{sizeof(LocalIndex)}) \cdot \#\mathcal{F},\end{aligned}$$

and the data size for polyhedral meshes is given by

$$\begin{aligned}\text{data size}_2^{**} = {} & \texttt{sizeof(Real)} \cdot (D\ \#\mathcal{V} + 2\ \#\mathcal{F}) + \texttt{sizeof(GlobalIndex)} \cdot \#\mathcal{VF} \cdot \#\mathcal{F} \\ & + 2\ \ (\texttt{sizeof(GlobalIndex)} + \texttt{sizeof(LocalIndex)}) \cdot \#\mathcal{F},\end{aligned}$$

where the symbols `sizeof` and D have the same meaning and the counts of entities and average counts of subentities can be found in Tables 2.2 and 2.3.

Table 2.4: Memory requirements [MiB] for storing the triangular and tetrahedral meshes from Table 2.2 in the data structure with different configurations. The minimal configuration includes only those incidence matrices that are needed for the benchmarks presented in this section, whereas all incidence matrices are included in the maximal configuration, even if they are unused. Each column corresponds to different data types for `Real`, `GlobalIndex` and `LocalIndex` in the mesh configuration, which are recorded in triplets in the second header row: `f` (`float`), `d` (`double`), `s` (`short int`), `i` (`int`), and `l` (`long int`).

Id.	Minimal configuration				Maximal configuration			
	f, i, s	f, l, i	d, i, s	d, l, i	f, i, s	f, l, i	d, i, s	d, l, i
$2D_1^\triangle$	0.01	0.03	0.02	0.03	0.03	0.06	0.03	0.06
$2D_2^\triangle$	0.05	0.10	0.06	0.10	0.11	0.21	0.11	0.21
$2D_3^\triangle$	0.20	0.38	0.21	0.40	0.41	0.79	0.42	0.80
$2D_4^\triangle$	0.79	1.51	0.84	1.57	1.60	3.10	1.66	3.15
$2D_5^\triangle$	3.14	6.06	3.37	6.29	6.36	12.32	6.59	12.55
$3D_1^\triangle$	0.12	0.24	0.13	0.25	0.48	0.94	0.48	0.94
$3D_2^\triangle$	0.32	0.64	0.33	0.65	1.21	2.36	1.20	2.36
$3D_3^\triangle$	2.55	5.02	2.61	5.09	9.16	18.12	9.22	18.19
$3D_4^\triangle$	20.38	40.25	20.87	40.75	72.91	144.31	73.40	144.80
$3D_5^\triangle$	163.43	323.01	167.28	326.86	583.43	1154.90	587.28	1158.76

Note that the calculated bandwidth is effective, because we assume in the data size calculation that all data is read exactly once, but in practice, some data (e.g., vertex coordinates) are accessed multiple times from the surrounding entities and it depends on other factors if these accesses can be efficiently cached or not.

2.2.9 Benchmark results: triangular and tetrahedral meshes

The results of the first benchmark problem are shown in Table 2.5 and the results of the second benchmark problem are shown in Table 2.6. The benchmarks were performed on all meshes shown in Table 2.2, but, for brevity, only the results for the finest mesh in 2D and 3D ($2D_5^\triangle$ and $3D_5^\triangle$, respectively) are shown in Tables 2.5 and 2.6.

Both benchmarks were performed with different configurations of the `Mesh` class in TNL using different data types for template parameters: `float` (`f`) and `double` (`d`) for `Real`, `int` (`i`) and `long int` (`l`) for `GlobalIndex`, and `short int` (`s`) and `int` (`i`) for `LocalIndex`. For brevity, only four representative sets of types are shown in the tables. It can be seen in Tables 2.5 and 2.6 that selecting the smallest possible types for the `Real`, `GlobalIndex`, and `LocalIndex` template parameters generally leads to the fastest computations on both CPU and GPU. In case of MOAB [M28], we used only one set of types (`double` for vertex coordinates, `long int` for global indices and `int` for counts of adjacent entities), which was specified when installing MOAB. Finally, we would like to remark that the performance of all benchmarks performed with the TNL library does not depend on other mesh configuration parameters, such as `subentityStorage` or `superentityStorage`, which can be used to disable the storage of unnecessary incidence matrices. This behavior is not included in Tables 2.5 and 2.6, because it would lead to duplicate tables with the same values. Hence, such configuration parameters influence the dynamic size required for the mesh representation as shown in Table 2.4, but not the performance of computations using the data structure.

The operations comprised in the benchmarks involve many independent tasks so they can be easily parallelized. In case of cell measure calculations shown in Table 2.5, the efficiency of the CPU parallelization depends on the ratio between compute operations and memory accesses. The 2D case of this benchmark involves simpler computations (a 2×2 determinant to calculate the triangle area,

Table 2.5: *Calculation of cell measures* on the finest triangular mesh $2D_5^{\triangle}$ and finest tetrahedral mesh $3D_5^{\triangle}$. Comparison of computational times *CT* [ms], effective bandwidth *EBW* [GB/s], CPU speed-ups Sp_ℓ, and GPU speed-ups GSp_ℓ. Each row corresponds to different data types for `Real`, `GlobalIndex` and `LocalIndex` in the mesh configuration, which are recorded in the triplet in the "types" column: `f` (`float`), `d` (`double`), `s` (`short int`), `i` (`int`), and `l` (`long int`).

			1 th.		12 threads			GPU			
		types	*CT*	*EBW*	*CT*	*EBW*	Sp_{12}	*CT*	*EBW*	GSp_1	GSp_{12}
TNL	$2D_5^{\triangle}$	f, i, s	0.103	11.6	0.018	66	5.7	0.010	115	9.9	1.7
		f, l, i	0.104	18.2	0.017	114	6.3	0.011	177	9.7	1.5
		d, i, s	0.115	14.4	0.020	84	5.8	0.011	153	10.6	1.8
		d, l, i	0.120	19.9	0.019	127	6.4	0.012	207	10.4	1.6
	$3D_5^{\triangle}$	f, i, s	12.656	3.4	0.748	57	16.9	0.087	493	145.7	8.6
		f, l, i	13.299	5.6	1.192	62	11.2	0.123	603	108.5	9.7
		d, i, s	13.118	4.2	0.828	66	15.9	0.113	483	116.0	7.3
		d, l, i	13.874	6.2	1.098	78	12.6	0.145	592	95.9	7.6
MOAB	$2D_5^{\triangle}$	d, l, i	1.691	1.4	0.207	12	8.2				
	$3D_5^{\triangle}$	d, l, i	59.122	1.4	5.845	15	10.1				

1.3 ops/mem) than the 3D case (a 3×3 determinant to calculate the tetrahedron volume, 2.4 ops/mem). Consequently, the effective bandwidth *EBW* for computations using 12 CPU threads on the $2D_5^{\triangle}$ mesh approaches the hardware peak performance (see Table 2.1) and the speed-up Sp_{12} is much lower than 12, but in case of the $3D_5^{\triangle}$ mesh the speed-up Sp_{12} is much higher and the effective bandwidth *EBW* is much lower. On the other hand, the efficiency of the GPU parallelization depends primarily on the amount of data elements being processed. The finest triangular mesh $2D_5^{\triangle}$ contains about $60 \cdot 10^3$ cells, while the finest tetrahedral mesh $3D_5^{\triangle}$ contains about $2 \cdot 10^6$ cells (see Table 2.2). Hence, this benchmark cannot utilize the full potential of the GPU on the triangular mesh where the GPU achieves only moderately higher effective bandwidth compared to the CPU and the speed-ups GSp_{12} are only moderately greater than 1. On the other hand, the tetrahedral mesh allows the benchmark to fully utilize the GPU leading to approximately 3-4 times higher effective bandwidth compared to the 2D case. Observing the highest GPU speed-ups GSp_1 and GSp_{12}, the GPU computation is more than 100× faster than the sequential CPU computation and almost 10× faster than the computation using 12 CPU threads.

Due to algorithmic differences, the second benchmark has different limiting factors than the first benchmark. Depending on the spatial dimension, the benchmark computes either lengths of 2D line segments, or 3D triangle areas. In both cases it is necessary to evaluate the `sqrt` function once per each face. Hence, we expect this benchmark to be compute-bound rather than memory-bound. Indeed, the effective bandwidth of CPU computations in Table 2.6 is significantly lower than the hardware peak performance. On the other hand, the effective bandwidth of GPU computations is still high and comparable to the values achieved in the first benchmark. In case of the triangular mesh $2D_5^{\triangle}$, GPU speed-ups GSp_1 rising above 100 and GSp_{12} rising above 10 demonstrate that the GPU can be highly utilized even on a mesh with relatively small number of cells. In case of the tetrahedral mesh $3D_5^{\triangle}$, GPU speed-ups GSp_1 and GSp_{12} rise above 300 and 30, respectively.

Finally, we would like to comment on the performance differences between the data structures implemented in TNL [A142] and MOAB [M28]. In case of the first benchmark shown in Table 2.5, the computations using MOAB were about 13× slower in 2D and about 5× slower in 3D. In case of the second benchmark shown in Table 2.6, the computations using MOAB were about 40× slower in 2D and more than 130× slower in 3D. This is surprising, because the second benchmark should be compute-bound rather than memory-bound, and the evaluation of the `sqrt` function, which is the most

Table 2.6: *Calculation of cell boundary measures* on the finest triangular mesh $2D_5^\triangle$ and finest tetrahedral mesh $3D_5^\triangle$. Comparison of computational times CT [ms], effective bandwidth EBW [GB/s], CPU speed-ups Sp_ℓ, and GPU speed-ups GSp_ℓ. Each row corresponds to different data types for `Real`, `GlobalIndex` and `LocalIndex` in the mesh configuration, which are recorded in the triplet in the "types" column: `f` (`float`), `d` (`double`), `s` (`short int`), `i` (`int`), and `l` (`long int`).

			1 th.		12 threads			GPU			
		types	CT	EBW	CT	EBW	Sp_{12}	CT	EBW	GSp_1	GSp_{12}
TNL	$2D_5^\triangle$	f, i, s	1.288	2.0	0.148	17	8.7	0.012	206	103.5	11.9
		f, l, i	1.275	3.3	0.144	29	8.9	0.013	325	99.5	11.2
		d, i, s	1.329	2.6	0.151	23	8.8	0.013	261	98.6	11.2
		d, l, i	1.288	4.0	0.145	35	8.9	0.014	358	90.0	10.1
	$3D_5^\triangle$	f, i, s	65.088	1.9	5.852	21	11.1	0.195	623	334.6	30.1
		f, l, i	64.552	3.2	6.492	32	9.9	0.275	752	234.6	23.6
		d, i, s	67.323	2.3	6.111	26	11.0	0.239	656	282.2	25.6
		d, l, i	68.515	3.5	6.109	40	11.2	0.314	771	218.1	19.4
MOAB	$2D_5^\triangle$	d, l, i	53.086	0.1	20.936	0	2.5				
	$3D_5^\triangle$	d, l, i	8839.300	0.0	5622.950	0	1.6				

expensive computation, needs to be done regardless of which data structure is used. Our investigation showed that about 80-90% of the computational time is spent in the MOAB function `get_adjacencies` that is responsible for obtaining the entities adjacent to given mesh entity. In the first benchmark, we used only the function `get_connectivity`, which is implemented efficiently in MOAB, to obtain the subvertices of given mesh cell. In the second benchmark, we need to access adjacent entities other than vertices, which is possible only via the function `get_adjacencies` that is not implemented efficiently in MOAB. On the other hand, MOAB was designed as a dynamic data structure and provides many advanced operations such as adaptive refinement of unstructured meshes, which are not possible with the data structure currently implemented in TNL.

2.2.10 Benchmark results: polygonal and polyhedral meshes

The results of the first benchmark problem are shown in Table 2.8 and the results of the second benchmark problem are shown in Table 2.9. The benchmarks were performed on all meshes shown in Table 2.3, but, for brevity, only the results for the finest mesh in 2D and 3D ($2D_5^*$ and $3D_5^*$, respectively) are shown in Tables 2.8 and 2.9.

Both benchmarks were performed with different configurations of the `Mesh` class in TNL using the same data types for template parameters as in the previous subsection. The configuration of MOAB [M28] was also the same as in the previous subsection. The results for polygonal and polyhedral meshes showed the same behavior related to the data types as we observed on triangular and tetrahedral meshes: selecting the smallest possible types for the `Real`, `GlobalIndex`, and `LocalIndex` template parameters generally leads to faster computations on both CPU and GPU compared to the larger types, and the performance of all benchmarks performed with the TNL library does not depend on other mesh configuration parameters, such as `subentityStorage` or `superentityStorage`, which can be used to disable the storage of unnecessary incidence matrices. Hence, such configuration parameters influence the dynamic size required for the mesh representation as shown in Table 2.7, but not the performance of computations using the data structure.

Similarly to the simplex meshes, the cell measure calculations shown in Table 2.8 have different characteristics on polygonal and polyhedral meshes. The areas of 2D polygons are calculated using

Table 2.7: Memory requirements [MiB] for storing the polygonal and polyhedral meshes from Table 2.3 in the data structure with different configurations. The minimal configuration includes only those incidence matrices that are needed for the benchmarks presented in this section, whereas all incidence matrices are included in the maximal configuration, even if they are unused. Each column corresponds to different data types for `Real`, `GlobalIndex` and `LocalIndex` in the mesh configuration, which are recorded in triplets in the second header row: `f` (`float`), `d` (`double`), `s` (`short int`), `i` (`int`), and `l` (`long int`).

Id.	Minimal configuration				Maximal configuration			
	f, i, s	f, l, i	d, i, s	d, l, i	f, i, s	f, l, i	d, i, s	d, l, i
$2D_1^*$	0.02	0.04	0.02	0.04	0.04	0.07	0.04	0.08
$2D_2^*$	0.07	0.12	0.08	0.13	0.13	0.24	0.14	0.25
$2D_3^*$	0.24	0.44	0.27	0.47	0.46	0.87	0.49	0.90
$2D_4^*$	0.90	1.68	1.02	1.80	1.74	3.32	1.85	3.43
$2D_5^*$	3.54	6.62	4.01	7.09	6.83	13.03	7.30	13.50
$3D_1^*$	0.22	0.41	0.24	0.43	0.65	1.25	0.66	1.27
$3D_2^*$	0.47	0.89	0.52	0.94	1.42	2.77	1.46	2.82
$3D_3^*$	3.42	6.46	3.80	6.84	10.28	20.04	10.65	20.41
$3D_4^*$	26.38	49.86	29.27	52.76	79.25	154.57	82.15	157.46
$3D_5^*$	207.55	392.34	230.32	415.11	623.71	1216.41	646.48	1239.18

the surveyor's formula [A33], which has arithmetic intensity comparable to the algorithm used for 2D triangles. On the other hand, the calculation of the volumes of 3D polyhedra is more complicated, but the complexity comes from the amount of data needed for the mesh representation rather than the complexity of operations. The algorithm is based on the decomposition of polyhedra into tetrahedra and adding the volumes of tetrahedra. This involves iterating over the faces of each polyhedron, then over the vertices of each face, and combining the vertices into suitable quadruples. Unfortunately, the vertex coordinates have to be read multiple times during the iteration from several faces and given the high average numbers of vertices per face and faces per cell given in Table 2.3, the memory access patterns may be inefficient. Consequently, the results for the finest polyhedral mesh in Table 2.8 show values of effective bandwidth that are significantly lower than the hardware peak performance. The GPU parallelization may be inefficient also due to mapping 1 CUDA thread per mesh cell, because even the finest polyhedral mesh $3D_5^*$ contains only about 3×10^5 cells. Still, the GPU speed-ups GSp_1 and GSp_{12} rise above 30 and 3.5, respectively.

The second benchmark has different limiting factors than the former. Same as for simplex meshes, calculations of the face measures (i.e., lengths of edges for polygonal meshes and areas of polygons for polyhedral meshes) involve the evaluation of the `sqrt` function once per each face. The results of CPU computations in Table 2.9 again show that the benchmark is compute-bound, but the effective bandwidth of GPU computations is still high and comparable to the values achieved on the simplex meshes. On the finest polyhedral mesh $3D_5^*$, the GPU speed-ups GSp_1 and GSp_{12} rise above 230 and 20, respectively.

Finally, the relative performance differences on polygonal and polyhedral meshes between the data structures implemented in TNL [A142] and MOAB [M28] are similar to those described in the previous subsection. In case of the first benchmark shown in Table 2.8, the computations using MOAB were about 10× slower in 2D and about 6× slower in 3D. In case of the second benchmark shown in Table 2.9, the computations using MOAB were about 35× slower in 2D and more than 40× slower in 3D. The reason for the difference observed in the second benchmark is due to the inefficient function `get_adjacencies` in MOAB as described in the previous subsection.

Table 2.8: *Calculation of cell measures* on the finest polygonal mesh $2D_5^*$ and finest polyhedral mesh $3D_5^*$. Comparison of computational times *CT* [ms], effective bandwidth *EBW* [GB/s], CPU speed-ups Sp_ℓ, and GPU speed-ups GSp_ℓ. Each row corresponds to different data types for `Real`, `GlobalIndex` and `LocalIndex` in the mesh configuration, which are recorded in the triplet in the "types" column: `f` (`float`), `d` (`double`), `s` (`short int`), `i` (`int`), and `l` (`long int`).

			1 th.		12 threads			GPU			
		types	*CT*	*EBW*	*CT*	*EBW*	Sp_{12}	*CT*	*EBW*	GSp_1	GSp_{12}
TNL	$2D_5^*$	f, i, s	0.240	5.8	0.035	40	6.8	0.012	115	19.9	2.9
		f, l, i	0.236	9.2	0.031	69	7.5	0.013	171	18.7	2.5
		d, i, s	0.295	6.7	0.040	50	7.5	0.013	148	22.0	2.9
		d, l, i	0.292	9.5	0.038	73	7.7	0.014	196	20.7	2.7
	$3D_5^*$	f, i, s	87.438	1.1	9.868	10	8.9	2.861	34	30.6	3.4
		f, l, i	119.207	1.4	14.333	12	8.3	4.004	42	29.8	3.6
		d, i, s	106.037	1.2	11.817	10	9.0	3.410	36	31.1	3.5
		d, l, i	133.311	1.5	16.329	12	8.2	4.437	44	30.0	3.7
MOAB	$2D_5^*$	d, l, i	2.664	1.0	2.416	1	1.1				
	$3D_5^*$	d, l, i	686.646	0.3	439.212	0	1.6				

Table 2.9: *Calculation of cell boundary measures* on the finest polygonal mesh $2D_5^*$ and finest polyhedral mesh $3D_5^*$. Comparison of computational times *CT* [ms], effective bandwidth *EBW* [GB/s], CPU speed-ups Sp_ℓ, and GPU speed-ups GSp_ℓ. Each row corresponds to different data types for `Real`, `GlobalIndex` and `LocalIndex` in the mesh configuration, which are recorded in the triplet in the "types" column: `f` (`float`), `d` (`double`), `s` (`short int`), `i` (`int`), and `l` (`long int`).

			1 th.		12 threads			GPU			
		types	*CT*	*EBW*	*CT*	*EBW*	Sp_{12}	*CT*	*EBW*	GSp_1	GSp_{12}
TNL	$2D_5^*$	f, i, s	1.344	2.1	0.150	19	9.0	0.013	223	105.4	11.7
		f, l, i	1.337	3.3	0.148	30	9.0	0.013	332	99.2	11.0
		d, i, s	1.379	2.9	0.152	27	9.1	0.014	289	98.4	10.8
		d, l, i	1.353	4.2	0.153	37	8.9	0.014	398	94.8	10.7
	$3D_5^*$	f, i, s	87.859	1.3	7.772	15	11.3	0.369	320	238.2	21.1
		f, l, i	93.225	2.1	7.973	24	11.7	0.511	379	182.3	15.6
		d, i, s	109.202	1.5	9.151	18	11.9	0.489	329	223.4	18.7
		d, l, i	114.673	2.1	9.307	25	12.3	0.653	362	175.5	14.2
MOAB	$2D_5^*$	d, l, i	46.439	0.1	19.438	0	2.4				
	$3D_5^*$	d, l, i	4768.330	0.0	1459.910	0	3.3				

Chapter 3

Solution of Sparse Linear Systems

This chapter is dedicated to methods for the solution of large and sparse systems of linear equations. It does not contain any novel work on the methods by the author, but provides an overview of the state of the art. High performance solvers for sparse linear systems are an important building block for the numerical methods developed in the following chapter.

The methods for the solution of linear systems are typically classified into *direct methods* [M13] and *iterative methods* [B16, B26]. Iterative methods have become preferable in many applications, especially where very large linear systems arise, because they can provide significantly lower computational cost and they are easier to implement efficiently for high performance parallel computing systems [B26]. In the following sections, we summarize the state of the art iterative methods (Section 3.1), preconditioning techniques (Section 3.2) and software packages (Section 3.3) providing ready-to-use implementations of these. In the final Section 3.4, we describe the concept of a distributed sparse matrix with more implementation details to provide a link between a distributed unstructured mesh and assembling the linear system for applications such as partial differential equation discretized on the mesh.

3.1 Iterative methods

The first iterative methods that were used for solving linear systems are the so called *stationary methods* such as the Jacobi, Gauss–Seidel, and SOR methods [B26]. Although they have been overcome by more efficient methods, they are still often used thanks to their simplicity as part of more complex methods (such as smoothers in multigrid methods).

Probably the most important class of iterative methods nowadays are the *Krylov subspace methods*, which include some of the most popular iterative methods used. The conjugate gradients (CG) method remains the obvious choice for systems with a symmetric positive-definite matrix as it is well-researched [B16, B26] and provides good convergence and error estimates. Options for symmetric indefinite systems include the MINRES and SYMMLQ methods [A145]. Similar and more general methods for systems with a non-symmetric matrix are the biconjugate gradient method (BiCG) and its stabilized transpose-free variant (BiCGstab) [A173]. GMRES [A153] is another famous and well-researched method providing many variants for implementation, such as [A13, A15, A16, A133, A176]. The flexible GMRES algorithm [A151] is a notable variant that allows more general preconditioning techniques to be applied in the GMRES algorithm. Methods based on augmentation and deflation [A46, A65, A80, A89] have been developed to improve the convergence properties of restarted GMRES. See [A157] for a thorough review of the development in Krylov subspace methods.

The aforementioned Krylov subspace methods are all suitable for modern high-performance computing platforms, either in their original formulation or with modifications improving their performance. Communication-avoiding variants have been developed for the CG, BiCG, and GMRES methods

[A86, B19, C39, C40]. There are also improved BiCGstab variants for parallel distributed systems [A50, A117, C41, A191]. Krylov subspace methods can also benefit from various optimizations developed specifically for GPU accelerators [C3, A79, C39].

The induced dimension reduction methods, denoted as IDR(s), are a new family of Krylov subspace methods for solving large non-symmetric systems of linear equations [A88, A150, A159]. Although they were reported to outperform traditional methods such as BiCGstab for several problems, they are still much less popular. In general, the research focuses on the development of more efficient preconditioners rather than new iterative methods themselves.

3.2 Preconditioning techniques

Preconditioning is the main technique to improve the robustness and efficiency of iterative methods. The technique transforms the original linear system into a different system with the same solution, but which may be easier to solve using an iterative method. Given a generic preconditioner represented by a non-singular matrix $\mathbf{M}$, the original linear system

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b} \tag{3.1}$$

can be preconditioned either from the left, leading to the preconditioned system

$$\mathbf{M}^{-1}\mathbf{A}\boldsymbol{x} = \mathbf{M}^{-1}\boldsymbol{b}, \tag{3.2}$$

or from the right, leading to

$$\mathbf{A}\mathbf{M}^{-1}\boldsymbol{y} = \boldsymbol{b}, \qquad \boldsymbol{x} = \mathbf{M}^{-1}\boldsymbol{y}, \tag{3.3}$$

where the system is solved in terms of the transformed unknown variables $\boldsymbol{y} = \mathbf{M}\boldsymbol{x}$. Finally, a split-preconditioning in the form

$$\mathbf{M}_1^{-1}\mathbf{A}\mathbf{M}_2^{-1}\boldsymbol{y} = \mathbf{M}_1^{-1}\boldsymbol{b}, \qquad \boldsymbol{x} = \mathbf{M}_2^{-1}\boldsymbol{y} \tag{3.4}$$

is also possible, where the preconditioner is $\mathbf{M} = \mathbf{M}_1\mathbf{M}_2$.

There is no consensus on which kind of preconditioning is the best and how to choose the preconditioning matrix $\mathbf{M}$. The choices typically depend on the iterative method used as well as on specific matrix properties for the given problem, however, several general-purpose preconditioners that do not require problem-specific input have been developed. Firstly, the minimal requirement imposed on the preconditioner $\mathbf{M}$ is that it must be non-singular. It may also be required to preserve the symmetry of the original matrix $\mathbf{A}$ in the preconditioned system. One way to achieve this is to use the split-preconditioning approach with $\mathbf{M} = \mathbf{M}_1\mathbf{M}_2$, where $\mathbf{M}_1$ and $\mathbf{M}_2$ are non-singular and $\mathbf{M}_1 = \mathbf{M}_2^T$. Additionally, the preconditioner should approximate $\mathbf{A}$ in some sense, such that $\mathbf{M}^{-1}\mathbf{A}$ is close to the identity matrix $\mathbf{I}$. Note that the matrix of the preconditioned system (i.e., $\mathbf{M}^{-1}\mathbf{A}$, $\mathbf{A}\mathbf{M}^{-1}$, or $\mathbf{M}_1^{-1}\mathbf{A}\mathbf{M}_2^{-1}$) is not assembled explicitly as that would be too expensive and sparsity would be lost. Instead, additional requirements are that the operation $\mathbf{M}^{-1}\boldsymbol{v}$ should be easy to apply to an arbitrary vector $\boldsymbol{v}$ and it should offer opportunities for an efficient parallel implementation. However, in practice, the last three requirements often go against each other: it is difficult to develop a preconditioner which is all *effective* (i.e., approximates the matrix $\mathbf{A}$ well), *cheap* (i.e., can be applied easily), and *parallel* at the same time.

3.2.1 State of the art

There are two general approaches to constructing preconditioners. One approach is to design specialized algorithms that are (nearly) optimal for a small class of problems. These typically utilize the knowledge of the underlying problem, such as the original partial differential equation from which

the discrete problem arose. While these preconditioners are often very effective, obtaining or using the underlying information may not be feasible or desirable. Hence, the other approach comprising general purpose, purely algebraic preconditioning techniques may be preferable. These preconditioners, while not optimal for any specific problem, achieve reasonable efficiency for a large class of problems and they are often easier to implement and use in software frameworks. In the following, we give a brief overview of the most successful techniques. Detailed reviews are available in, e.g., [A24, A77, A146].

The simplest preconditioning techniques are based on stationary methods such as Jacobi, Gauss–Seidel, SOR, or their symmetric variants [A24, B26].

Another traditional class of generic preconditioners are incomplete factorization techniques, such as incomplete Cholesky (IC) and incomplete LU (ILU), which bring ideas developed for direct solution of sparse linear systems to iterative methods. There are multiple strategies to control the fill-in by selecting the sparsity pattern of the factors either statically based on the matrix entry positions or dynamically based on the matrix entry values [A29, A124, A137, A152]. Additionally, algebraic multilevel techniques originating from domain decomposition and multigrid methods were applied also to incomplete factorization methods [A32, A95, A154] and algorithmic variants were developed to address parallelization and scalability issues [A4, A48, A49, A100, C25, C27, A177]. Several incomplete factorization algorithms were also ported to GPU accelerators [M16, M17], though their efficiency remains limited compared to other preconditioning techniques due to two sparse triangular solves per each preconditioning step. Multiple iterative or otherwise inexact techniques were proposed to address this issue [C1, C2, A130, A156]. Finally, several approaches to incrementally update an existing factorization were developed [A5, A41].

Sparse approximate inverse preconditioners provide an alternative approach to remedy problems arising when incomplete factorizations are applied to indefinite systems or systems that are not diagonally dominant [A24, B26]. They also attracted attention thanks to their potential in parallel environments [A92, A165], including GPU accelerators [A27, A28, A57, A144]. Approximate inverse preconditioners may be applied to symmetric as well as non-symmetric systems and they may be computed in a non-factored form where the preconditioner is expressed as a single matrix, or in a factored form where the preconditioner is expressed as a product of two or more matrices. For both classes there exist multiple completely different approaches to compute the approximate inverse or approximate inverse factors, the two main approaches are Frobenius norm minimization [A47, C16, A112–A114, A185] and incomplete bi-conjugation [A25, A26, A28, A36]. The sparse approximate inverse algorithms based on bi-conjugation show algebraic behavior that is similar to the incomplete LU factorizations [A30, A31]. When matrix values are changed but the sparsity pattern remains the same, it is possible to reuse an existing approximate inverse preconditioner to compute the updated approximate inverse more efficiently [A36, A185]. Several dynamic pattern selection strategies and multilevel techniques for sparse approximate inverse preconditioners have been recently investigated [A27, A66, A103, A104, A115].

Polynomial preconditioning is closely related to the development of Krylov methods. It utilizes spectral information of the linear system matrix and is favorable for parallel architectures where the sparse matrix–vector multiplication delivers high performance. Several types of polynomials can be used, the Chebyshev [C5, A9], least-squares [A87, A125], and Newton [A129] polynomials are the most common in practice.

Several non-algebraic techniques were developed for solving linear systems arising from the discretization of partial differential equations. The convergence rate of Krylov subspace methods preconditioned by the aforementioned techniques applied to these linear systems deteriorates as the systems become larger [B16, B26]. Consequently, the efficiency of these methods drops significantly due to increasing number of iterations combined with the sheer problem size. To this effect, multigrid methods [B7, B31, B33] were developed to obtain (nearly) optimal techniques in the sense that the number of iterations does not depend on the problem size, or increases only mildly. Although multigrid,

hereafter referred to as geometric multigrid (GMG), was originally developed as a method for solving partial differential equations, it was later generalized into algebraic multigrid (AMG) techniques [B24, B25, A162, B31, A174, A183] that can be applied even to problems not involving partial differential equations at all. Unfortunately, multigrid techniques are naturally communication-bound due to coarse grid computations, interpolation and restriction operations. Consequently, both geometric multigrid [A20, C21, B15, A190] and algebraic multigrid [C7, A21, A78, A128] have been recently subject of intensive research to address the efficiency and scalability challenges on modern parallel platforms, including GPU accelerators.

Domain decomposition methods [B28, B30] are another class of techniques developed to efficiently solve partial differential equations. Contrary to multigrid, domain decomposition methods were associated with parallel processing since their origin and massively parallel implementations for contemporary platforms exist of both main domain decomposition methods, BDDC and FETI [A116, A160]. Although both multigrid and domain decomposition were derived as methods for the solution of partial differential equations, they are nowadays used mostly as preconditioners to accelerate the convergence of Krylov subspace methods. Multigrid and domain decomposition methods borrowed ideas from each other during their development, which lead to the development of various algebraic multilevel preconditioning techniques [A11, A12, A62]. Similarly to multigrid, recent development of domain decomposition methods tends towards purely algebraic preconditioning techniques [A2, A172].

To conclude the section, we mention some general techniques that can be used when multiple preconditioners accelerating the solution of a linear system are available. Rather than formulating a new preconditioner that combines the features of multiple preconditioners, they can be applied together within a *multi-preconditioned* iterative method [A35, A91, A161]. More generally, when solving sequences of similar linear systems, such as those arising from the discretization of time-dependent partial differential equations, it may be possible to reuse information from previously computed problems, including preconditioners [A3, B10, B14, A157].

3.3 Software packages

The software listed here can be divided into two main groups: the first containing software dedicated specifically to providing iterative methods and/or preconditioners for linear systems, and the other group consisting of software with larger ambitions, such as scientific computing frameworks. Packages from the latter group can often be interfaced with smaller libraries from the former group to supplement their native algorithms. In both groups, we focus on open-source software with active development status. Each package is listed with a brief description and overview of the main features. In the final subsection, we summarize the current status in the Template Numerical Library that was introduced in Section 1.3.

3.3.1 Dedicated projects

Hypre [C15, C19, C20] is an open-source library of high performance preconditioners and iterative solvers for large sparse linear systems. Additionally, it provides several interfaces to create linear systems based on structured or semi-structured grids, finite element discretizations, or general linear-algebraic interface. The implemented solvers include common Krylov subspace methods such as CG, GMRES, and BiCGstab. Available preconditioners are algebraic multigrid (BoomerAMG [A184]), several variants of incomplete LU factorization [A100, C25, C27], sparse approximate inverse [A47, A104, A114], and other. The library is implemented in the C language, uses MPI [M15] for distributed computing and optionally uses CUDA [M19], HIP [M1], or SYCL [M11] for GPU acceleration.

PARALUTION [O19, M14] is an open-source library implemented in the C++ language providing parallel iterative solvers and preconditioners. Additionally, it provides plugins for several high-level

libraries, including deal.II [A17] and OpenFOAM [M9]. Among the implemented iterative solvers are stationary methods (Jacobi, Gauss–Seidel, symmetric Gauss–Seidel, SOR, SSOR), Krylov subspace solvers (CG, BiCGstab, GMRES, IDR), geometric and algebraic multigrid, and other. Available preconditioners are stationary methods, several variants of incomplete LU factorization, additive Schwarz preconditioner, and other. The library also provides multiple sparse matrix formats, including CSR and Ellpack. PARALUTION uses MPI [M15] for distributed computing and provides several back-ends for parallel execution: OpenMP [A51], CUDA [M19], OpenCL [M10]. It was also ported to the ROCm [M2] platform as rocALUTION [O27].

AmgX [A141, O25] is an open-source library of GPU accelerated algebraic multigrid and preconditioned iterative methods developed by NVIDIA. Besides algebraic multigrid, it provides also standard Krylov subspace methods (CG, BiCGstab, GMRES) and several preconditioners/smoothers (block Jacobi, Gauss–Seidel, ILU, polynomial). AmgX uses CUDA [M19] for GPU acceleration and MPI [M15] for distributed computing. The algebraic multigrid implementation is partially based on the Hypre library.

AMGCL [A58, A59] is an open-source header-only C++ library for solving large sparse linear systems with algebraic multigrid. It also provides several Krylov subspace methods (CG, BiCGstab, GMRES, IDR) and preconditioners/smoothers (Jacobi, Gauss–Seidel, Chebyshev, ILU, SPAI, and some problem-specific preconditioners for Navier–Stokes and reservoir simulations). Furthermore, it provides matrix adapters for wrapping data structures of various other libraries, including Eigen [O9], Trilinos Epetra [O8] and uBlas [O39]. AMGCL builds the algebraic multigrid hierarchy on a CPU and then transfers it to one of the provided back-ends. The currently supported acceleration frameworks are OpenMP [A51], CUDA [M19], and OpenCL [M10]. AMGCL also supports MPI [M15] for distributed computing.

Ginkgo [A6] is an open-source linear algebra library implemented in modern C++. It provides multiple sparse matrix formats, Krylov solvers (CG, BiCGstab, GMRES, IDR), and preconditioners (Jacobi, IC, ILU [A4], incomplete sparse approximate inverse [A7]). There is also a preliminary implementation of an algebraic multigrid solver and preconditioner. Ginkgo supports native parallel execution via OpenMP [A51], CUDA [M19], HIP [M1], or SYCL [M11]. Support for distributed computing via MPI [M15] in Ginkgo is a work in progress.

There are many smaller open-source software packages which serve as the basis for the development of new or improved numerical methods. The following projects are some of the most successful ones with continuing development. The PyAMG [A22] project implements the algebraic multigrid method and supporting tools in the Python language, with the help of C++ for performance-critical operations. RAPtor [O3] is a C++ implementation of parallel algebraic multigrid based on MPI [M15]. Monolis [O22] is a monolithic domain decomposition based linear system solver implemented in Fortran. Finally, BDDCML [O29, A160] is a massively parallel linear system solver based on the adaptive multilevel BDDC method.

3.3.2 Large frameworks

Trilinos [A96, C23, A97, O37] is a large collection of open-source packages with various objectives in scientific computing. The individual packages are highly inter-operable, but not strictly interdependent, i.e., each package depends on interfaces that can be satisfied by multiple other packages rather than on an explicit implementation. Trilinos packages are developed in the C++ language and include many direct and iterative linear system solvers, ILU-type preconditioners, smoothers, multigrid and domain decomposition methods. Trilinos supports distributed computing using MPI [M15], multi-core parallel execution using a variety of approaches, as well as GPU acceleration.

PETSc [M4, O2, C8] is an open-source library of data structures and algorithms for parallel solution of scientific problems modeled by partial differential equations. Unlike Trilinos, PETSc is a monolithic library developed in the C language. Its KSP module provides many parallel and sequential, direct

and iterative solvers for linear systems and the PC module provides preconditioners such as stationary methods, ILU factorizations, algebraic multigrid, or BDDC.

OpenFOAM [M9, C26] is a large open-source multiphysics package based on the finite volume method. It also provides its own implementation of various iterative methods and preconditioners for linear systems, such as stationary methods, incomplete factorizations, CG and BiCGstab methods, and geometric agglomerated algebraic multigrid. OpenFOAM itself does not support GPU acceleration, though the possibilities are being explored via external extensions [M6, C29, C33].

DUNE (Distributed and Unified Numerics Environment) [C10] is a modular C++ library for solving partial differential equations with finite element, finite volume, or finite difference methods. Its ISTL (Iterative Solver Template Library) module provides implementations of several Krylov subspace solvers and preconditioners, including algebraic multigrid.

3.3.3 Template Numerical Library

In this subsection, we summarize the current status related to the solution of linear systems in the TNL library that was introduced in Section 1.3. Unless stated otherwise, all implementations support any matrix format implemented in TNL, distributed computing with MPI [M15], and GPU acceleration using CUDA [M19].

TNL does not implement any direct method for the solution of large, dense or sparse, linear systems. The currently implemented stationary iterative methods are Jacobi and SOR (the latter works only in a shared memory environment). The currently implemented Krylov subspace methods are CG, BiCGstab, GMRES, TFQMR, IDR.

The only preconditioner fully implemented in TNL is the Jacobi (diagonal) preconditioner. Additionally, TNL provides sequential implementations of the ILU(0) and ILUT factorizations that can also be applied as a block-Jacobi preconditioner for distributed matrices.

To easily utilize the state of the art solvers and preconditioners in TNL, we started implementing *wrappers* for external libraries with more advanced features. There are currently finished wrappers for iterative methods and preconditioners from the Hypre [C15, C19, C20] library (they require the CSR matrix format with specific conventions that are described in the following section). Additionally, TNL provides a wrapper for UMFPACK [A56] which applies a direct method to solve the linear system (it requires a non-distributed matrix in the CSR format).

3.4 Distributed sparse matrix

Solving a linear system on a distributed computing platform requires a suitable partitioning of the data among the MPI ranks. The system is typically distributed in a row-wise manner such that the rows of the matrix and vectors are partitioned into non-overlapping ranges that are assigned to individual ranks. A *distributed data structure* is then used for combining the local data of each rank with information about the partitioning (e.g., assigning global indices). While the implementation of a distributed vector is straightforward, data structures for a distributed sparse matrix that provides coupling between blocks of the partitioning may be designed in different ways. This section describes data structures for distributed sparse matrices implemented in the TNL (see Section 1.3) and Hypre [C15, C19, C20] libraries.

The implementation of a distributed sparse matrix in TNL is closely bound to the distributed mesh and its MPI synchronizer described in Section 2.2.6. Each row of the global matrix corresponds to one *degree of freedom* associated to an entity of the global mesh and the matrix partitioning is determined by the partitioning of the mesh. The data owned by a particular rank is stored in a single local matrix represented by a sparse matrix data structure in TNL. Each row and column of the local matrix corresponds to an entity of the local mesh:

- Rows correspond to entities owned by the rank (i.e., not ghost entities).

- Columns may correspond to entities owned by the rank or to ghost entities. Columns owned by the rank belong to the diagonal block of the global matrix and columns corresponding to ghost entities belong to the off-diagonal blocks.

It is important to realize that *local indexing* is used in the local matrix, given by indices used in the local mesh. Hence, a local matrix with N_r rows and $N_c > N_r$ columns is stored such that the first N_r columns represent the diagonal block of the global matrix and the remaining $N_c - N_r$ columns represent the off-diagonal blocks in a compact form (there are no gaps between the off-diagonal columns as there would be if the global indexing was used). Global indices of an entry in the local matrix can be determined based on the global indices of the corresponding mesh entity. As explained in Section 2.2.6, local entities owned by the rank are contiguous and thus their global indices can be determined by adding an offset to the local indices, but global indices of ghost entities must be stored explicitly in an array. The compact representation of the local matrix is advantageous for operations such as distributed sparse matrix–vector multiplication, which can be performed in two steps:

1. Use the `DistributedMeshSynchronizer` class described in Section 2.2.6 to synchronize data corresponding to ghost entities in the input vector. Note that the distributed vector contains N_c local elements where the last $N_c - N_r$ values correspond to ghost entities.
2. Perform a sparse matrix–vector multiplication with the $N_r \times N_c$ local matrix and the local vector of N_c elements to compute the first N_r elements of the output vector. The $N_c - N_r$ ghost elements in the output vector can be synchronized with `DistributedMeshSynchronizer` if needed.

The implementation of a distributed sparse matrix in the Hypre library [C15, C19, C20] is different from TNL in several aspects. Firstly, the data structure is purely algebraic in the sense that it provides all necessary information without relying on a mesh data structure. Internally, Hypre uses the CSR format with specific conventions for the representation of sparse matrices:

- Each matrix block is represented by the `hypre_CSRMatrix` structure. An important convention is that the diagonal entry in each row must be stored as the first value of the row in the CSR format.
- The `hypre_ParCSRMatrix` structure represents a distributed matrix. Among others, it contains the attributes `diag`, `offd`, and `col_map_offd`.
- The `diag` and `offd` attributes are instances of the `hypre_CSRMatrix` structure, which represent the diagonal and off-diagonal block of the local matrix, respectively. The former is typically a square matrix for which data is always available in the local part of a distributed vector.
- The `col_map_offd` attribute is an array that provides global indices for the off-diagonal block. It is indexed by columns of the `offd` matrix.

The exchange of non-local data in Hypre is implemented based on the *assumed partition scheme* [A14] which allows for a scalable determination of global information. Overall, it can be understood as an algebraic generalization of the indexing with ghost entities in TNL.

Chapter 4

Mixed-Hybrid Finite Element Method

The content of this chapter is based on the author's previous thesis [B20] and the paper [A72], which describe *NumDwarf*, a numerical scheme for the solution of a system of partial differential equations in a general coefficient form. *NumDwarf* is based on the mixed-hybrid finite element method (MHFEM) [B6] and it was originally developed for simulating multicomponent flow and transport phenomena in porous media (see [A10, A158] for several recent applications), but it can be used for any problem whose governing equations can be written in a compatible form. Since the publication of the paper, the author incorporated an *implicit upwind scheme* for advective terms and greatly extended the computational aspects of the solver. Most notably, the solver supports MPI computations on distributed unstructured meshes.

Compared to the original computational performance study from [A72], which was computed on structured grids as well as unstructured meshes but using only CUDA and OpenMP parallelization, the presented results were computed only on unstructured meshes, but include MPI-based distributed computations using multiple GPUs and multiple CPU nodes. This extends the updated performance study from [A110] with a comparison between OpenMP and MPI for multi-core CPU parallelization on a single node, and a comparison between different solvers for sparse linear systems.

The chapter is organized as follows. First, we introduce the general formulation of the PDE system solved by *NumDwarf* in Section 4.1 and summarize the numerical scheme in Section 4.2. Section 4.3 summarizes the mathematical model of incompressible two-phase flow in porous media and Section 4.4 presents the results of a benchmark problem that was used to verify the numerical scheme and implementation by means of comparison against known semi-analytical solutions. The same benchmark problem was used to also evaluate the performance of the solver using various computational parameters. The benchmarking methodology is described in Section 4.5 and the benchmark results are presented in Section 4.6.

4.1 General formulation

The numerical scheme is implemented for a PDE system written in the form

$$\sum_{j=1}^{n} N_{i,j} \frac{\partial Z_j}{\partial t} + \sum_{j=1}^{n} \boldsymbol{u}_{i,j} \cdot \nabla Z_j + \nabla \cdot \left[m_i \left(- \sum_{j=1}^{n} \mathbf{D}_{i,j} \nabla Z_j + \boldsymbol{w}_i \right) + \sum_{j=1}^{n} Z_j \boldsymbol{a}_{i,j} \right] + \sum_{j=1}^{n} r_{i,j} Z_j = f_i \tag{4.1}$$

for $i \in \{1, \ldots, n\}$, where $\boldsymbol{Z} = [Z_1, \ldots, Z_n]^T$ is the vector of unknown functions depending on spatial coordinates $\boldsymbol{x} \in \Omega \subset \mathbb{R}^D$ and time $t \in [0, t_{\max}]$, where $D \in \{1, 2, 3\}$ denotes the spatial dimension, Ω is a polyhedral domain, and $t_{\max}$ is the final simulation time. The remaining symbols $\mathbf{N} = [N_{i,j}]_{i,j=1}^{n}$, $\boldsymbol{u} = [\boldsymbol{u}_{i,j}]_{i,j=1}^{n}$, $\boldsymbol{m} = [m_i]_{i=1}^{n}$, $\mathbf{D} = [\mathbf{D}_{i,j}]_{i,j=1}^{n}$, $\boldsymbol{w} = [\boldsymbol{w}_i]_{i=1}^{n}$, $\boldsymbol{a} = [\boldsymbol{a}_{i,j}]_{i,j=1}^{n}$, $\boldsymbol{r} = [r_{i,j}]_{i,j=1}^{n}$, $\boldsymbol{f} = [f_i]_{i=1}^{n}$ are given problem-specific coefficients. Note that the coefficients may, in general, depend on the vector of

unknown functions Z. The idea behind the notation is that coefficients dependent on Z in non-linear terms are evaluated by the numerical scheme using the values from the previous time level, whereas the symbols Z_j in Equation (4.1) are treated implicitly. See [A72, B20] for details.

The diffusive flux in Equation (4.1) is denoted as $\boldsymbol{q}_i = m_i \boldsymbol{v}_i$, where

$$\boldsymbol{v}_i = -\sum_{j=1}^{n} \mathbf{D}_{i,j} \nabla Z_j + \boldsymbol{w}_i. \tag{4.2}$$

We also denote the full flux involving diffusion and advection as $\boldsymbol{Q}_i = \boldsymbol{q}_i + \sum_{j=1}^{n} Z_j(\boldsymbol{a}_{i,j} + \boldsymbol{u}_{i,j})$.

The system of equations (4.1) must be supplemented by a suitable initial condition

$$\boldsymbol{Z}(\boldsymbol{x}, 0) = \boldsymbol{Z}_{\text{ini}}(\boldsymbol{x}), \qquad \forall x \in \Omega, \tag{4.3}$$

and boundary conditions for all $i \in \{1, \ldots, n\}$, all $\boldsymbol{x} \in \partial\Omega$, and all $t \in (0, t_{\max})$. There are multiple options, such as the Dirichlet-type fixed-value condition

$$Z_i(\boldsymbol{x}, t) = Z_i^{\mathcal{D}}, \qquad \forall x \in \Gamma_i^{\mathcal{D}} \subset \partial\Omega, \tag{4.4a}$$

or the Neumann-type condition for the diffusive flux

$$\boldsymbol{v}_i(\boldsymbol{x}, t) \cdot \boldsymbol{n}_{\partial\Omega}(\boldsymbol{x}) = v_i^{\mathcal{N}_1}, \qquad \forall x \in \Gamma_i^{\mathcal{N}_1} \subset \partial\Omega, \tag{4.4b}$$

or the Neumann-type condition for the total flux

$$\boldsymbol{Q}_i(\boldsymbol{x}, t) \cdot \boldsymbol{n}_{\partial\Omega}(\boldsymbol{x}) = Q_i^{\mathcal{N}_2}, \qquad \forall x \in \Gamma_i^{\mathcal{N}_2} \subset \partial\Omega, \tag{4.4c}$$

where $\Gamma_i^{\mathcal{D}}$, $\Gamma_i^{\mathcal{N}_1}$ and $\Gamma_i^{\mathcal{N}_2}$ denote parts of the domain boundary where the respective conditions are active and $\boldsymbol{n}_{\partial\Omega}$ denotes the outward unit normal vector.

Note that in case of non-linear problems where the coefficient $\boldsymbol{m}$ depends on the unknowns $\boldsymbol{Z}$, it may be necessary to specify fixed values of Z_i even on parts of the domain boundary $\Gamma_i^{\mathcal{N}_1}$ and $\Gamma_i^{\mathcal{N}_2}$ where the Neumann-type conditions are used to specify inflow, see [A72, B20] for details.

4.2 Numerical scheme

This section provides detailed derivation of the numerical scheme for solving Equation (4.1) based on [A72, B20]. The scheme is based on the combination of the mixed-hybrid finite element method (MHFEM) [B6] and the discontinuous Galerkin method [A8] for spatial discretization, the Euler method for temporal discretization and the semi-implicit approach of the frozen coefficients method for the linearization in time. The scheme is stabilized the mass-lumping technique and two upwind schemes for the advective terms: an *explicit upwind* that was used in [A72, B20], or an *implicit upwind* based on [A38, A149]. The explicit upwind scheme allows to employ a modification of MHFEM from [A72] to solve problems with vanishing diffusion. The scheme is locally conservative and leads to the solution of one linear system per time step.

In this work, we consider conforming unstructured homogeneous meshes denoted as $\mathcal{K}_h$ and use the lowest-order Raviart–Thomas–Nédélec space $\mathbf{RTN}_0(\mathcal{K}_h)$ for spatial discretization. The set of faces of the mesh $\mathcal{K}_h$ will be denoted as $\mathcal{E}_h$ and the set of faces of an individual element $K \in \mathcal{K}_h$ will be denoted as $\mathcal{E}_K$. Each face $E \in \mathcal{E}_h$ is either inner or lies on the domain boundary, the set of interior faces is denoted by $\mathcal{E}_h^{\text{int}}$ and the set of boundary faces is denoted by $\mathcal{E}_h^{\text{ext}}$. The number of elements of the discrete set $\mathcal{A}$ is denoted by $\#\mathcal{A}$. See Section 2.2.2 for details on the terminology related to meshes.

4.2.1 Function spaces

We assume that the scalar functions $N_{i,j}$, m_i, $r_{i,j}$, f_i and Z_j belong to the function space $L^2(\Omega)$, $\Omega \subset \mathbb{R}^D$ for all $i, j \in \{1, \dots, n\}$. We further assume that the vector functions $\boldsymbol{u}_{i,j}$, $\boldsymbol{a}_{i,j}$, $\boldsymbol{w}_i$, $\boldsymbol{v}_i$ and $\boldsymbol{q}_i$ are for all $i, j \in \{1, \dots, n\}$ elements of the function space $\mathbf{H}(\mathrm{div}, \Omega)$. The mixed variational formulation of the problem given by Equations (4.1) and (4.2) can be formally derived by multiplying Equation (4.1) by the function $\varphi \in V \equiv L^2(\Omega)$, Equation (4.2) by the function $\boldsymbol{\omega} \in \mathbf{W} \equiv \mathbf{H}(\mathrm{div}, \Omega)$, and integrating both equations over the domain Ω. The solution $Z_i \in V$, $\boldsymbol{v}_i \in \mathbf{W}$, $i \in \{1, \dots, n\}$ of the variational problem is approximated using the mixed finite element method in suitable finite-dimensional subspaces $V_h \subset V$, $\mathbf{W}_h \subset \mathbf{W}$. In this work, we use the lowest order spaces, the space of piecewise constant functions $S_0(\mathcal{K}_h)$ and the Raviart–Thomas–Nédélec space $\mathbf{RTN}_0(\mathcal{K}_h)$.

The Raviart–Thomas–Nédélec space $\mathbf{RTN}_0(\mathcal{K}_h)$ is defined in several phases [B6]. First, the local space $\mathbf{RTN}_0(K) \subset \mathbf{H}(\mathrm{div}, K)$ is defined on each element $K \in \mathcal{K}_h$ generated by the basis functions $\boldsymbol{\omega}_{K,E} \in \mathbf{H}(\mathrm{div}, K)$, $E \in \mathcal{E}_K$. The choice of the basis functions $\boldsymbol{\omega}_{K,E}$ is not unique, but in general it is convenient for them to satisfy the conditions

$$\nabla \cdot \boldsymbol{\omega}_{K,E}(\boldsymbol{x}) = \frac{1}{|K|_D}, \qquad \forall E \in \mathcal{E}_K, \forall \boldsymbol{x} \in K, \tag{4.5a}$$

$$\boldsymbol{\omega}_{K,E}(\boldsymbol{x}) \cdot \boldsymbol{n}_{K,F} = \frac{1}{|F|_{D-1}} \delta_{EF}, \qquad \forall E, F \in \mathcal{E}_K, \forall \boldsymbol{x} \in F, \tag{4.5b}$$

where $\boldsymbol{n}_{K,F}$ denotes the unit normal vector on the face $F \in \mathcal{E}_K$ oriented outward from the element K, δ_{EF} represents the Kronecker delta, and $|\cdot|_d$ denotes the d-dimensional Lebesgue measure, $d \in \{0, \dots, D\}$ and $|\cdot|_0 \equiv 1$. The particular form of the basis functions depends on the spatial dimension D and the geometry of the element K. The basis functions on rectangles, cuboids, and simplex elements satisfying the conditions (4.5) are shown in Appendix D.

In the second phase, we use the local spaces $\mathbf{RTN}_0(K)$ to define the *broken* space

$$\mathbf{RTN}_0^0(\mathcal{K}_h) = \left\{ \boldsymbol{\omega} \in [L^2(\Omega)]^D \;\middle|\; \forall K \in \mathcal{K}_h,\ \boldsymbol{\omega}|_K \in \mathbf{RTN}_0(K) \right\}. \tag{4.6}$$

However, the space $\mathbf{RTN}_0^0(\mathcal{K}_h)$ is not a subspace of the space $\mathbf{H}(\mathrm{div}, \Omega)$.

The space $\mathbf{RTN}_0(\mathcal{K}_h)$ can be defined as a subspace of the space $\mathbf{H}(\mathrm{div}, \Omega)$ by adding additional requirements. In the sense of Proposition 1.2 in [B6, p. 95], the functions $\boldsymbol{\omega} \in \mathbf{RTN}_0(\mathcal{K}_h)$ must have continuous normal traces on the inner faces of the mesh. This property can be equivalently expressed by a balance condition, which we use to define the space $\mathbf{RTN}_0(\mathcal{K}_h)$:

$$\mathbf{RTN}_0(\mathcal{K}_h) = \left\{ \boldsymbol{\omega} \in \mathbf{RTN}_0^0(\mathcal{K}_h) \;\middle|\; \forall E \in \mathcal{E}_h^{\mathrm{int}}, E \in \mathcal{E}_{K_1} \cap \mathcal{E}_{K_2}, \int_E \boldsymbol{\omega}|_{K_1} \cdot \boldsymbol{n}_{K_1,E} + \int_E \boldsymbol{\omega}|_{K_2} \cdot \boldsymbol{n}_{K_2,E} = 0 \right\}. \tag{4.7}$$

4.2.2 Approximation of vector functions

The vector functions $\boldsymbol{v}_i$, $\boldsymbol{w}_i$, $\boldsymbol{q}_i$, $\boldsymbol{a}_{i,j}$ and $\boldsymbol{u}_{i,j}$ are approximated in the space $\mathbf{RTN}_0(\mathcal{K}_h)$. The expression in the basis of the broken space $\mathbf{RTN}_0^0(\mathcal{K}_h)$ gives

$$\boldsymbol{v}_i(\boldsymbol{x}, t) \approx \sum_{K \in \mathcal{K}_h} \sum_{E \in \mathcal{E}_K} v_{i,K,E}(t) \boldsymbol{\omega}_{K,E}(\boldsymbol{x}), \tag{4.8a}$$

$$\boldsymbol{w}_i(\boldsymbol{x}, t) \approx \sum_{K \in \mathcal{K}_h} \sum_{E \in \mathcal{E}_K} w_{i,K,E}(t) \boldsymbol{\omega}_{K,E}(\boldsymbol{x}), \tag{4.8b}$$

$$\boldsymbol{q}_i(\boldsymbol{x},t) \approx \sum_{K\in\mathcal{K}_h}\sum_{E\in\mathcal{E}_K} q_{i,K,E}(t)\boldsymbol{\omega}_{K,E}(\boldsymbol{x}), \tag{4.8c}$$

$$\boldsymbol{a}_{i,j}(\boldsymbol{x},t) \approx \sum_{K\in\mathcal{K}_h}\sum_{E\in\mathcal{E}_K} a_{i,j,K,E}(t)\boldsymbol{\omega}_{K,E}(\boldsymbol{x}), \tag{4.8d}$$

$$\boldsymbol{u}_{i,j}(\boldsymbol{x},t) \approx \sum_{K\in\mathcal{K}_h}\sum_{E\in\mathcal{E}_K} u_{i,j,K,E}(t)\boldsymbol{\omega}_{K,E}(\boldsymbol{x}), \tag{4.8e}$$

for all $\boldsymbol{x} \in \Omega$ and $t \in (0, t_{\max})$, where $\boldsymbol{\omega}_{K,E}$ are the basis functions of the space $\mathbf{RTN}_0(K)$. Let $\boldsymbol{n}_{K,E}$ denote the unit normal vector on the face $E \in \mathcal{E}_K$ oriented outward from the element K. By integrating the normal trace $\boldsymbol{v}_i|_K \cdot \boldsymbol{n}_{K,E}$ over the face $E \in \mathcal{E}_K$ and using the properties of the basis functions (4.5), we obtain

$$\int_E \boldsymbol{v}_i|_K \cdot \boldsymbol{n}_{K,E} \approx \int_E \sum_{F\in\mathcal{E}_K} v_{i,K,F}\boldsymbol{\omega}_{K,F}\cdot\boldsymbol{n}_{K,E} = v_{i,K,E}. \tag{4.9}$$

Analogous relations hold for the functions $\boldsymbol{w}_i$, $\boldsymbol{q}_i$, $\boldsymbol{a}_{i,j}$ and $\boldsymbol{u}_{i,j}$. Thus, the coefficients $v_{i,K,E}$, $w_{i,K,E}$, $q_{i,K,E}$, $u_{i,j,K,E}$ and $a_{i,j,K,E}$ appearing in Equation (4.8) correspond to the numerical flux of the corresponding function through the face $E \in \mathcal{E}_K$ in the direction of the outward normal vector with respect to the element K. In order to obtain an approximation of the functions in the space $\mathbf{RTN}_0(\mathcal{K}_h)$, the coefficients must satisfy the continuity condition for the normal traces on interior faces of the mesh (see Equation (4.7)). Thus, on the face $E \in \mathcal{E}_h^{\text{int}}$ separating the elements K_1 and K_2, the following conditions must hold:

$$v_{i,K_1,E} + v_{i,K_2,E} = 0, \tag{4.10a}$$

$$w_{i,K_1,E} + w_{i,K_2,E} = 0, \tag{4.10b}$$

$$q_{i,K_1,E} + q_{i,K_2,E} = 0, \tag{4.10c}$$

$$a_{i,j,K_1,E} + a_{i,j,K_2,E} = 0, \tag{4.10d}$$

$$u_{i,j,K_1,E} + u_{i,j,K_2,E} = 0. \tag{4.10e}$$

These conditions correspond to the assumption of zero source terms on the face E and thus represent the local balance of the quantities $\boldsymbol{v}_i$, $\boldsymbol{w}_i$, $\boldsymbol{q}_i$, $\boldsymbol{a}_{i,j}$ and $\boldsymbol{u}_{i,j}$.

4.2.3 Approximation of scalar functions

For the approximation of scalar functions, we consider a finite-dimensional space $V_h \equiv S_0(\mathcal{K}_h)$ of piecewise constant functions on elements $K \in \mathcal{K}_h$ generated by the basis functions

$$\varphi_K(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{x} \in K, \\ 0 & \text{otherwise.} \end{cases} \tag{4.11}$$

Important properties of the basis functions of the space $S_0(\mathcal{K}_h)$ are:

$$\operatorname{supp}\varphi_K = K, \qquad \int_\Omega \varphi_K = |K|_D, \qquad \nabla\varphi_K = \mathbf{0} \text{ in } K. \tag{4.12}$$

The scalar coefficients $N_{i,j}$, m_i, $r_{i,j}$, and f_i in Equation (4.1) are approximated by projection into the space $S_0(\mathcal{K}_h)$, i.e.

$$Z_j(\boldsymbol{x},t) \approx \sum_{K\in\mathcal{K}_h} Z_{j,K}(t)\varphi_K(\boldsymbol{x}), \tag{4.13a}$$

$$N_{i,j}(\boldsymbol{x},t) \approx \sum_{K\in\mathcal{K}_h} N_{i,j,K}(t)\varphi_K(\boldsymbol{x}), \tag{4.13b}$$

$$m_i(\boldsymbol{x},t) \approx \sum_{K\in\mathcal{K}_h} m_{i,K}(t)\varphi_K(\boldsymbol{x}), \tag{4.13c}$$

$$r_{i,j}(\boldsymbol{x},t) \approx \sum_{K\in\mathcal{K}_h} r_{i,j,K}(t)\varphi_K(\boldsymbol{x}), \tag{4.13d}$$

$$f_i(\boldsymbol{x},t) \approx \sum_{K\in\mathcal{K}_h} f_{i,K}(t)\varphi_K(\boldsymbol{x}), \tag{4.13e}$$

for all $\boldsymbol{x} \in \Omega$ and $t \in (0, t_{\max})$. The coefficients $Z_{j,K}$, $N_{i,j,K}$, $m_{i,K}$, $r_{i,j,K}$ and $f_{i,K}$ appearing in Equation (4.13) can be interpreted as the mean values of the respective quantity on the element $K \in \mathcal{K}_h$ at given time. The variational approximation in the space of piecewise constant functions thus corresponds to the cell-centered finite volume method (CFVM).

4.2.4 Discretization of the diffusive term

Now we discretize the diffusive term given by Equation (4.2). Let us define the partial velocity $\boldsymbol{v}_{i,j}$ for $i, j \in \{1, \dots, n\}$ and assume that $\boldsymbol{v}_{i,j}$ belongs to the space $\mathbf{H}(\mathrm{div}, \Omega)$:

$$\boldsymbol{v}_{i,j} = -\mathbf{D}_{i,j}\nabla Z_j. \tag{4.14}$$

The approximation of $\boldsymbol{v}_{i,j}$ in the local space $\mathbf{RTN}_0(K)$ can be written by

$$\boldsymbol{v}_{i,j}|_K \approx \sum_{E\in\mathcal{E}_K} v_{i,j,K,E}\boldsymbol{\omega}_{K,E}, \tag{4.15}$$

where $\boldsymbol{\omega}_{K,E}$ denotes the basis function of the space $\mathbf{RTN}_0(K)$. First, let us assume that the tensor $\mathbf{D}_{i,j}$ is symmetric and positive definite on the element K. Then we can multiply Equation (4.14) by its inverse and require the resulting equality to hold in the weak sense

$$\sum_{E\in\mathcal{E}_K} v_{i,j,K,E}\int_K \boldsymbol{\omega}_{K,F}^T\mathbf{D}_{i,j}^{-1}\boldsymbol{\omega}_{K,E} = -\int_K \boldsymbol{\omega}_{K,F}^T\nabla Z_j, \qquad \forall F \in \mathcal{E}_K, \tag{4.16}$$

where $\boldsymbol{\omega}_{K,F}$ denotes the basis function of the space $\mathbf{RTN}_0(K)$. Next, to modify the right-hand side of Equation (4.16), we use the Green's formula, Equation (4.13a), the properties of the basis functions (4.5), and Equation (4.12):

$$-\int_K \boldsymbol{\omega}_{K,F}^T\nabla Z_j = -\int_{\partial K} Z_j\boldsymbol{\omega}_{K,F}^T\boldsymbol{n}_{\partial K} + \int_K Z_j\nabla\cdot\boldsymbol{\omega}_{K,F} = -Z_{j,F} + Z_{j,K}, \tag{4.17}$$

where the coefficients $Z_{j,F}$ represent new degrees of freedom for the variable Z_j on the faces of the mesh. According to [B6, ch. 5.1.2], the coefficients $Z_{j,F}$ play the role of Lagrange multipliers in the scheme, which allow to formulate the variational problem in Equation (4.16) locally without requiring the continuity of the normal traces of the basis functions $\boldsymbol{\omega}_{K,F}$. When the lowest order spaces are used, the coefficients $Z_{j,F}$ can be interpreted as the mean value of the quantity Z_j on the face F.

Using the notation $B_{i,j,K,E,F} = \int_K \boldsymbol{\omega}_{K,E}^T\mathbf{D}_{i,j}^{-1}\boldsymbol{\omega}_{K,F}$, Equation (4.16) can be written as

$$\sum_{E\in\partial K} v_{i,j,K,E}B_{i,j,K,E,F} = Z_{j,K} - Z_{j,F}. \tag{4.18}$$

The coefficients $B_{i,j,K,E,F}$ form a symmetric and positive definite matrix $\mathbf{B}_{i,j,K} = \left[B_{i,j,K,E,F}\right]_{E,F\in\mathcal{E}_K}$ and its inverse is denoted $\mathbf{b}_{i,j,K} = \left[b_{i,j,K,E,F}\right]_{E,F\in\mathcal{E}_K}$. The matrices $\mathbf{B}_{i,j,K}$ and $\mathbf{b}_{i,j,K}$ for different types of elements are given explicitly in Appendix E. Next, Equation (4.18) can be inverted as

$$v_{i,j,K,E} = b_{i,j,K,E} Z_{j,K} - \sum_{F\in\mathcal{E}_K} b_{i,j,K,E,F} Z_{j,F}, \tag{4.19}$$

where $b_{i,j,K,E} = \sum_{F\in\mathcal{E}_K} b_{i,j,K,E,F}$.

If the tensor $\mathbf{D}_{i,j}$ is zero on the element K, the projection $\boldsymbol{v}_{i,j}|_K$ is also zero, and hence the coefficients $v_{i,j,K,E}$ must be zero and we can define $b_{i,j,K,E,F} = 0$ for all $E, F \in \mathcal{E}_K$. In both cases, the coefficients $v_{i,K,E}$ can be expressed using $Z_{j,K}$ and $Z_{j,F}$ as

$$v_{i,K,E} = \sum_{j=1}^{n} \left(b_{i,j,K,E} Z_{j,K} - \sum_{F\in\mathcal{E}_K} b_{i,j,K,E,F} Z_{j,F} \right) + w_{i,K,E}. \tag{4.20}$$

4.2.5 Discretization of the scalar equation

The variational formulation of the problem given by Equation (4.1) is obtained by multiplying the i-th equation by the function $\varphi \in V \equiv L^2(\Omega)$ and integrating over the domain Ω:

$$\sum_{j=1}^{n} \int_\Omega N_{i,j} \frac{\partial Z_j}{\partial t} \varphi + \sum_{j=1}^{n} \int_\Omega \boldsymbol{u}_{i,j} \cdot \nabla Z_j \varphi + \int_\Omega \nabla \cdot \left(m_i \boldsymbol{v}_i + \sum_{j=1}^{n} Z_j \boldsymbol{a}_{i,j} \right) \varphi + \sum_{j=1}^{n} \int_\Omega r_{i,j} Z_j \varphi = \int_\Omega f_i \varphi. \tag{4.21}$$

We use the approach of the discontinuous Galerkin method (DGM) to define the approximate solution of this problem. As already mentioned, we switch from the space V to the finite-dimensional subspace $V_h \equiv S_0(\mathcal{K}_h)$. By setting $\varphi = \varphi_K$ and using Equations (4.12) and (4.13), we obtain the following substitutions for terms containing only scalar coefficients:

$$\int_\Omega N_{i,j} \frac{\partial Z_j}{\partial t} \varphi_K \approx |K|_D N_{i,j,K} \frac{\mathrm{d}Z_{j,K}}{\mathrm{d}t}, \tag{4.22a}$$

$$\int_\Omega r_{i,j} Z_j \varphi_K \approx |K|_D r_{i,j,K} Z_{j,K}, \tag{4.22b}$$

$$\int_\Omega f_i \varphi_K \approx |K|_D f_{i,K}. \tag{4.22c}$$

To modify the terms containing vector coefficients, we use Equation (4.8), the properties of the basis functions in Equations (4.5) and (4.12), and the Green's formula:

$$\begin{aligned} \int_\Omega \boldsymbol{u}_{i,j} \cdot \nabla Z_j \varphi_K &= \int_{\partial K} Z_j \varphi_K \boldsymbol{u}_{i,j} \cdot \boldsymbol{n}_{\partial K} - \int_K Z_j \varphi_K \nabla \cdot \boldsymbol{u}_{i,j} - \underbrace{\int_K Z_j \boldsymbol{u}_{i,j} \cdot \nabla \varphi_K}_{=0} \\ &\approx \sum_{E\in\mathcal{E}_K} Z^{\mathrm{upw}}_{i,j,K,E} u_{i,j,K,E} - Z_{j,K} \sum_{E\in\mathcal{E}_K} u_{i,j,K,E}, \end{aligned} \tag{4.23a}$$

$$\int_\Omega \nabla\cdot(m_i\boldsymbol{v}_i)\,\varphi_K = \int_{\partial K} m_i\varphi_K\boldsymbol{v}_i\cdot\boldsymbol{n}_{\partial K} - \underbrace{\int_K m_i\boldsymbol{v}_i\cdot\nabla\varphi_K}_{=0} \approx \sum_{E\in\mathcal{E}_K} m_{i,E}^{\mathrm{upw}} v_{i,K,E}, \tag{4.23b}$$

$$\int_\Omega \nabla\cdot\left(Z_j\boldsymbol{a}_{i,j}\right)\varphi_K = \int_{\partial K} Z_j\varphi_K\boldsymbol{a}_{i,j}\cdot\boldsymbol{n}_{\partial K} - \underbrace{\int_K Z_j\boldsymbol{a}_{i,j}\cdot\nabla\varphi_K}_{=0} \approx \sum_{E\in\mathcal{E}_K} Z_{i,j,K,E}^{\mathrm{upw}} a_{i,j,K,E}, \tag{4.23c}$$

where $\boldsymbol{n}_{K,E}$ denotes the unit normal vector on the face E oriented outward from the element K and the symbols $m_{i,E}^{\mathrm{upw}}(t)$ and $Z_{i,j,K,E}^{\mathrm{upw}}(t)$ represent still unspecified values of the quantities m_i and Z_j on the face E, which will be defined in Section 4.2.7.

Using the above substitutions in Equation (4.21), we obtain the following semi-discrete scheme:

$$\begin{aligned}&|K|_D\sum_{j=1}^n N_{i,j,K}(t)\frac{\mathrm{d}Z_{j,K}(t)}{\mathrm{d}t} + \sum_{j=1}^n\sum_{E\in\mathcal{E}_K} Z_{i,j,K,E}^{\mathrm{upw}}(t)\left(a_{i,j,K,E}(t)+u_{i,j,K,E}(t)\right) + \sum_{E\in\mathcal{E}_K} m_{i,E}^{\mathrm{upw}}(t)v_{i,K,E}(t)\\ &+\sum_{j=1}^n\left(|K|_D r_{i,j,K}(t) - \sum_{E\in\mathcal{E}_K} u_{i,j,K,E}(t)\right)Z_{j,K}(t) = |K|_D f_{i,K}(t).\end{aligned} \tag{4.24}$$

The interval $[0, t_{\max}]$ is discretized with a finite set of time levels

$$\mathcal{T} = \left\{0 = t_0 < t_1 < \ldots < t_{N_t} = t_{\max}\right\}, \tag{4.25}$$

where the integer N_t denotes the number of time steps and $\Delta t_k = t_{k+1} - t_k$, $k \in \{0,\ldots,N_t-1\}$ denotes the length of the time step. In the following, we denote the values of the time-dependent functions evaluated on the time level $t_k \in \mathcal{T}$ by the superscript k.

The nonlinear semi-discrete scheme (4.24) is discretized in time using the backward Euler method and linearized using the method of frozen coefficients. The resulting discrete scheme

$$\begin{aligned}&\frac{|K|_D}{\Delta t_k}\sum_{j=1}^n N_{i,j,K}^k\left(Z_{j,K}^{k+1}-Z_{j,K}^k\right) + \sum_{j=1}^n\sum_{E\in\mathcal{E}_K} Z_{i,j,K,E}^{\mathrm{upw}}\left(a_{i,j,K,E}^k + u_{i,j,K,E}^k\right) + \sum_{E\in\mathcal{E}_K} m_{i,E}^{k,\mathrm{upw}} v_{i,K,E}^{k+1}\\ &+\sum_{j=1}^n\left(|K|_D r_{i,j,K}^k - \sum_{E\in\mathcal{E}_K} u_{i,j,K,E}^k\right)Z_{j,K}^{k+1} = |K|_D f_{i,K}^k\end{aligned} \tag{4.26}$$

is valid for all $i \in \{1,\ldots,n\}$, all elements $K \in \mathcal{K}_h$, and time levels $k \in \{0,\ldots,N_t-1\}$.

In order to use the coefficients $v_{i,K,E}$ from Equation (4.20) in Equation (4.26), they need to be expressed at time $t = t_{k+1}$. We linearize the nonlinear Equation (4.20) in time again using the method of frozen coefficients

$$v_{i,K,E}^{k+1} = \sum_{j=1}^n\left(b_{i,j,K,E}Z_{j,K}^{k+1} - \sum_{F\in\mathcal{E}_K} b_{i,j,K,E,F}Z_{j,F}^{k+1}\right) + w_{i,K,E}^k, \tag{4.27}$$

where the values of $b_{i,j,K,E}$ and $b_{i,j,K,E,F}$ are evaluated at the time level t_k, but the superscript k is omitted for clarity.

4.2.6 Discretization of boundary conditions

For the discretization of boundary conditions, we assume that a condition of exactly one type is given on all boundary faces $E \in \mathcal{E}_h^{\mathrm{ext}}$, i.e., for all $j \in \{1,\ldots,n\}$, we have either $E \subset \Gamma_j^{\mathcal{D}}$ or $E \subset \Gamma_j^{\mathcal{N}_1}$ or $E \subset \Gamma_j^{\mathcal{N}_2}$.

According to Equation (4.4a), we can directly prescribe the value of the function Z_j on the Dirichlet part of the boundary $\Gamma_j^{\mathcal{D}}$. In the discrete formulation, we directly obtain the mean value of $Z_{j,E}^{k+1}$ on the boundary face $E \in \mathcal{E}_h^{\text{ext}}$:

$$Z_{j,E}^{k+1} = \frac{1}{|E|_{D-1}} \int_E Z_j^{\mathcal{D}}(\boldsymbol{x}, t_{k+1}) \equiv Z_{j,E}^{\mathcal{D}}(t_{k+1}). \tag{4.28}$$

The discretization of the Neumann condition for the diffusive flux from Equation (4.4b) on the boundary face $E \in \mathcal{E}_K$ leads to

$$v_{i,K,E}^{k+1} = \int_E v_i^{\mathcal{N}_1}(\boldsymbol{x}, t_{k+1}) \equiv v_{i,E}^{\mathcal{N}_1}(t_{k+1}), \tag{4.29}$$

which according to Equation (4.9) represents the numerical flux of the function $\boldsymbol{v}_i$ through the face E in the outward normal direction with respect to the element K.

The discretization of the Neumann condition for the total flux from Equation (4.4c) on the boundary face $E \in \mathcal{E}_K$ leads to

$$m_{i,E}^{k,\text{upw}} v_{i,K,E}^{k+1} + \sum_{j=1}^{n} Z_{i,j,K,E}^{k+1,\text{upw}} \left(a_{i,j,K,E}^{k} + u_{i,j,K,E}^{k} \right) = Q_{i,E}^{\mathcal{N}_2}(t_{k+1}), \tag{4.30}$$

where the terms $m_{i,E}^{k,\text{upw}}$ and $Z_{i,j,K,E}^{k+1,\text{upw}}$ are specified in the next subsection.

4.2.7 Upwind approximation for advective terms

We can now specify the coefficients $m_{i,E}^{k,\text{upw}}$ and $Z_{i,j,K,E}^{\text{upw}}$. To ensure the stability of the scheme for advection problems, we use the *upwind* stabilization technique [B22]. The coefficients $m_{i,E}^{k,\text{upw}}$ and $Z_{i,j,K,E}^{\text{upw}}$ are defined based on the direction of the numerical flux of the functions $\boldsymbol{v}_i$ and $\boldsymbol{a}_{i,j} + \boldsymbol{u}_{i,j}$, respectively, through the face $E \in \mathcal{E}_h$.

We use the *explicit upwind scheme* to define the coefficient $m_{i,E}^{k,\text{upw}}$ as

$$m_{i,E}^{k,\text{upw}} = \begin{cases} m_{i,K_1}^{k} & \text{if } v_{i,K_1,E}^{k} \geq 0, \\ m_{i,K_2}^{k} & \text{otherwise,} \end{cases} \tag{4.31a}$$

for all interior faces $E \in \mathcal{E}_h^{\text{int}}$ with $E \in \mathcal{E}_{K_1} \cap \mathcal{E}_{K_2}$, and

$$m_{i,E}^{k,\text{upw}} = \begin{cases} m_{i,E}^{\mathcal{D}}(t_k) & \text{if } E \in \mathcal{E}_{h,i}^{\text{ext,in}}(t_k), \\ m_{i,K_1}^{k} & \text{otherwise,} \end{cases} \tag{4.31b}$$

for all boundary faces $E \in \mathcal{E}_h^{\text{ext}}$ with $E \in \mathcal{E}_{K_1}$. The purpose of the abstract set $\mathcal{E}_{h,i}^{\text{ext,in}}(t_k) \subset \mathcal{E}_h^{\text{ext}}$ appearing in Equation (4.31b) will be explained later.

Two schemes for the coefficient $Z_{i,j,K,E}^{\text{upw}}$ are used in this work. In the *explicit upwind scheme*, it is defined as

$$Z_{i,j,K_1,E}^{\text{upw}} \equiv Z_{i,j,K_1,E}^{k,\text{upw}} = \begin{cases} Z_{j,K_1}^{k} & \text{if } a_{i,j,K_1,E}^{k} + u_{i,j,K_1,E}^{k} \geq 0, \\ Z_{j,K_2}^{k} & \text{otherwise,} \end{cases} \tag{4.32a}$$

for all interior faces $E \in \mathcal{E}_h^{\text{int}}$ with $E \in \mathcal{E}_{K_1} \cap \mathcal{E}_{K_2}$, and

$$Z_{i,j,K_1,E}^{\text{upw}} \equiv Z_{i,j,K_1,E}^{k,\text{upw}} = \begin{cases} Z_{j,E}^{\mathcal{D}}(t_k) & \text{if } a_{i,j,K_1,E}^{k} + u_{i,j,K_1,E}^{k} < 0, \\ Z_{j,K_1}^{k} & \text{otherwise,} \end{cases} \tag{4.32b}$$

for all boundary faces $E \in \mathcal{E}_h^{\text{ext}}$ with $E \in \mathcal{E}_{K_1}$. Note that this scheme is symmetric, i.e., $Z^{k,\text{upw}}_{i,j,K_1,E} = Z^{k,\text{upw}}_{i,j,K_2,E}$. This scheme was used also in [A72].

In the second scheme, hereafter referred to as *implicit upwind scheme*, the coefficient $Z^{\text{upw}}_{i,j,K,E}$ is defined as

$$Z^{\text{upw}}_{i,j,K_1,E} \equiv Z^{k+1,\text{upw}}_{i,j,K_1,E} = \begin{cases} Z^{k+1}_{j,K_1} & \text{if } a^k_{i,j,K_1,E} + u^k_{i,j,K_1,E} \geq 0, \\ 2Z^{k+1}_{j,E} - Z^{k+1}_{j,K_1} & \text{otherwise,} \end{cases} \tag{4.33a}$$

for all interior faces $E \in \mathcal{E}_h^{\text{int}}$ with $E \in \mathcal{E}_{K_1} \cap \mathcal{E}_{K_2}$, and

$$Z^{\text{upw}}_{i,j,K_1,E} \equiv Z^{k+1,\text{upw}}_{i,j,K_1,E} = \begin{cases} Z^{\mathcal{D}}_{j,E}(t_{k+1}) & \text{if } a^k_{i,j,K_1,E} + u^k_{i,j,K_1,E} < 0, \\ Z^{k+1}_{j,K_1} & \text{otherwise,} \end{cases} \tag{4.33b}$$

for all boundary faces $E \in \mathcal{E}_h^{\text{ext}}$ with $E \in \mathcal{E}_{K_1}$. This scheme was introduced in [A38, A149]. The motivation for this choice is that since $Z_{j,E}$ provides an approximation to the mean value $\frac{1}{2}(Z_{j,K_1}+Z_{j,K_2})$, then $2Z_{j,E} - Z_{j,K_1}$ approximates Z_{j,K_2}. Avoiding Z_{j,K_2} in the definition of the upwinded term allows to proceed with the hybridization procedure in MHFEM even when the terms are treated implicitly. Note that the scheme is not symmetric, i.e., $Z^{k+1,\text{upw}}_{i,j,K_1,E} \neq Z^{k+1,\text{upw}}_{i,j,K_2,E}$. However, it is not a conceptual problem since MHFEM leads to systems of linear algebraic equations with non-symmetric matrices even when the explicit upwind scheme is applied.

We assume that the values $m^{\mathcal{D}}_{i,E}$ and $Z^{\mathcal{D}}_{j,E}$ representing the prescribed traces of the corresponding quantities on the outer face E are available whenever they are needed according to Equations (4.31) to (4.33). The availability of the values $m^{\mathcal{D}}_{i,E}$ can be further configured for a particular problem via the abstract set $\mathcal{E}^{\text{ext,in}}_{h,i}(t_k) \subset \mathcal{E}^{\text{ext}}_h$ that appears in (4.31b). In the simplest case, the set can be chosen as $\mathcal{E}^{\text{ext,in}}_{h,i}(t_k) = \left\{E \in \mathcal{E}^{\text{ext}}_h \mid v^{\mathcal{N}_1}_{i,E}(t_k) < 0\right\}$ using a condition similar to Equation (4.32b). In case the problem represents a coupled system of non-linear partial differential equations, the condition used in the definition of the set may be more complicated. For example, the problem of two-phase flow in porous media that will be described later in Section 4.3 requires the set to be chosen as $\mathcal{E}^{\text{ext,in}}_{h,i}(t_k) = \left\{E \in \mathcal{E}^{\text{ext}}_h \mid v^{\mathcal{N}_1}_{j,E}(t_k) < 0 \text{ for some } j \in \{1,2\}\right\}$ for $i \in \{1,2\}$, where the condition detects inflow of any of the two phases denoted by the index j. See [B20] for details.

4.2.8 Balance conditions for interior faces

Using Equation (4.27) in Equation (4.26) for $K \in \mathcal{K}_h$, $i \in \{1,\dots,n\}$ leads to a system of $n \times \#\mathcal{K}_h$ linear equations with $n \times (\#\mathcal{K}_h + \#\mathcal{E}_h)$ unknowns $Z^{k+1}_{j,K}$ and $Z^{k+1}_{j,E}$, where $j \in \{1,\dots,n\}$, $K \in \mathcal{K}_h$, $E \in \mathcal{E}_h$. Assuming that the diffusion tensor $\mathbf{D}$ is non-zero, the system can be closed by adding $n \times \#\mathcal{E}^{\text{int}}_h$ equations representing balance conditions for the interior faces of the mesh and $n \times \#\mathcal{E}^{\text{ext}}_h$ equations representing boundary conditions given by Equations (4.28) to (4.30).

Assuming zero sources on the interior faces of the mesh, the standard mass conservation law corresponds to the local balance of the flux $\boldsymbol{Q}_i = m_i \boldsymbol{v}_i + \sum_{j=1}^{n} Z_j(\boldsymbol{a}_{i,j} + \boldsymbol{u}_{i,j})$. The balance conditions for interior faces can be written in the discrete form as

$$\sum_{\ell=1}^{2} \left[m^k_{i,K_\ell,E} v^{k+1}_{i,K_\ell,E} + \sum_{j=1}^{n} Z^{\text{upw}}_{i,j,K_\ell,E} \left(a^k_{i,j,K_\ell,E} + u^k_{i,j,K_\ell,E} \right) \right] = 0 \tag{4.34}$$

for all $i \in \{1,\dots,n\}$ and all interior faces $E \in \mathcal{E}^{\text{int}}_h$ such that $E \in \mathcal{E}_{K_1} \cap \mathcal{E}_{K_2}$. The following steps depend on which upwind technique is used. In case the symmetric explicit upwind scheme given by Equation (4.32) is used, the term $\left(a^k_{i,j,K_\ell,E} + u^k_{i,j,K_\ell,E} \right)$ is multiplied by a common value that does

not depend on K_ℓ. Furthermore, due to Equation (4.10), the advection terms in Equation (4.34) cancel out. Thus, the resulting conditions enforce only the balance of the diffusive flux $\boldsymbol{q}_i = m_i \boldsymbol{v}_i$, which corresponds to the scheme described in [A72]. Substituting Equation (4.27) into Equation (4.34) leads to

$$\sum_{\ell=1}^{2} m_{i,K_\ell,E}^{k} \left[\sum_{j=1}^{n} \left(b_{i,j,K_\ell,E} Z_{j,K_\ell}^{k+1} - \sum_{F \in \mathcal{E}_{K_\ell}} b_{i,j,K_\ell,E,F} Z_{j,F}^{k+1} \right) + w_{i,K_\ell,E}^{k} \right] = 0 \tag{4.35}$$

for all $i \in \{1, \dots, n\}$ and all interior faces $E \in \mathcal{E}_h^{\text{int}}$ such that $E \in \mathcal{E}_{K_1} \cap \mathcal{E}_{K_2}$. A problem arises when the mobility m_i is zero, which can occur, for example, when modeling a multiphase flow in situations where one of the phases does not occur or disappears at a given location. If $m_{i,K_\ell,E}^{k} = 0$ for $\ell \in \{1, 2\}$, then the global system of linear equations closed with Equation (4.35) is singular. Therefore, we use a modified procedure inspired by the work of [A73, A74, A99] and proposed in [A72], and replace the mobility $m_{i,K_\ell,E}^{k}$ in Equation (4.35) with a common value $m_{i,E}^{k,\text{upw}}$ for both elements, i.e., $m_{i,E}^{k,\text{upw}} = m_{i,K_\ell,E}^{k}$ for $\ell \in \{1, 2\}$. This allows us to eliminate the mobility $m_{i,E}^{k,\text{upw}}$ from Equation (4.35), assuming that $m_{i,E}^{k,\text{upw}} > 0$. This leads us to balance the quantity $\boldsymbol{v}_i$ instead of the flux $\boldsymbol{q}_i$, which we use for all values of mobility, including $m_{i,E}^{k,\text{upw}} = 0$. The resulting balance equation is therefore

$$\sum_{\ell=1}^{2} \left[\sum_{j=1}^{n} \left(b_{i,j,K_\ell,E} Z_{j,K_\ell}^{k+1} - \sum_{F \in \mathcal{E}_{K_\ell}} b_{i,j,K_\ell,E,F} Z_{j,F}^{k+1} \right) + w_{i,K_\ell,E}^{k} \right] = 0 \tag{4.36}$$

for all $i \in \{1, \dots, n\}$ and all interior faces $E \in \mathcal{E}_h^{\text{int}}$ such that $E \in \mathcal{E}_{K_1} \cap \mathcal{E}_{K_2}$. Note that Equation (4.36) can be obtained alternatively by substituting Equation (4.27) into (4.10a).

In case the non-symmetric implicit upwind scheme given by Equation (4.33) is used, the balance Equation (4.34) must be considered as a whole since the advection terms no longer cancel out. Similarly to the above, we can replace the term $m_{i,K_\ell,E}^{k}$ with the common value $m_{i,E}^{k,\text{upw}}$ for both elements to obtain the balance condition

$$\sum_{\ell=1}^{2} \left\{ m_{i,E}^{k,\text{upw}} \left[\sum_{j=1}^{n} \left(b_{i,j,K_\ell,E} Z_{j,K_\ell}^{k+1} - \sum_{F \in \mathcal{E}_{K_\ell}} b_{i,j,K_\ell,E,F} Z_{j,F}^{k+1} \right) + w_{i,K_\ell,E}^{k} \right] + \sum_{j=1}^{n} Z_{i,j,K_\ell,E}^{\text{upw}} \left(a_{i,j,K_\ell,E}^{k} + u_{i,j,K_\ell,E}^{k} \right) \right\} = 0 \tag{4.37}$$

for all $i \in \{1, \dots, n\}$ and all interior faces $E \in \mathcal{E}_h^{\text{int}}$ such that $E \in \mathcal{E}_{K_1} \cap \mathcal{E}_{K_2}$. The aforementioned procedure, where the mobility $m_{i,E}^{k,\text{upw}}$ was eliminated from the balance equation, no longer applies. However, when the mobility m_i does not depend on the unknown variables Z_j and it is strictly positive, the scheme still results in a non-singular system of equations and the implicit upwind scheme may be advantageous for advection problems.

4.2.9 Hybridization

The global system of linear algebraic equations results from combining the following:

- discretization of the original PDE system: Equations (4.26) and (4.27) ($n \times \#\mathcal{K}_h$ equations),
- balance conditions for interior faces ($n \times \#\mathcal{E}_h^{\text{int}}$ equations): either Equation (4.36) (when using the explicit upwind scheme Equation (4.32)) or Equation (4.37) (when using the implicit upwind scheme Equation (4.33)),
- boundary conditions Equations (4.28) to (4.30) ($n \times \#\mathcal{E}_h^{\text{ext}}$ equations).

Overall, we have a system of $n \times (\#\mathcal{K}_h + \#\mathcal{E}_h)$ linear equations for the same number of unknowns $Z_{j,K}^{k+1}$, $Z_{j,E}^{k+1}$, $j \in \{1, \dots, n\}$, $K \in \mathcal{K}_h$, $E \in \mathcal{E}_h$, which can be symbolically written in the block form

$$\begin{pmatrix} \mathbf{Q} & -\mathbf{R} \\ \mathbf{S} & -\mathbf{T} \end{pmatrix} \begin{pmatrix} Z_{\mathcal{K}_h}^{k+1} \\ Z_{\mathcal{E}_h}^{k+1} \end{pmatrix} = \begin{pmatrix} \boldsymbol{G} \\ -\boldsymbol{H} \end{pmatrix}, \tag{4.38}$$

where $Z_{\mathcal{K}_h}^{k+1} = [Z_K^{k+1}]_{K \in \mathcal{K}_h}$, $Z_K^{k+1} = [Z_{j,K}^{k+1}]_{j=1}^n$, $Z_{\mathcal{E}_h}^{k+1} = [Z_F^{k+1}]_{F \in \mathcal{E}_h}$ and $Z_F^{k+1} = [Z_{j,F}^{k+1}]_{j=1}^n$. However, this system is unnecessarily large as we can use its block structure to further reduce the number of equations in the problem. If the blocks $\mathbf{Q}$, $\mathbf{R}$, and $\boldsymbol{G}$ represent Equation (4.26), then $\mathbf{Q}$ is block-diagonal and contains $\#\mathcal{K}_h$ blocks of the size $n \times n$. By computing $Z_{\mathcal{K}_h}^{k+1} = \mathbf{Q}^{-1}(\mathbf{R} Z_{\mathcal{E}_h}^{k+1} + \boldsymbol{G})$, we can eliminate the unknowns $Z_{\mathcal{K}_h}^{k+1}$ from the system. In the context of MHFEM, the technique is called *hybridization* or *static condensation* [B6].

For each $K \in \mathcal{K}_h$, the block of unknowns $Z_K^{k+1} \in \mathbb{R}^n$ can be expressed using $Z_F^{k+1} \in \mathbb{R}^n$, $F \in \mathcal{E}_K$ as

$$Z_K^{k+1} = \sum_{F \in \mathcal{E}_K} \mathbf{Q}_K^{-1} \mathbf{R}_{K,F} Z_F^{k+1} + \mathbf{Q}_K^{-1} \boldsymbol{G}_K, \tag{4.39}$$

where $\mathbf{Q}_K, \mathbf{R}_{K,F} \in \mathbb{R}^{n,n}$, and $\boldsymbol{G}_K \in \mathbb{R}^n$ denote the blocks of the matrix $\mathbf{Q}$, matrix $\mathbf{R}$, and vector $\boldsymbol{G}$, respectively. When the explicit upwind scheme Equation (4.32) is used, the term $Z_{i,j,K,E}^{\mathrm{upw}} = Z_{i,j,K,E}^{k,\mathrm{upw}}$ appears in the right-hand-side block $\boldsymbol{G}_K$ and we can simply following Equation (4.26) to obtain

$$[\mathbf{Q}_K]_{i,j} = \frac{|K|_D}{\Delta t_k} N_{i,j,K}^k + |K|_D r_{i,j,K}^k + \sum_{E \in \mathcal{E}_K} \left(m_{i,E}^{k,\mathrm{upw}} b_{i,j,K,E} - u_{i,j,K,E}^k \right), \tag{4.40a}$$

$$[\mathbf{R}_{K,F}]_{i,j} = \sum_{E \in \mathcal{E}_K} m_{i,E}^{k,\mathrm{upw}} b_{i,j,K,E,F}, \tag{4.40b}$$

$$\begin{aligned} [\boldsymbol{G}_K]_i = |K|_D f_{i,K}^k + \frac{|K|_D}{\Delta t_k} \sum_{j=1}^n N_{i,j,K}^k Z_{j,K}^k - \sum_{E \in \mathcal{E}_K} m_{i,E}^{k,\mathrm{upw}} w_{i,K,E}^k \\ - \sum_{j=1}^n \sum_{E \in \mathcal{E}_K} Z_{i,j,K,E}^{k,\mathrm{upw}} \left(a_{i,j,K,E}^k + u_{i,j,K,E}^k \right), \end{aligned} \tag{4.40c}$$

for all $i, j \in \{1, \dots, n\}$, $K \in \mathcal{K}_h$ and $F \in \mathcal{E}_K$. When the implicit upwind scheme Equation (4.33) is used, the term multiplying $Z_{i,j,K,E}^{\mathrm{upw}} = Z_{i,j,K,E}^{k+1,\mathrm{upw}}$ moves to the blocks $\mathbf{Q}_K$ and $\mathbf{R}_{K,F}$. Combining Equations (4.26) and (4.33) leads to

$$[\mathbf{Q}_K]_{i,j} = \frac{|K|_D}{\Delta t_k} N_{i,j,K}^k + |K|_D r_{i,j,K}^k + \sum_{E \in \mathcal{E}_K} \left(m_{i,E}^{k,\mathrm{upw}} b_{i,j,K,E} - u_{i,j,K,E}^k + \left| a_{i,j,K,E}^k + u_{i,j,K,E}^k \right| \right), \tag{4.41a}$$

$$[\mathbf{R}_{K,F}]_{i,j} = \sum_{E \in \mathcal{E}_K} m_{i,E}^{k,\mathrm{upw}} b_{i,j,K,E,F} - 2 \min \left\{ 0, a_{i,j,K,F}^k + u_{i,j,K,F}^k \right\}, \tag{4.41b}$$

$$[\boldsymbol{G}_K]_i = |K|_D f_{i,K}^k + \frac{|K|_D}{\Delta t_k} \sum_{j=1}^n N_{i,j,K}^k Z_{j,K}^k - \sum_{E \in \mathcal{E}_K} m_{i,E}^{k,\mathrm{upw}} w_{i,K,E}^k, \tag{4.41c}$$

for all $i, j \in \{1, \dots, n\}$, $K \in \mathcal{K}_h$ and $F \in \mathcal{E}_K$.

Analogously, let $\mathbf{T}_{E,F}, \mathbf{S}_{E,K} \in \mathbb{R}^{n,n}$, and $\boldsymbol{H}_E \in \mathbb{R}^n$ denote the blocks of the matrix $\mathbf{T}$, matrix $\mathbf{S}$, and vector $\boldsymbol{H}$, respectively. The values of these blocks also depend on which upwind scheme is used. The

explicit upwind scheme Equation (4.32) uses the balance conditions in Equation (4.36), which leads to

$$[\mathbf{T}_{E,F}]_{i,j} = \sum_{\substack{K\in\mathcal{K}_h \\ E,F\in\mathcal{E}_K}} b_{i,j,K,E,F}, \qquad \forall F \in \mathcal{E}_h, \tag{4.42a}$$

$$[\mathbf{S}_{E,K}]_{i,j} = b_{i,j,K,E}, \qquad \forall K \in \mathcal{K}_h : E \in \mathcal{E}_K, \tag{4.42b}$$

$$[\boldsymbol{H}_E]_i = \sum_{\substack{K\in\mathcal{K}_h \\ E\in\mathcal{E}_K}} w^k_{i,K,E}, \tag{4.42c}$$

for all $i, j \in \{1, \dots, n\}$ and interior faces $E \in \mathcal{E}_h^{\text{int}}$. The implicit upwind scheme Equation (4.33) uses the balance conditions in Equation (4.37), which leads to

$$[\mathbf{T}_{E,F}]_{i,j} = \sum_{\substack{K\in\mathcal{K}_h \\ E,F\in\mathcal{E}_K}} \left(m^{k,\text{upw}}_{i,E} b_{i,j,K,E,F} - 2\delta_{EF} \min\left\{0, a^k_{i,j,K,F} + u^k_{i,j,K,F}\right\}\right), \qquad \forall F \in \mathcal{E}_h, \tag{4.43a}$$

$$[\mathbf{S}_{E,K}]_{i,j} = m^{k,\text{upw}}_{i,E} b_{i,j,K,E} + \left| a^k_{i,j,K,E} + u^k_{i,j,K,E} \right|, \qquad \forall K \in \mathcal{K}_h : E \in \mathcal{E}_K, \tag{4.43b}$$

$$[\boldsymbol{H}_E]_i = m^{k,\text{upw}}_{i,E} \sum_{\substack{K\in\mathcal{K}_h \\ E\in\mathcal{E}_K}} w^k_{i,K,E}, \tag{4.43c}$$

for all $i, j \in \{1, \dots, n\}$ and interior faces $E \in \mathcal{E}_h^{\text{int}}$, where δ_{EF} represents the Kronecker delta.

Values of the blocks $\mathbf{T}_{E,F}$, $\mathbf{S}_{E,K}$, and $\boldsymbol{H}_E$ for boundary faces $E \in \mathcal{E}_h^{\text{ext}}$ are given by the discretized boundary conditions:

- In case of the Dirichlet condition Equation (4.28) for some $j \in \{1, \dots, n\}$ and $E \in \mathcal{E}_h^{\text{ext}}$, the j-th row of $\mathbf{S}_{E,K}$ is zero, $[\mathbf{T}_{E,F}]_{j,j} = 1$, and $[\boldsymbol{H}_E]_j = Z^{\mathcal{D}}_{j,E}(t_{k+1})$.
- In case of the diffusive flux Neumann condition Equation (4.29) for some $i \in \{1, \dots, n\}$, the i-th row of the blocks $\mathbf{T}_{E,F}$ and $\mathbf{S}_{E,K}$ follows the same pattern as in the case of the balance Equation (4.36) for interior faces, except the sums over mesh elements reduce to a single element adjacent to the boundary face. Furthermore, the i-th component of the vector $\boldsymbol{H}_E$ additionally contains the prescribed flux value $v^{\mathcal{N}_1}_{i,E}(t_{k+1})$.
- In case of the total flux Neumann condition Equation (4.30) for some $i \in \{1, \dots, n\}$, the i-th row of the blocks $\mathbf{T}_{E,F}$ and $\mathbf{S}_{E,K}$ follows the same pattern as in the case of the balance Equation (4.37) for interior faces, except the sums over mesh elements reduce to a single element adjacent to the boundary face. Furthermore, the i-th component of the vector $\boldsymbol{H}_E$ additionally contains the prescribed flux value $Q^{\mathcal{N}_2}_{i,E}(t_{k+1})$.

Note that for each boundary face $E \in \mathcal{E}_h^{\text{ext}}$ there may be different types of boundary conditions prescribed for $j \in \{1, \dots, n\}$, so the patterns of the blocks corresponding to such faces may be quite complicated.

When the unknowns $\boldsymbol{Z}^{k+1}_{\mathcal{K}_h}$ are eliminated from Equation (4.38), we obtain the system

$$\mathbf{A}\boldsymbol{Z}^{k+1}_{\mathcal{E}_h} = \boldsymbol{b}, \tag{4.44}$$

where $\mathbf{A} = \mathbf{T} - \mathbf{S}\mathbf{Q}^{-1}\mathbf{R}$ and $\boldsymbol{b} = \boldsymbol{H} + \mathbf{S}\mathbf{Q}^{-1}\boldsymbol{G}$. The entries are given by

$$[\mathbf{A}_{E,F}]_{i,j} = [\mathbf{T}_{E,F}]_{i,j} - \sum_{\substack{K\in\mathcal{K}_h \\ E,F\in\mathcal{E}_K}} [\mathbf{S}_{E,K}\mathbf{Q}_K^{-1}\mathbf{R}_{K,F}]_{i,j}, \tag{4.45a}$$

$$[\boldsymbol{b}_E]_i = [\boldsymbol{H}_E]_i \quad + \sum_{\substack{K\in\mathcal{K}_h \\ E\in\mathcal{E}_K}} [\mathbf{S}_{E,K}\mathbf{Q}_K^{-1}\boldsymbol{G}_K]_{i,j}, \tag{4.45b}$$

for all $i, j \in \{1, \dots, n\}$ and $E, F \in \mathcal{E}_h$. The matrix $\mathbf{A}$ is sparse and non-singular under the assumptions used in the derivation. Even when both tensor coefficients $\mathbf{N}$ and $\mathbf{D}$ in Equation (4.1) are symmetric, the upwind stabilization causes the matrix $\mathbf{A}$ to be non-symmetric. The solution $\boldsymbol{Z}_{\mathcal{E}_h}^{k+1} = \mathbf{A}^{-1}\boldsymbol{b}$ together with the explicit Equation (4.39) allows to obtain all the values needed for the transition to the next time level $t = t_{k+1}$.

4.2.10 Computational algorithm

The system (4.44) together with the explicit Equation (4.39) specify how to compute the approximation of the solution $\boldsymbol{Z}_{\mathcal{K}_h}^{k+1}$ at time t_{k+1} using the approximation $\boldsymbol{Z}_{\mathcal{K}_h}^{k}$ from the previous time level t_k. The procedure to obtain the approximation of the solution at time $t = t_{\max}$ from the initial state at time t_0 involves iterative time-stepping through all time levels $t_k \in \mathcal{T}$. The detailed procedure for computing the numerical solution is summarized in Algorithm 2.

Algorithm 2

1. Read input data characterizing the problem, i.e., mesh $\mathcal{K}_h$, data specifying the initial and boundary conditions, and the coefficients of Equation (4.1) in the discrete form.
2. Set $k = 0$, $t = t_0 = 0$.
3. Initialize $Z_{j,K}^0$ for all $j \in \{1, \dots, n\}$ and $K \in \mathcal{K}_h$ according to the initial condition (4.3).
4. Initialize the coefficients $v_{i,K,E}^0 = 0$ for all $i \in \{1, \dots, n\}$, $K \in \mathcal{K}_h$ and $E \in \mathcal{E}_K$.
5. While $t_k < t_{\max}$:
   5.1. Update the local matrices $\mathbf{b}_{i,j,K}$ given in Appendix E for all $i, j \in \{1, \dots, n\}$ and $K \in \mathcal{K}_h$.
   5.2. Update the coefficients $N_{i,j,K}^k$, $u_{i,j,K,E}^k$, $m_{i,K}^k$, $w_{i,K,E}^k$, $a_{i,j,K,E}^k$, $r_{i,j,K}^k$, $f_{i,K}^k$ for all $i, j \in \{1, \dots, n\}$, $K \in \mathcal{K}_h$, $E \in \mathcal{E}_K$.
   5.3. Use Equation (4.31) to update the coefficients $m_{i,E}^{k,\mathrm{upw}}$ for all $i \in \{1, \dots, n\}$, $E \in \mathcal{E}_h$.
   5.4. If the explicit upwind scheme for advection terms is used, use Equation (4.32) to update the coefficients $Z_{i,j,K,E}^{k,\mathrm{upw}}$ for all $i, j \in \{1, \dots, n\}$, $E \in \mathcal{E}_h$.
   5.5. Update the local matrices $\mathbf{R}_{K,F}$ using Equation (4.40b) (explicit upwind) or Equation (4.41b) (implicit upwind), and vectors $\boldsymbol{G}_K$ using Equation (4.40c) (explicit upwind) or Equation (4.41c) (implicit upwind), for all $K \in \mathcal{K}_h$, $F \in \mathcal{E}_K$.
   5.6. Construct the local matrices $\mathbf{Q}_K$ using Equation (4.40a) (explicit upwind) or Equation (4.41a) (implicit upwind), and compute $\mathbf{Q}_K^{-1}\mathbf{R}_{K,F}$, $\mathbf{Q}_K^{-1}\boldsymbol{G}_K$ for all $K \in \mathcal{K}_h$, $F \in \mathcal{E}_K$.
   5.7. Construct the matrix $\mathbf{A}$ and the right-hand-side vector $\boldsymbol{b}$ in the system (4.44).
   5.8. Solve the system (4.44) for $n \times \#\mathcal{E}_h$ unknowns $Z_{i,E}^{k+1}$, $i \in \{1, \dots, n\}$, $E \in \mathcal{E}_h$.
   5.9. Use Equation (4.39) to compute $Z_{i,K}^{k+1}$ for all $i \in \{1, \dots, n\}$, $K \in \mathcal{K}_h$.
   5.10. Use Equation (4.27) to update the coefficients $v_{i,K,E}^{k+1}$ for all $i \in \{1, \dots, n\}$, $K \in \mathcal{K}_h$, $E \in \mathcal{E}_K$.
   5.11. Set $t_{k+1} = t_k + \Delta t_k$ and $k := k + 1$ to proceed to the next time level.
6. At time $t_k = t_{\max}$ we obtain the numerical solution of the given problem.

4.3 Two-phase flow in porous media

This section summarizes the mathematical model of incompressible two-phase flow in porous media. See [B3, B13] for details on the theory behind the model.

The two-phase incompressible flow in porous media can be described macroscopically by the system of partial differential equations

$$\Phi\rho_w \frac{\partial S_w}{\partial t} - \rho_w \nabla \cdot (\lambda_w K(\nabla p_w - \rho_w \boldsymbol{g})) = 0, \tag{4.46a}$$

$$\Phi\rho_n \frac{\partial S_n}{\partial t} - \rho_n \nabla \cdot (\lambda_n K(\nabla p_n - \rho_n \boldsymbol{g})) = 0, \tag{4.46b}$$

where Φ [-] is the material porosity, K [m^2] is the material permeability, and $\boldsymbol{g}$ [$\mathrm{m\,s}^{-2}$] is the gravitational acceleration vector. Subscripts $\alpha \in \{w, n\}$ are used to denote symbols related to the *wetting* and *non-wetting* phase in the system, respectively. In particular, the quantities ρ_α [$\mathrm{kg\,m}^{-3}$], S_α [-], λ_α [$\mathrm{m\,s\,kg}^{-1}$], and p_α [Pa] represent the α-phase density, saturation, mobility, and pressure, respectively. The densities ρ_α are assumed to be constant, the mobilities are defined as $\lambda_\alpha = k_{r,\alpha}(S_\alpha)/\mu_\alpha$, where $k_{r,\alpha}(S_\alpha)$ [-] is the relative α-phase permeability given by a specific empirical model and μ_α [$\mathrm{kg\,m}^{-1}\,\mathrm{s}^{-1}$] is the α-phase dynamic viscosity, and the quantities S_α, p_α for $\alpha \in \{w, n\}$ are the four unknown variables in Equation (4.46). The model is closed by algebraic constraints for the phase pressures and saturations

$$p_n - p_w = p_c, \tag{4.47a}$$

$$S_w + S_n = 1, \tag{4.47b}$$

where p_c [Pa] is the capillary pressure that is related to the wetting phase saturation S_w according to $p_c = p_c(S_w)$, which is a formula supplied by a specific empirical model. The function $p_c(S_w)$ must be invertible to allow the calculation of saturation from given capillary pressure. Hence, multiple formulations of the problem are possible by selecting different pairs of primary unknown variables, such as $\{p_w, p_n\}$, $\{p_c, p_n\}$, or $\{p_w, S_w\}$.

The empirical model for capillary pressure used in this work is the Brooks–Corey model [A37]

$$p_c(S_w) = (S_{w,e})^{-\frac{1}{\lambda_*}} p_*, \tag{4.48}$$

where p_* [Pa] and λ_* [-] are model parameters for given porous material, and $S_{w,e}$ is the effective saturation of the wetting phase. The effective saturation of the α-phase is defined by

$$S_{\alpha,e} = \frac{S_\alpha - S_{\alpha,r}}{1 - S_{w,r} - S_{n,r}} \tag{4.49}$$

for $\alpha \in \{w, n\}$, where $S_{\alpha,r}$ denotes the residual saturation of the α-phase in the material. The relative permeability model consistent with the Brooks–Corey model for capillary pressure is the Burdine model [A39, A40]

$$k_{r,w}(S_w) = (S_{w,e})^{3+\frac{2}{\lambda_*}}, \tag{4.50a}$$

$$k_{r,n}(S_n) = (S_{n,e})^2 \left(1 - (1 - S_{n,e})^{1+\frac{2}{\lambda_*}}\right). \tag{4.50b}$$

The Equation (4.46) governing the two-phase incompressible flow in porous media can be transformed to the form of Equation (4.1) by taking $n = 2$ and setting the general coefficients appropriately for the selected pair of primary unknown variables. Note that formulations using S_w or S_n as a primary unknown are not admissible for the numerical scheme, because phase saturations may become

discontinuous across material interfaces in heterogeneous porous media. The $\{p_w, p_n\}$ formulation is used in this work, i.e., $Z = [p_w, p_n]^T$, and the coefficients in Equation (4.1) are set as follows:

$$\mathbf{N} = \begin{pmatrix} -\Phi\frac{\mathrm{d}S_w}{\mathrm{d}p_c} & \Phi\frac{\mathrm{d}S_w}{\mathrm{d}p_c} \\ \Phi\frac{\mathrm{d}S_w}{\mathrm{d}p_c} & -\Phi\frac{\mathrm{d}S_w}{\mathrm{d}p_c} \end{pmatrix}, \quad \boldsymbol{u} = \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}, \quad \boldsymbol{m} = \begin{pmatrix} \frac{\lambda_w}{\lambda_t} \\ \frac{\lambda_n}{\lambda_t} \end{pmatrix},$$
$$\mathbf{D} = \begin{pmatrix} \lambda_t K\mathbf{I} & \mathbf{0} \\ \mathbf{0} & \lambda_t K\mathbf{I} \end{pmatrix}, \quad \boldsymbol{w} = \begin{pmatrix} \lambda_t \rho_w K\boldsymbol{g} \\ \lambda_t \rho_n K\boldsymbol{g} \end{pmatrix}, \quad \boldsymbol{a} = \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}, \quad \boldsymbol{r} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \quad \boldsymbol{f} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \tag{4.51}$$

where $\lambda_t = \lambda_w + \lambda_n$ and $\mathbf{I}$ denotes the identity tensor. The coefficients $\boldsymbol{m}$ and $\mathbf{D}$ are chosen based on the approach used in [A99]. Consequently, the modified balance Equation (4.36) can be used in the numerical scheme based on [A72] to allow solving problems with vanishing diffusion.

4.4 Generalized McWhorter–Sunada problem

In this section, we present the results of the generalized McWhorter–Sunada problem, a special case of incompressible two-phase flow in porous media described by Equations (4.46) and (4.47) with known semi-analytical solution [A71, A134].

The geometrical configuration of the problem and boundary conditions are shown in Figure 4.1. The computational domain $\Omega \subset \mathbb{R}^D$ is a unit square in the two-dimensional case and a unit cube in the three-dimensional case. A homogeneous porous medium is assumed in the domain and the effect of gravity is neglected. Initially, at time $t = 0$, the entire domain is filled mostly with water and a small amount of the non-wetting phase. The initial conditions are specified in terms of the reference pressure $p_w = p_{\mathrm{ref}}$ and the constant initial water saturation $S_{w,\mathrm{ini}}$, from which the remaining primary variable p_n is calculated. The non-aqueous phase liquid (NAPL) is being injected into the domain since $t = 0$ until the final time $t_{\max}$ through a point source located in the origin $\boldsymbol{x} = \mathbf{0}$. The volumetric rate of injection, denoted as Q_0 [$\mathrm{m}^D\ \mathrm{s}^{-1}$], is prescribed by

$$Q_0 = Q_0(t) = A_D t^{\frac{D-2}{2}}, \tag{4.52}$$

where $D \in \{2, 3\}$ denotes the spatial dimension and A_D [$\mathrm{m}^D\ \mathrm{s}^{-D/2}$] is an injection rate parameter.

As noted in [A72, B20], the boundary condition involving the point source injection cannot be treated exactly in the numerical scheme and it must be approximated using boundary faces of the numerical mesh adjacent to the origin $\boldsymbol{x} = \mathbf{0}$. The part of the domain boundary $\Gamma = \partial\Omega$ that is used for the approximation of the injection is illustrated in Figure 4.2 and denoted as Γ_{in}. The injection rate Q_0 is enforced on Γ_{in} for all $t \in [0, t_{\max}]$ as

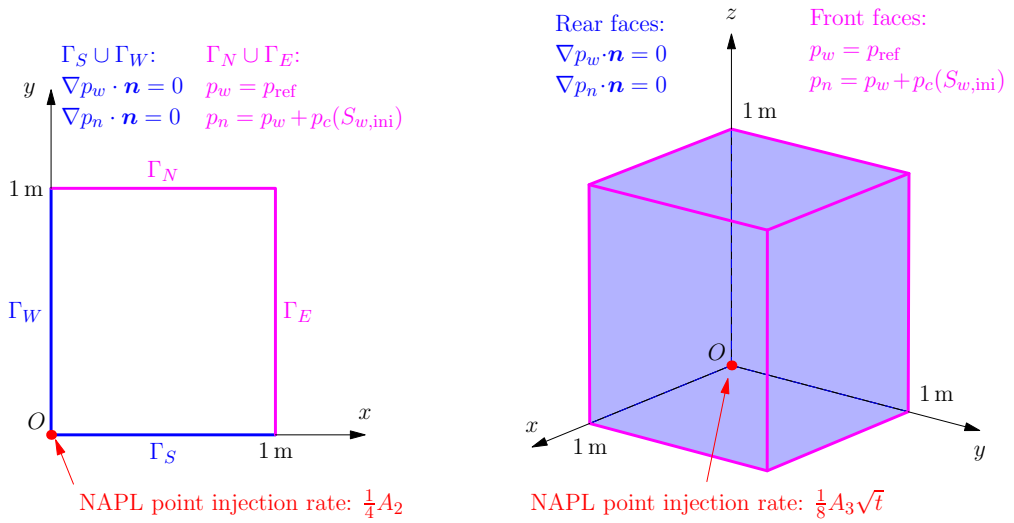
$$\int_{\Gamma_{\mathrm{in}}} \lambda_n K \nabla p_n \cdot \boldsymbol{n} = Q_0(t), \tag{4.53}$$

where $\boldsymbol{n}$ is the outward unit normal vector on Γ_{in}. The boundary condition for the wetting phase on Γ_{in} remains the same, i.e., $\nabla p_w \cdot \boldsymbol{n} = 0$.

All parameters of the benchmark problem were chosen identically to [B20]. Except for p_* and λ_*, they also correspond to the values used in [A72]. The values of p_* and λ_* used here correspond to the drainage rather than wetting regime of the material, see [B13]. The values of all parameters used in this work are summarized in Table 4.1.

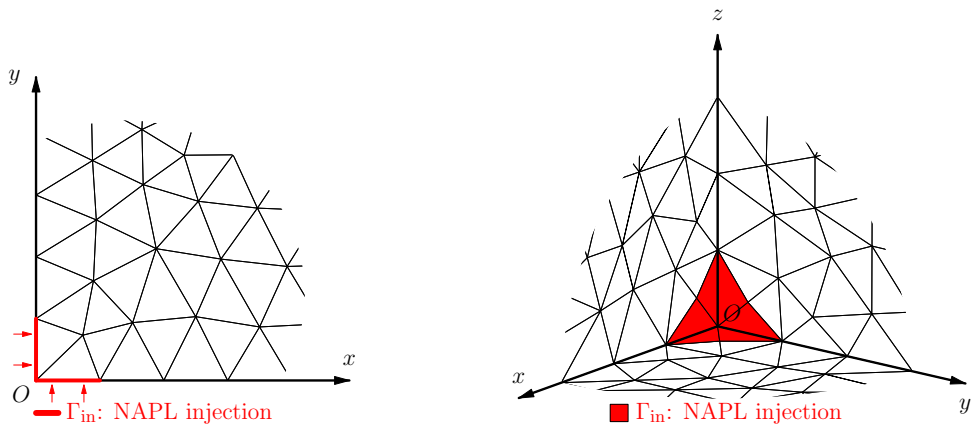
4.4.1 Verification results

For each mesh $\mathcal{K}_h$, we define the error $E_{h,S_n} = S_{n,\mathrm{ex}}(t_{\max}) - S_{n,h}(t_{\max})$ in terms of the injected phase saturation S_n as the difference between the semi-analytical solution $S_{n,\mathrm{ex}}$ and numerical solution $S_{n,h}$



(a) 2D (b) 3D

Figure 4.1: Setup of the computational domain for the generalized McWhorter–Sunada problem with boundary conditions in (a) 2D and (b) 3D. Adapted from [A72].



(a) Triangles (b) Tetrahedra

Figure 4.2: Approximation of the point injection boundary condition at $\boldsymbol{x} = \boldsymbol{0}$ on (a) triangular and (b) tetrahedral meshes. Adapted from [A72].

on the mesh $\mathcal{K}_h$ at time $t_{\max}$. The L_p norm $\|E_{h,S_n}\|_p$ is evaluated in $\Omega \subset \mathbb{R}^D$ for $p \in \{1, 2\}$ as

$$\|E_{h,S_n}\|_p = \left(\int_\Omega |E_{h,S_n}(\boldsymbol{x})|^p \mathrm{d}\boldsymbol{x} \right)^{\frac{1}{p}} = \left(\sum_{K \in \mathcal{K}_h} \int_K |E_{h,S_n}(\boldsymbol{x})|^p \mathrm{d}\boldsymbol{x} \right)^{\frac{1}{p}}. \tag{4.54}$$

The integrals over mesh cells $K \in \mathcal{K}_h$ are first transformed to integrals over the reference cell, and then the Fubini theorem is applied to obtain one-dimensional integrals that are computed numerically using the Gauss–Kronrod quadrature (G7–K15) with adaptive interval subdivision, which is implemented in the C++11 library `qdt` [A140, O23].

Given two approximate solutions S_{n,h_1} and S_{n,h_2} on meshes $\mathcal{K}_{h_1}$ and $\mathcal{K}_{h_2}$, respectively, the order of convergence of the numerical scheme with respect to S_n in the L_p norm can be approximated using an experimental order of convergence as

$$EOC_{S_n,p} = \frac{\ln \|E_{h_1,S_n}\|_p - \ln \|E_{h_2,S_n}\|_p}{\ln h_1 - \ln h_2}. \tag{4.55}$$

Table 4.1: Parameters of the mathematical model used in the generalized McWhorter–Sunada problem.

Parameter	Symbol	Value	Unit
Material porosity	Φ	0.343	
Material permeability	K	5.168×10^{-12}	m^2
Brooks–Corey model parameter	p_*	8027.52	Pa
Brooks–Corey model parameter	λ_*	5.408	
Residual saturation – wetting phase	$S_{w,r}$	0.04	
– non-wetting phase	$S_{n,r}$	0	
Fluid density – wetting phase	ρ_w	10^3	$\mathrm{kg\,m^{-3}}$
– non-wetting phase	ρ_n	1.4×10^3	$\mathrm{kg\,m^{-3}}$
Dynamic viscosity – wetting phase	μ_w	10^{-3}	$\mathrm{kg\,m^{-1}\,s^{-1}}$
– non-wetting phase	μ_n	10^{-3}	$\mathrm{kg\,m^{-1}\,s^{-1}}$
Gravitational acceleration	$\boldsymbol{g}$	$\mathbf{0}$	$\mathrm{m\,s^{-2}}$
Reference pressure	p_{ref}	10^5	Pa
Initial wetting phase saturation	$S_{w,\mathrm{ini}}$	0.95	
Injection rate parameter – 2D case	A_2	10^{-5}	$\mathrm{m^2\,s^{-1}}$
– 3D case	A_3	10^{-7}	$\mathrm{m^3\,s^{-3/2}}$
Final time	$t_{\max}$	2×10^4	s

Table 4.2: Results of the numerical analysis using the L_1 and L_2 norms of E_{h,S_n} for the 2D problem on a series of triangular discretizations.

Id.	h [m]	τ [s]	$\|E_{h,S_n}\|_1$	$EOC_{S_n,1}$	$\|E_{h,S_n}\|_2$	$EOC_{S_n,2}$
$2D_1^\triangle$	6.71×10^{-2}	454.55	1.54×10^{-2}		3.25×10^{-2}	
				0.97		0.84
$2D_2^\triangle$	3.49×10^{-2}	145.99	8.14×10^{-3}		1.89×10^{-2}	
				0.80		0.61
$2D_3^\triangle$	1.64×10^{-2}	44.64	4.44×10^{-3}		1.19×10^{-2}	
				0.97		0.67
$2D_4^\triangle$	8.73×10^{-3}	13.44	2.41×10^{-3}		7.79×10^{-3}	
				0.86		0.64
$2D_5^\triangle$	4.23×10^{-3}	5.00	1.29×10^{-3}		4.90×10^{-3}	

Table 4.3: Results of the numerical analysis using the L_1 and L_2 norms of E_{h,S_n} for the 3D problem on a series of tetrahedral discretizations.

Id.	h [m]	τ [s]	$\|E_{h,S_n}\|_1$	$EOC_{S_n,1}$	$\|E_{h,S_n}\|_2$	$EOC_{S_n,2}$
$3D_1^\triangle$	2.13×10^{-1}	833.33	1.15×10^{-2}		3.48×10^{-2}	
				0.69		0.62
$3D_2^\triangle$	1.27×10^{-1}	571.43	8.02×10^{-3}		2.52×10^{-2}	
				0.86		0.75
$3D_3^\triangle$	6.29×10^{-2}	232.56	4.41×10^{-3}		1.49×10^{-2}	
				1.03		0.93
$3D_4^\triangle$	3.48×10^{-2}	101.01	2.40×10^{-3}		8.62×10^{-3}	
				1.01		0.71
$3D_5^\triangle$	1.84×10^{-2}	25.00	1.26×10^{-3}		5.48×10^{-3}	

In order to investigate the convergence and accuracy of the numerical scheme using the experimental order of convergence, several simulations were computed on a series of triangular and tetrahedral meshes. The meshes are the same as those used in [A72] and their properties are listed in Table 2.2. All simulations were computed in `double` precision. Tables 4.2 and 4.3 show the resulting L_1 and L_2 norms of the error, the corresponding experimental orders of convergence, as well as the mesh size parameters h and constant time steps τ that were used in the simulations. The results of the numerical analysis indicate that the scheme converges with the first order of accuracy in both dimensions and both norms.

4.5 Benchmarking methodology

The generalized McWhorter–Sunada problem was computed as a benchmark for several configurations of the solver. All configurations were tested on the triangular as well as tetrahedral meshes with the same problem parameters as in the verification study described in the previous subsection. The computational parameters that varied in the benchmark are:

- *Linear system solver.* First, we used the BiCGstab method implemented in TNL with the Jacobi preconditioner. Additionally, we used the BoomerAMG preconditioner [A184] from the Hypre library [C15, C19, C20] version 2.25.0 together with the BiCGstab method implemented in the Hypre library. Some other components of the solver also had to be changed due to compatibility with these two different BiCGstab implementations. While the TNL implementation is not bound to any specific matrix format and the Ellpack format was used in the solver, the Hypre library requires the CSR format with specific conventions (see Section 3.4 for details). Hence, the sparse matrix assembly uses slightly different procedures in the two configurations.
- *Hardware architecture.* The benchmark was computed either solely on CPUs, or using GPU accelerators. In the latter case, all parts of the MHFEM algorithm were executed directly on GPUs, so data transfers between the GPU and main system memory do not limit the performance. Both TNL and Hypre rely on the CUDA framework [M19] for GPU parallelization and they use the so-called CUDA-aware MPI [O18] to avoid buffering of the data passed between multiple GPUs in the system memory.
- *Programming framework for CPU parallelization.* For the TNL implementation, we tested two approaches to CPU parallelization within a single node. We used the shared-memory multi-thread approach via the OpenMP framework [A51] and also the general multi-process approach based on MPI [M15]. The Hypre implementation was used only with the MPI-based parallelization.

To ensure comparable results obtained with different preconditioners for the BiCGstab method, care must be taken to select suitable stopping criteria for the iterative algorithm. The linear system solvers implemented in TNL use left-preconditioning, whereas the Hypre library implements solvers with right-preconditioning (see Chapter 3 for details). Recall that right-preconditioned methods operate with the *original* residual and right-hand-side vectors, whereas left-preconditioned methods operate with *preconditioned* residual and right-hand-side vectors. Hence, assuming the Jacobi preconditioner, the right-hand-side vector is scaled with the matrix diagonal entries during left-preconditioning and the stopping criterion based on the norm of the residual vector divided by the norm of the right-hand-side vector cannot be used with the same threshold as for the unpreconditioned or right-preconditioned system. To remedy this problem, we assemble the linear system with scaled rows to obtain ones on the main diagonal of the matrix (i.e., the Jacobi preconditioning is applied during the matrix assembly). This scales the elements of the right-hand-side vector to the same magnitude in both preconditioning techniques and allows the stopping criterion to be used with the same threshold (10^{-11} in this study).

Besides the stopping criterion, the BiCGstab method requires no other configuration. The following common parameters were set for the BoomerAMG preconditioner:

- The PDE system size (i.e., n in Equation (4.1)) was set to 2:
  `HYPRE_BoomerAMGSetNumFunctions( precond, 2 );`
- Aggressive coarsening with 1 level: `HYPRE_BoomerAMGSetAggNumLevels( precond, 1 );`
- For both 2D and 3D problems, the strength threshold was set to 0.25:
  `HYPRE_BoomerAMGSetStrongThreshold( precond, 0.25 );`
- The `extended+i` interpolation operator with truncation to at most 4 elements per row
  `HYPRE_BoomerAMGSetInterpType( precond, 6 );` and `HYPRE_BoomerAMGSetPMaxElmts( precond, 4 );`

Additionally, the following parameters were set for CPU computations:

- The HMIS coarsening algorithm: `HYPRE_BoomerAMGSetCoarsenType( precond, 10 );`
- The ℓ_1-scaled hybrid symmetric Gauss–Seidel smoother:
  `HYPRE_BoomerAMGSetRelaxType( precond, 8 );`
- The truncation factor for the interpolation was set to 0.3:
  `HYPRE_BoomerAMGSetTruncFactor( precond, 0.3 );`

The following parameters were set specifically for GPU computations:

- The PMIS coarsening algorithm: `HYPRE_BoomerAMGSetCoarsenType( precond, 8 );`
- The ℓ_1-scaled Jacobi smoother: `HYPRE_BoomerAMGSetRelaxType( precond, 18 );`
- The 2-stage `extended+e` interpolation in matrix-matrix form was used on the levels of aggressive coarsening: `HYPRE_BoomerAMGSetAggInterpType( precond, 7 );`
- The local interpolation transposes were saved to avoid sparse matrix–transpose–vector multiplications: `HYPRE_BoomerAMGSetKeepTranspose( precond, 1 );`
- The triple matrix product **RAP** was replaced by two matrix products:
  `HYPRE_BoomerAMGSetRAP2( precond, 1 );`

There are many other configurable parameters listed in the Hypre documentation [O10] that might have an effect on the performance, but they were not investigated in this benchmark.

To reduce the computational time with the Hypre implementation, the BoomerAMG preconditioner is not updated in every time step and instead its setup is reused multiple times as long as the number of iterations necessary for the BiCGstab method to converge does not increase significantly. The specific heuristics used is that the preconditioner is reused, as long as the number of iterations needed in the previous time step is less then 5 plus the number of iterations needed when the preconditioner was last updated.

The computations were performed on the same hardware as described in Section 2.2.7, and also the software versions and compiler parameters were the same. The MPI version is OpenMPI 4.0.5 with CUDA support and system-specific drivers. Each computation was performed exactly once and the times spent in individual parts of the algorithm were measured. The computational time CT was measured since the solver was fully initialized until the final simulation time was reached.

The secondary quantities of interest for the evaluation of parallel computations are speed-up Sp, which is defined as the ratio between computational times using 1 and ℓ compute units (e.g., CPU cores), and parallel efficiency Eff, which quantifies the scalability of the computation and is defined as the ratio between the speed-up and the number of compute units, i.e.

$$Eff = \frac{CT \text{ for 1 unit}}{\ell \times (CT \text{ for } \ell \text{ units})}.$$

In case of GPU computations, the speed-up and efficiency are calculated relative to the number of whole GPUs rather than the number of individual GPU cores, because the GPU cores are not independent.

4.6 Computational benchmark results

The benchmark results are presented as strong scaling studies for the finest triangular mesh $2D_5^{\triangle}$ and finest tetrahedral mesh $3D_5^{\triangle}$. Table 4.4 shows the CPU computational times CT and parallel efficiency Eff for the 2D problem. The computational times of the TNL solver using OpenMP and MPI are comparable, but the latter is faster in most cases. The OpenMP efficiency even drops below 0.5 when

Table 4.4: Comparison of CPU computational times *CT* [s], speed-up *Sp*, and parallel efficiency *Eff* for the generalized McWhorter–Sunada problem computed on the finest triangular mesh $2D_5^{\triangle}$.

	TNL						Hypre		
	OpenMP			MPI			MPI		
Cores	*CT*	*Sp*	*Eff*	*CT*	*Sp*	*Eff*	*CT*	*Sp*	*Eff*
1	10743.9	1.0	1.00	10800.2	1.0	1.00	3510.1	1.0	1.00
2	6349.0	1.7	0.85	5693.5	1.9	0.95	2058.0	1.7	0.85
4	3375.9	3.2	0.80	3143.0	3.4	0.86	1097.1	3.2	0.80
6	2294.6	4.7	0.78	2506.0	4.3	0.72	750.5	4.7	0.78
8	1818.1	5.9	0.74	1787.6	6.0	0.76	587.6	6.0	0.75
12	1296.2	8.3	0.69	1096.8	9.8	0.82	424.8	8.3	0.69
24	977.0	11.0	0.46	549.3	19.7	0.82	215.5	16.3	0.68

all 24 cores of the two processors on a cluster node are used, which is most likely caused by inefficient memory allocations in the NUMA system and excessive communication over the interconnection between the processors. The Hypre computations are faster than TNL by a factor of 2.5 to 3, which manifests the importance of a strong preconditioner. The linear systems arising from the 2D problem took the Jacobi-preconditioned BiCGstab method more than 500 iterations on average to converge, whereas the BoomerAMG-preconditioned BiCGstab converged after just 23 iterations on average.

The results of CPU computations for the 3D problem are shown in Table 4.5. For a single node (up to 24 cores), the results are analogous to the 2D problem, except the differences between the methods are much larger. For TNL computations, the OpenMP and MPI results are comparable only for 1 core (i.e., sequential computation) and MPI is significantly faster than OpenMP in all other cases. This is surprising, because OpenMP operates on the shared memory level whereas the MPI approach is based on messages explicitly passed between the processes, but it demonstrates that a simple and user-friendly programming interface may incur significant performance penalty that is hard to analyze and eliminate. The Hypre computations exhibit parallel efficiency similar to the MPI-based TNL computations, but the computational times are almost 5 times shorter. The linear systems arising from the 3D problem took the Jacobi-preconditioned BiCGstab about 530 iterations on average to converge, whereas the BoomerAMG-preconditioned BiCGstab converged after just 17 iterations on average.

Additionally, Table 4.5 includes results of CPU computations distributed across multiple nodes using MPI. The computational cluster used for the computations consists of 20 dual-processor nodes containing CPUs listed in Table 2.1, but only 14 nodes at most could be employed together in one MPI computation. It can be noticed that while the parallel efficiency with respect to 1 core dropped down to about 0.65 on 24 cores, which is caused by the saturation of the memory bandwidth and automatic CPU core frequency scaling, it stays on this level or even increases slightly when multiple nodes are utilized for the computation. This increase in efficiency may be attributed to the increasing total cache size and the problem size staying constant in the strong scaling study. Using Hypre for the solution of the linear systems lead to lower parallel efficiency compared to the simple Jacobi preconditioner used in TNL computations, which is an inevitable consequence of more complex communication patterns during preconditioning. Even for the computations using the most nodes, Hypre is more than 4 times faster than TNL. The Hypre library provides state–of–the–art linear system solvers and preconditioners targetting contemporary supercomputers consisting of thousands of nodes, but much larger problems would be necessary to efficiently utilize these computational resources.

The results of GPU computations for both 2D and 3D problems are shown in Table 4.6. Note that each MPI rank manages its dedicated GPU, so each computation used as many GPUs as there

Table 4.5: Comparison of CPU computational times *CT* [s], speed-up *Sp*, and parallel efficiency *Eff* for the generalized McWhorter–Sunada problem computed on the finest tetrahedral mesh $3D_5^{\triangle}$.

			TNL						Hypre		
			OpenMP			MPI			MPI		
Cores	CPUs	Nodes	*CT*	*Sp*	*Eff*	*CT*	*Sp*	*Eff*	*CT*	*Sp*	*Eff*
1			188243.0	1.0	1.00	188706.0	1.0	1.00	37991.2	1.0	1.00
2			102074.0	1.8	0.92	93659.1	2.0	1.01	21170.2	1.8	0.90
4			55937.6	3.4	0.84	49553.0	3.8	0.95	11252.2	3.4	0.84
6			40796.4	4.6	0.77	35594.3	5.3	0.88	7798.1	4.9	0.81
8			32026.3	5.9	0.73	28958.6	6.5	0.81	6085.4	6.2	0.78
12	1	1/2	26369.7	7.1	0.59	23839.0	7.9	0.66	4708.8	8.1	0.67
24	2	1	15695.0	12.0	0.50	12184.2	15.5	0.65	2485.0	15.3	0.64
48	4	2				6029.0	31.3	0.65	1249.1	30.4	0.63
72	6	3				4054.7	46.5	0.65	880.2	43.2	0.60
96	8	4				2974.5	63.4	0.66	592.3	64.1	0.67
120	10	5				2483.0	76.0	0.63	471.2	80.6	0.67
144	12	6				2000.0	94.4	0.66	415.8	91.4	0.63
168	14	7				1607.7	117.4	0.70	372.2	102.1	0.61
192	16	8				1380.4	136.7	0.71	310.7	122.3	0.64
216	18	9				1209.6	156.0	0.72	277.5	136.9	0.63
240	20	10				1082.0	174.4	0.73	240.3	158.1	0.66
264	22	11				974.9	193.6	0.73	251.5	151.0	0.57
288	24	12				892.5	211.4	0.73	223.9	169.7	0.59
312	26	13				901.8	209.3	0.67	202.9	187.2	0.60
336	28	14				851.9	221.5	0.66	201.9	188.2	0.56

were MPI ranks. It can be noticed that in the 2D case, using multiple GPUs does not lead to faster computations (the speed-ups *Sp* for the mesh $2D_5^{\triangle}$ are less than 1). This can be attributed to the latency of communication between multiple GPUs, which is high compared to the actual computation, because the triangular meshes used in the benchmark do not provide enough degrees of freedom to fully utilize the computational power of the GPUs. On the other hand, the tetrahedral meshes used in the benchmark provide more degrees of freedom, so the speed-ups *Sp* for the mesh $3D_5^{\triangle}$ rise above 1. Comparing the GPU computational times from Table 4.6 with the corresponding CPU computations from Tables 4.4 and 4.5, it can be seen that using GPU acceleration brings significant advantage for the simple Jacobi preconditioner in TNL, but not so significant for the BoomerAMG preconditioner from the Hypre library. For the TNL solver and the 3D problem, using 1 GPU leads to a faster computational time than when using 8 CPUs (4 nodes) and at least 32 CPUs (16 nodes) are necessary for a faster computational time than when using 4 GPUs. The GPU computations with Hypre are only slightly faster than TNL computations, only 2 CPUs (1 node) are necessary for Hypre to be competitive with 1 GPU and using Hypre on 8 CPUs (4 nodes) is faster than the fastest computation using 4 GPUs.

The final result shown in this section is the breakdown of the overall computational times from Table 4.5 on the finest tetrahedral mesh $3D_5^{\triangle}$. In Tables 4.7 and 4.8 it can be observed that regardless of the preconditioner used, the major portion (over 90 %) corresponds to the linear system solver (BiCGstab) rather than operations involving the unstructured mesh. Recall that the BoomerAMG preconditioner is not updated in every time step, so its contribution to the total computational time stays below 1 %. The other entries in the tables correspond to various operations for the MHFEM discretization

Table 4.6: Comparison of GPU computational times *CT* [s], speed-up *Sp*, and parallel efficiency *Eff* for the generalized McWhorter–Sunada problem computed on the finest triangular and tetrahedral meshes. Each MPI rank manages its dedicated GPU.

	TNL						Hypre					
	$2D_5^{\triangle}$			$3D_5^{\triangle}$			$2D_5^{\triangle}$			$3D_5^{\triangle}$		
GPUs	*CT*	*Sp*	*Eff*	*CT*	*Sp*	*Eff*	*CT*	*Sp*	*Eff*	*CT*	*Sp*	*Eff*
1	528.6	1.0	1.00	2654.8	1.0	1.00	389.8	1.0	1.00	2014.5	1.0	1.00
2	566.1	0.9	0.47	1415.4	1.9	0.94	500.6	0.8	0.39	1233.1	1.6	0.82
3	642.5	0.8	0.27	996.7	2.7	0.89	634.1	0.6	0.20	868.9	2.3	0.77
4	709.7	0.7	0.19	793.3	3.3	0.84	726.8	0.5	0.13	704.2	2.9	0.72

Table 4.7: Comparison of the portions contributing to the total CPU computational times *CT* [s] for the generalized McWhorter–Sunada problem computed on the finest tetrahedral mesh $3D_5^{\triangle}$ using BiCGstab from TNL as the linear system solver. All values are average times of all MPI ranks.

Number of MPI ranks	12	24	48	96	192	288	336
MHFEM routines	0.9 %	0.9 %	1.0 %	1.0 %	1.1 %	1.1 %	1.0 %
MPI communication of mesh data	0.0 %	0.0 %	0.1 %	0.1 %	0.1 %	0.2 %	0.2 %
Sparse matrix assembly	0.7 %	0.6 %	0.6 %	0.6 %	0.6 %	0.6 %	0.5 %
Linear system solver (compute)	94.1 %	90.0 %	89.6 %	85.6 %	80.1 %	74.3 %	65.0 %
Linear system solver (MPI sync.)	4.3 %	8.4 %	8.7 %	12.7 %	18.1 %	23.8 %	33.3 %
Total	23839 s	12184 s	6029 s	2974 s	1380 s	893 s	852 s

Table 4.8: Comparison of the portions contributing to the total CPU computational times *CT* [s] for the generalized McWhorter–Sunada problem computed on the finest tetrahedral mesh $3D_5^{\triangle}$ using BoomerAMG-preconditioned BiCGstab from the Hypre library as the linear system solver. All values are average times of all MPI ranks.

Number of MPI ranks	12	24	48	96	192	288	336
MHFEM routines	4.8 %	4.6 %	4.6 %	4.8 %	4.6 %	4.3 %	4.1 %
MPI communication of mesh data	0.1 %	0.2 %	0.3 %	0.3 %	0.4 %	1.0 %	1.1 %
Sparse matrix assembly	3.5 %	3.2 %	3.1 %	3.2 %	2.9 %	2.6 %	2.4 %
BoomerAMG update	0.4 %	0.4 %	0.4 %	0.5 %	0.5 %	0.8 %	0.8 %
Linear system solver	91.1 %	91.5 %	91.6 %	91.2 %	91.5 %	91.3 %	91.6 %
Total	4709 s	2485 s	1249 s	592 s	311 s	224 s	202 s

(*MHFEM routines*), data synchronizations in the ghost regions with `DistributedMeshSynchronizer` (*MPI communication of mesh data*), and the sparse matrix assembly. The MHFEM routines involve the same operations in both TNL and Hypre implementations and thus take the same time, but they are relatively more significant in the Hypre implementation where the linear system solver is faster. The sparse matrix assembly for Hypre takes slightly longer time than for TNL, but it is still faster than the MHFEM routines. Note that the linear system solver also includes a considerable amount of MPI communication which is segregated in its own contribution in Table 4.7, but included in the linear system solver entry in Table 4.8.

Chapter 5

Lattice Boltzmann Method

This chapter is dedicated to the lattice Boltzmann method (LBM), a popular numerical approach for the simulation of fluid flow [B21], chemical or thermal transport [A52, A155], radiative transport [A135, A136], and other problems. The method can be easily and efficiently parallelized [B21] and the advent of general-purpose computing on graphics processing units (GPGPU) made large-scale and highly resolved numerical simulations of turbulent flows feasible [A84, A106, A118, A147, A181, A188].

The content of the chapter focuses on the implementation of the method and performance optimizations for distributed parallel computing platforms, which is the author's contribution within the research group. Hence, LBM is described with the objective to formulate the computational algorithm and details such as rigorous derivation of the method are omitted. The LBM implementation used in this work was developed, analyzed and validated in [A23, A63, A64, A68–A70, A75].

The chapter is organized as follows. First, the background of the method in the kinetic theory of gases is described in Section 5.1 and its components are introduced in Section 5.2. The following Section 5.3 describes the streaming schemes and the computational algorithm is formulated in Section 5.4 in several variants leading to the scalable distributed algorithm. Section 5.5 provides insights into the implementation and Section 5.6 describes important performance optimizations. The final Section 5.7 contains the results of several computational benchmarks, including strong and weak scaling studies on the Karolina supercomputer [O15].

5.1 Background

The lattice Boltzmann method [B21] (LBM) is an alternative to traditional computational methods such as finite difference, finite volume, and finite element methods. Instead of directly solving a macroscopic PDE, such as the Navier–Stokes equations or a general advection-diffusion equation, LBM approximates the temporal evolution of macroscopic quantities such as density ρ [$\mathrm{kg\,m^{-3}}$], velocity $\boldsymbol{v}$ [$\mathrm{m\,s^{-1}}$], and other variables (e.g., pressure, stress tensor, etc.) using probability moments of the density distribution function $f(\boldsymbol{x}, \boldsymbol{\xi}, t)$ [$\mathrm{kg\,s^3\,m^{-6}}$] that represents the density of particles with velocity $\boldsymbol{\xi} = [\xi_x, \xi_y, \xi_z]^T$ [$\mathrm{m\,s^{-1}}$] at position $\boldsymbol{x}$ and time t. Based on the kinetic theory [B21], the density distribution function f evolves according to the Boltzmann transport equation

$$\frac{\partial f}{\partial t} + \sum_{i=1}^{3} \xi_i \frac{\partial f}{\partial x_i} + \sum_{i=1}^{3} \frac{F_i}{\rho} \frac{\partial f}{\partial \xi_i} = C, \tag{5.1}$$

where ρ [$\mathrm{kg\,m^{-3}}$] is the mass density, $\boldsymbol{F} = [F_x, F_y, F_z]^T$ [$\mathrm{N\,m^{-3}}$] represents the external body force density acting on the mass and C [$\mathrm{kg\,s^2\,m^{-6}}$] denotes a general collision operator that depends on the distribution function and other parameters (e.g., relaxation times and equilibrium distribution). LBM follows from the discretization of Equation (5.1) in space, velocity space, and time.

5.2 Components of the method

The detailed derivation of LBM is not within the scope of this thesis. Instead, we review the components needed to describe the implementation of the method and refer the reader to [B21] for further details.

5.2.1 Lattice and velocity set

We consider the discretization of a cuboidal domain $\Omega \subset \mathbb{R}^3$ using an equidistant orthogonal grid represented by a finite set $\mathcal{G}_\Omega$ of grid nodes. The extended grid $\mathcal{G}_{\overline{\Omega}}$ consists of the interior nodes from $\mathcal{G}_\Omega$ and an additional outer layer of nodes, which is added to facilitate the discretization of the domain boundary. The Cartesian coordinates of grid nodes are defined as $\boldsymbol{x}_{i,j,k} = [i\delta_x + O_x, j\delta_x + O_y, k\delta_x + O_z]^T$, where δ_x [m] is the grid spacing parameter and $\boldsymbol{O} = [O_x, O_y, O_z]^T$ [m] is an offset vector that allows to fit the grid nodes to the physical coordinates of the domain. The sets $\mathcal{G}_\Omega$ and $\mathcal{G}_{\overline{\Omega}}$ of interior and all grid nodes, respectively, are defined by specifying the indices i, j, k for the points $\boldsymbol{x}_{i,j,k}$ as

$$\mathcal{G}_\Omega = \{\boldsymbol{x}_{i,j,k} \in \Omega \mid i \in \{1, \dots, N_x - 2\},\ j \in \{1, \dots, N_y - 2\},\ k \in \{1, \dots, N_z - 2\}\}, \tag{5.2a}$$

$$\mathcal{G}_{\overline{\Omega}} = \{\boldsymbol{x}_{i,j,k} \mid i \in \{0, \dots, N_x - 1\},\ j \in \{0, \dots, N_y - 1\},\ k \in \{0, \dots, N_z - 1\}\}, \tag{5.2b}$$

where the integers N_x, N_y, N_z determine the numbers of grid nodes along the x, y, and z axes, respectively. To avoid mixing different unit systems, we also define a non-dimensional lattice $\mathcal{L}_\Omega$ and extended lattice $\mathcal{L}_{\overline{\Omega}}$ as

$$\mathcal{L}_\Omega = \{\Delta x[i, j, k]^T \mid i \in \{1, \dots, N_x - 2\},\ j \in \{1, \dots, N_y - 2\},\ k \in \{1, \dots, N_z - 2\}\}, \tag{5.3a}$$

$$\mathcal{L}_{\overline{\Omega}} = \{\Delta x[i, j, k]^T \mid i \in \{0, \dots, N_x - 1\},\ j \in \{0, \dots, N_y - 1\},\ k \in \{0, \dots, N_z - 1\}\}, \tag{5.3b}$$

where Δx is the lattice spacing parameter that we set to $\Delta x = 1$ as usual in the LBM community [B21]. Note that a grid node $\boldsymbol{x}_{i,j,k}$ represents the same point in space as the corresponding lattice site $\Delta x[i, j, k]^T$, but in different unit systems. Hence, we will use the notation $\boldsymbol{x} \in \mathcal{G}_{\overline{\Omega}}$ to refer to a point using its physical coordinates and $\boldsymbol{x} \in \mathcal{L}_{\overline{\Omega}}$ to refer to its lattice coordinates.

Similarly to the space discretization, the time interval $[0, t_{\max}]$ is discretized by a finite set $\mathcal{G}_t = \{\ell\delta_t \mid \ell \in \{0, \dots, N_t\}\}$, where $\delta_t = t_{\max}/N_t$ [s] is the time step in physical units and N_t is the number of time steps. The set of non-dimensional time levels is denoted as $\mathcal{L}_t = \{\ell\Delta t \mid \ell \in \{0, \dots, N_t\}\}$, where $\Delta t = 1$ is the non-dimensional time step.

The next step in the discretization of the Boltzmann transport equation (5.1) is the selection of an appropriate discrete velocity set. The velocity sets are typically named as DDQQ, where D stands for the spatial dimension and Q denotes the number of discrete velocities per lattice site. The most common examples are D1Q3, D2Q5, D2Q9, D3Q7, D3Q15, D3Q19, and D3Q27. Not all velocity sets are suitable for accurate resolution of the desired macroscopic equation [A68, A75, B21]. In this thesis, we aim to use LBM for the solution of the Navier–Stokes equations in $\mathbb{R}^3$ and thus consider the D3Q27 velocity set consisting of $Q = 27$ discrete velocities.

A velocity set for LBM is fully defined by two sets of quantities: lattice velocities $\boldsymbol{\xi}_q$ and the corresponding weights w_q for $q \in \{1, \dots, Q\}$. All vectors $\boldsymbol{\xi}_q$ in all aforementioned velocity sets have components in $\{-c, 0, c\}$, where $c = \Delta x/\Delta t$ is the lattice speed. Another important quantity that can be derived from these two sets is the lattice speed of sound c_s. The aforementioned velocity sets can be derived using the Gauss–Hermite quadrature [B21, Appendix A.4]. Another approach explained in [B21, Section 3.4.7.2] is to determine the weights by solving the system of equations representing general conditions that ensure rotational isotropy of the lattice. The Python script in Appendix F can be used to quickly verify the consistency of discrete velocities, weights and the speed of sound for a given velocity set, or even to obtain the necessary conditions when designing a new velocity set.

5.2.2 Discrete lattice Boltzmann equation

Before discretization, Equation (5.1) is first transformed to a non-dimensional form by scaling the physical quantities based on a characteristic length, time and density [B21]. For the purpose of lattice computations, a system of *lattice units* is defined by selecting δ_x, δ_t and ρ_0 (i.e., density of an incompressible fluid) as the conversion factors. In lattice units, we set the space step as $\Delta x = 1$, the time step as $\Delta t = 1$, and the average density $\hat{\rho}_0 = 1$.

The discretization of Equation (5.1) in velocity space (using a velocity set), space (using a lattice) and time (using a discrete set of time levels) leads to the lattice Boltzmann equation

$$f_q(\boldsymbol{x} + \Delta t \boldsymbol{\xi}_q, t + \Delta t) - f_q(\boldsymbol{x}, t) = \Delta t \left(C_q(\boldsymbol{x}, t) + \mathcal{S}_q(\boldsymbol{x}, t) \right) \tag{5.4}$$

for all $q \in \{1, \ldots, Q\}$, $\boldsymbol{x} \in \mathcal{L}_\Omega$, and $t \in \mathcal{L}_t$. In Equation (5.4), $f_q = f_q(\boldsymbol{x}, t)$ is the discrete density distribution functions corresponding to the microscopic velocity $\boldsymbol{\xi}_q$, C_q is the discrete collision operator and $\mathcal{S}_q$ is the discrete source term. Note that the collision operator C_q depends on the whole set of Q discrete density distribution functions (i.e., the Q equations written in (5.4) are not independent).

Similarly to the continuous case, moments of the discrete density distribution functions f_q are of interest as they allow to recover the macroscopic quantities. The scheme for the approximation of the Navier–Stokes equations is constructed such that the conserved moments correspond to the macroscopic density ρ, which can be obtained in lattice units in $\mathcal{L}_\Omega \times \mathcal{L}_t$ by

$$\rho(\boldsymbol{x}, t) = \sum_{q=1}^{Q} f_q(\boldsymbol{x}, t) \tag{5.5}$$

and the macroscopic momentum density $\rho \boldsymbol{v}$, which can be computed in lattice units as

$$\rho(\boldsymbol{x}, t) \boldsymbol{v}(\boldsymbol{x}, t) = \sum_{q=1}^{Q} f_q(\boldsymbol{x}, t) \boldsymbol{\xi}_q + \frac{1}{2} \Delta t \hat{\boldsymbol{F}}(\boldsymbol{x}, t), \tag{5.6}$$

where $\hat{\boldsymbol{F}}$ denotes the external force density in lattice units exerted on the fluid. The macroscopic velocity $\boldsymbol{v}$ can be obtained by dividing the macroscopic momentum density $\rho \boldsymbol{v}$ by the macroscopic mass density ρ. Other macroscopic quantities, such as the stress tensor, can be obtained from higher-order, non-conserved moments of the distribution functions [B21].

5.2.3 Collision operator

The term C_q in Equation (5.4) denotes the discrete collision operator; in this thesis we use the cumulant operator proposed in [A84]. The operator has several parameters that drive the relaxation rates of different cumulants towards their equilibria. The relaxation rate ω_1 is linked to the kinematic viscosity ν in the Navier–Stokes equations via

$$\nu = c_s^2 \left(\frac{1}{\omega_1} - \frac{\Delta t}{2} \right) \frac{\delta_x^2}{\delta_t}. \tag{5.7}$$

The other relaxation rates are set as suggested by the authors of the original paper, including the limiters proposed in [A81, A82]. Furthermore, the approximations of the spatial velocity derivatives to reduce the artifacts due to the absence of higher-order cumulants are also incorporated in the collision operator according to [A81, A82].

5.2.4 Equilibrium function

In the kinetic theory of gases, the Maxwell–Boltzmann distribution $f^{\mathrm{eq}}(\boldsymbol{x}, |\boldsymbol{v}|, t)$ describes the speeds of particles in an ideal gas moving with a macroscopic velocity $\boldsymbol{v}$ in a thermodynamic equilibrium [B21]. In the derivation of the discrete lattice Boltzmann equation, the equilibrium distribution function f^{eq} is discretized to obtain an approximation of the equilibrium state for the given discrete velocity set. The discrete equilibrium distribution function f_q^{eq} for the cumulant collision operator can be derived in the space of cumulants following [A84].

5.2.5 Boundary conditions

The formulation of proper boundary conditions for LBM is a major complication in its applications, since the desired macroscopic conditions (e.g., given in terms of ρ and $\boldsymbol{v}$) must be reproduced on the mesoscopic level using the discrete density distribution functions f_q [B21]. This problem does not have a unique solution and thus multiple boundary schemes for LBM exist that approximate the same macroscopic condition. As of this writing, the LBM implementation used by the author contains elementary boundary conditions based on the wet-node equilibrium approach for inflow [A93, B21, A121, A139], extrapolation method for outflow, and full-way bounce-back scheme for solid walls [B21]. Since the analysis of boundary conditions is not within the scope of this thesis, the details are omitted.

5.2.6 Initialization

The simplest initialization approach, which is used in this work, is to set the discrete density distribution functions at $t = 0$ to their equilibria according to the supplied macroscopic density and velocity using

$$f_q(\boldsymbol{x}, 0) = f_q^{\mathrm{eq}}(\rho(\boldsymbol{x}, 0), \boldsymbol{v}(\boldsymbol{x}, 0)) \tag{5.8}$$

for all lattice sites $\boldsymbol{x} \in \mathcal{L}_{\overline{\Omega}}$ and $q \in \{1, \ldots, Q\}$. Unless noted otherwise, the values $\rho(\boldsymbol{x}, 0) = \hat{\rho}_0$ and $\boldsymbol{v}(\boldsymbol{x}, 0) = \boldsymbol{0}$ are used. Different initialization schemes can be found in [B21].

5.3 Streaming schemes

5.3.1 Push and pull schemes with A-B pattern

Based on Equation (5.4), the computational steps needed to evolve the state from the current time t to the next time level $t + \Delta t$ can be decomposed into two successive steps:

1. The *collision* step:

$$f_q^*(\boldsymbol{x}, t) = f_q(\boldsymbol{x}, t) + \Delta t \left(C_q(\boldsymbol{x}, t) + \mathcal{S}_q(\boldsymbol{x}, t) \right), \qquad \forall q \in \{1, \ldots, Q\}, \tag{5.9a}$$

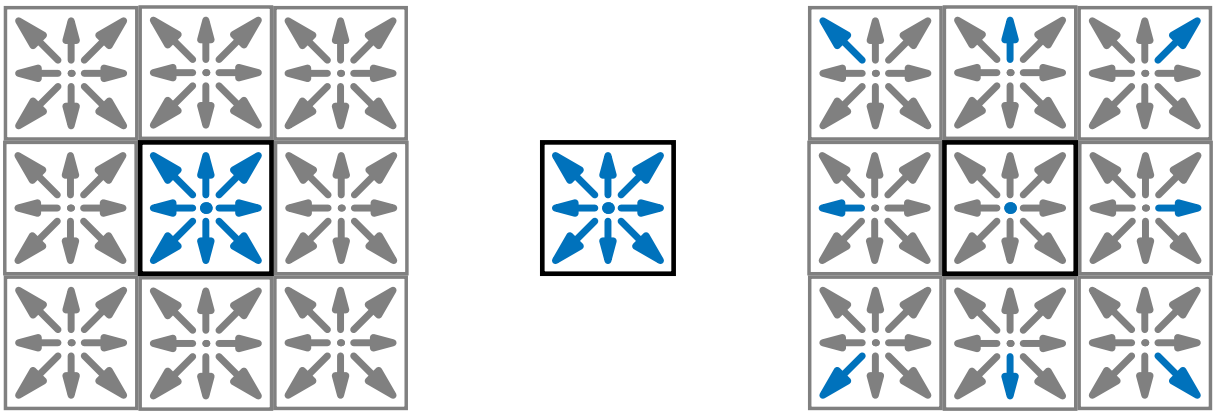
   where f_q^* denotes the value of the distribution function *after* collision in the given lattice site.

2. The *streaming* step:

$$f_q(\boldsymbol{x} + \Delta t \boldsymbol{\xi}_q, t + \Delta t) = f_q^*(\boldsymbol{x}, t), \qquad \forall q \in \{1, \ldots, Q\}, \tag{5.9b}$$

   where the post-collision distribution function f_q^* is propagated to the neighboring lattice site in the direction of the corresponding velocity $\boldsymbol{\xi}_q$.

This scheme is called the *push scheme*, since the streaming step propagates the values from the reference lattice site $\boldsymbol{x} \in \mathcal{L}_\Omega$ to its neighbors. As these two steps alternate in a time loop, it essentially does not matter if the time step is formulated to start with the collision step or the streaming step. The alternative, yet equivalent scheme can be formulated as:

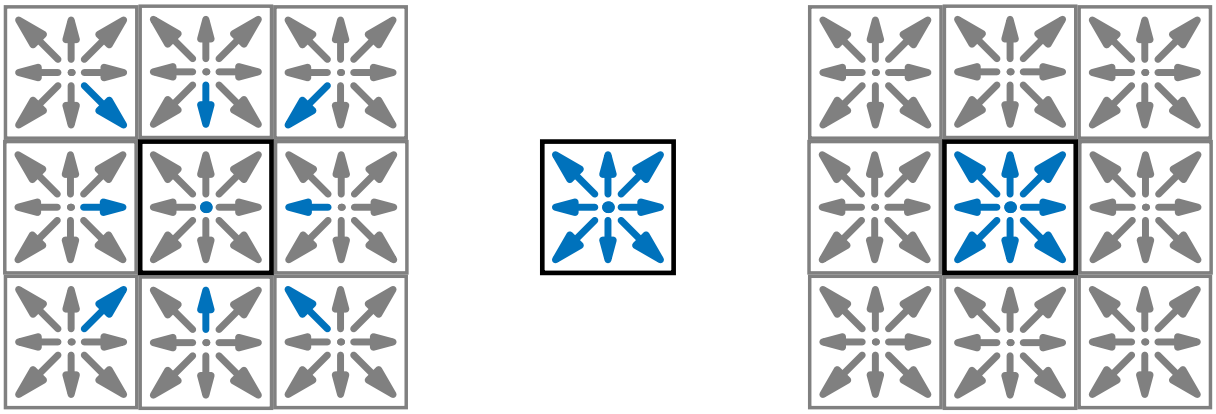


(a) Values read from the global memory (array A)

(b) Values stored in the registers

(c) Values stored into the global memory (array B)

Figure 5.1: Illustration of the *push* streaming scheme with the *A-B pattern*.



(a) Values read from the global memory (array A)

(b) Values stored in the registers

(c) Values stored into the global memory (array B)

Figure 5.2: Illustration of the *pull* streaming scheme with the *A-B pattern*.

1. The *streaming* step:

$$f_q(\boldsymbol{x}, t) = f_q^*(\boldsymbol{x} - \Delta t \boldsymbol{\xi}_q, t - \Delta t), \qquad \forall q \in \{1, \ldots, Q\}, \tag{5.10a}$$

   where the post-collision distribution function f_q^* is propagated from the neighboring lattice site in the direction of $-\boldsymbol{\xi}_q$ from the previous time level.

2. The *collision* step:

$$f_q^*(\boldsymbol{x}, t) = f_q(\boldsymbol{x}, t) + \Delta t \left(C_q(\boldsymbol{x}, t) + \mathcal{S}_q(\boldsymbol{x}, t) \right), \qquad \forall q \in \{1, \ldots, Q\}, \tag{5.10b}$$

   where f_q^* denotes the value of the distribution function *after* collision in the given lattice site.

This scheme is called the *pull scheme*, since the streaming step propagates the values to the reference lattice site $\boldsymbol{x} \in \mathcal{L}_\Omega$ from its neighbors. The visualization of the difference between the push and pull streaming schemes is illustrated in Figures 5.1 and 5.2 for a D2Q9 velocity set.

Note that regardless of the chosen streaming scheme, the collision step is *local* in terms of the lattice sites $\boldsymbol{x} \in \mathcal{L}_\Omega$ (i.e., collisions in different lattice sites are independent of each other), but collective for all

$q \in \{1, \ldots, Q\}$ since the collision operator C_q depends on the whole set of discrete density distribution functions in given lattice site. On the other hand, the operations in the streaming step are *non-local*, but trivial as they do not require any computation (the post-collision values are merely transferred from one place to another).

Both the push and pull schemes require attention during the implementation to avoid overwriting memory locations containing values that may be needed in the future. In the former scheme, the values of f_q at $t + \Delta t$ cannot be pushed to the neighboring lattice sites overwriting the values at current time level t, because the neighboring nodes may be still unprocessed in this time step. Similarly, after the collision step in the pull scheme, the values of f_q^* cannot overwrite the values of f_q in the reference lattice site, because those values may be pulled later when processing neighboring nodes in the same time step. Hence, the classical implementation of LBM based on the push or pull scheme requires the use of two arrays for storing the state of the simulated system. In each time step, an array A is used for input and array B is used for output, and while going to the next time step (i.e., from t to $t + \Delta t$), the arrays A and B are swapped. Hence, this approach is named the *A-B pattern*. The difference between the push and pull schemes is that while the arrays A and B in the push scheme contain the values of f_q at two successive time levels, the arrays in the pull scheme contain the values of f_q^*. However, this is not a problem since these values are not directly visualized, correct values are still used thanks to the placement of the streaming in the time step, and the relevant macroscopic quantities computed just before the collision step can be output for consistent visualization of the results.

5.3.2 A-A pattern

The *A-A pattern* [C6, A180] is an alternative approach which allows to perform the streaming step in-place using just one storage array `A` for the density distribution functions. This feature of the streaming scheme reduces memory requirements for a given lattice by a factor of two, which is crucial for high-resolution LBM simulations on GPU accelerators due to their limited amount of global memory. For $q \in \{1, \ldots, Q\}$ and $\boldsymbol{x} \in \mathcal{L}_{\overline{\Omega}}$, let us denote the access to a value stored in the array `A` as $\mathtt{A}[q, \boldsymbol{x}]$. The time-stepping procedure using the A-A pattern can be formalized as follows:

1. Even iteration ($t \in \{0, 2, 4, \ldots\} \times \Delta t$):

   a) *Pre-collision streaming* step:

   $$f_q(\boldsymbol{x}, t) = \mathtt{A}[q, \boldsymbol{x}], \qquad \forall q \in \{1, \ldots, Q\}, \tag{5.11a}$$

   where the values are read from the array `A` and the direction q is preserved.

   b) The *collision* step:

   $$f_q^*(\boldsymbol{x}, t) = f_q(\boldsymbol{x}, t) + \Delta t \left(C_q(\boldsymbol{x}, t) + \mathcal{S}_q(\boldsymbol{x}, t) \right), \qquad \forall q \in \{1, \ldots, Q\}, \tag{5.11b}$$

   where f_q^* denotes the value of the distribution function *after* collision in the given lattice site.

   c) *Post-collision streaming* step:

   $$\mathtt{A}[\overline{q}, \boldsymbol{x}] = f_q^*(\boldsymbol{x}, t), \qquad \forall q \in \{1, \ldots, Q\}, \tag{5.11c}$$

   where the post-collision values are written to the same array `A` at the same lattice site $\boldsymbol{x}$, but the direction q is inverted: $\overline{q} \in \{1, \ldots, Q\}$ is defined such that $\boldsymbol{\xi}_{\overline{q}} = -\boldsymbol{\xi}_q$.

2. Go to the next iteration ($t := t + \Delta t$).

3. Odd iteration ($t \in \{1, 3, 5, \ldots\} \times \Delta t$):

   a) *Pre-collision streaming* step:

   $$f_q(\boldsymbol{x}, t) = \mathtt{A}[\overline{q}, \boldsymbol{x} - \Delta t \boldsymbol{\xi}_q], \qquad \forall q \in \{1, \ldots, Q\}, \tag{5.11d}$$

   where the values are read from the array A from neighboring lattice sites, but the direction q is inverted for consistency with Equation (5.11c).

   b) The *collision* step:

   $$f_q^*(\boldsymbol{x}, t) = f_q(\boldsymbol{x}, t) + \Delta t \left(C_q(\boldsymbol{x}, t) + \mathcal{S}_q(\boldsymbol{x}, t) \right), \qquad \forall q \in \{1, \ldots, Q\}, \tag{5.11e}$$

   where f_q^* denotes the value of the distribution function *after* collision in the given lattice site.

   c) *Post-collision streaming* step:

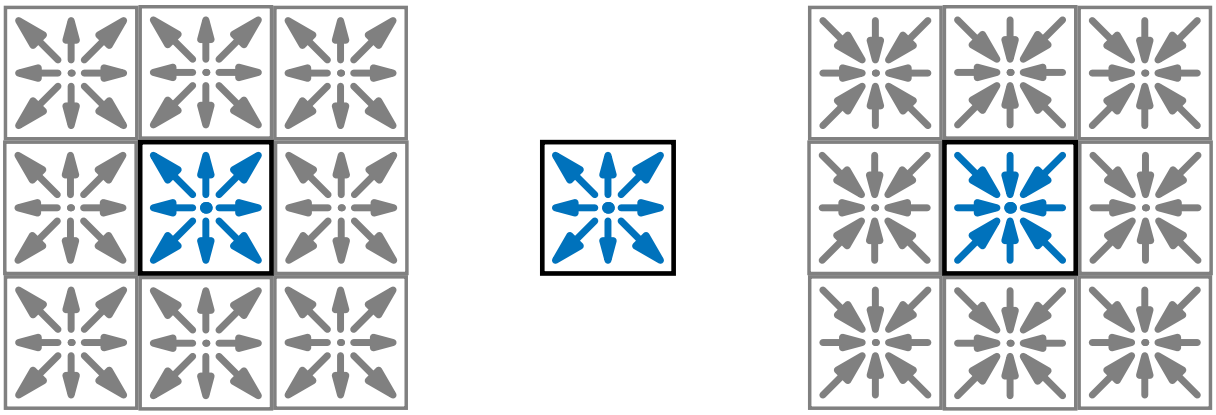
   $$\mathtt{A}[q, \boldsymbol{x} + \Delta t \boldsymbol{\xi}_q] = f_q^*(\boldsymbol{x}, t), \qquad \forall q \in \{1, \ldots, Q\}, \tag{5.11f}$$

   where the values are written to the array A to the neighboring lattice sites and the original direction q is again preserved.

4. Go to the next iteration ($t := t + \Delta t$).

Note that for each lattice site $\boldsymbol{x} \in \mathcal{L}_\Omega$, the steps Equations (5.11d) and (5.11f) access exactly the same memory locations in the array A, since for the whole set of discrete density distribution functions, we can invert the direction q in Equation (5.11d) to obtain $f_{\overline{q}}(\boldsymbol{x}, t) = \mathtt{A}[q, \boldsymbol{x} + \Delta t \boldsymbol{\xi}_q]$, which uses the same array access indices as Equation (5.11f). The operations in the A-A pattern streaming scheme in the even and odd iterations are visualized for a D2Q9 velocity set in Figures 5.3 and 5.4, respectively.

The A-A pattern reduces memory requirements for LBM, but removes the freedom of accessing neighboring values at the current time level separately from the streaming pattern. This causes problems when implementing certain types of boundary conditions, such as those based on extrapolation. The author is not aware of any literature dealing with issues related to correct implementation of boundary conditions using the A-A pattern.

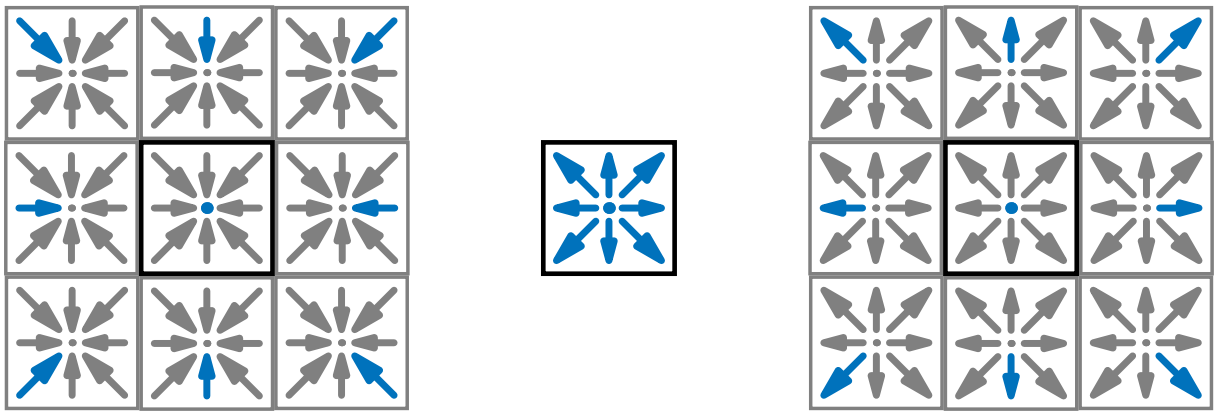


(a) Values read from the global memory (array A)

(b) Values stored in the registers

(c) Values stored into the global memory (array A)

Figure 5.3: Illustration of the *A-A pattern* streaming scheme in the *even iteration*.



(a) Values read from the global memory (array A)

(b) Values stored in the registers

(c) Values stored into the global memory (array A)

Figure 5.4: Illustration of the *A-A pattern* streaming scheme in the *odd iteration*.

5.3.3 Other patterns

The A-A pattern is not the only technique that allows to perform streaming in-place in a single array. Alternatives include, for example, the Esoteric twist [A83], the compact LRnLA streaming pattern [A188], Esoteric push and Esoteric pull [A122]. However, none of the aforementioned papers discuss advanced topics such as MPI communication or the implementation of boundary conditions. Hence, these streaming patterns are not yet implemented in our LBM code.

5.4 Computational algorithm

The time-stepping algorithm for LBM can be generalized to cover all aforementioned streaming patterns. Algorithm 3 summarizes the steps according to our implementation for the A-A pattern and the push/pull schemes with the A-B pattern.

Algorithm 3 (LBM)

1. Initialization: for all $\boldsymbol{x} \in \mathcal{L}_{\overline{\Omega}}$ and $q \in \{1, \ldots, Q\}$, compute $f_q(\boldsymbol{x}, 0)$ using (5.8) and write the results to the A array.
2. Set $t = 0$.
3. While $t < t_{\max}$:

3.1. Compute the external force density $\hat{\boldsymbol{F}}(\boldsymbol{x}, t)$ for all $\boldsymbol{x} \in \mathcal{L}_{\Omega}$.

3.2. Pre-collision step according to the streaming pattern – for all $\boldsymbol{x} \in \mathcal{L}_{\Omega}$ and $q \in \{1, \ldots, Q\}$:

- A-B push scheme: read $f_q(\boldsymbol{x}, t)$ from its natural location in the A array.
- A-B pull scheme: read $f_q(\boldsymbol{x}, t)$ from the A array according to Equation (5.10a).
- A-A pattern: read $f_q(\boldsymbol{x}, t)$ from the A array according to Equation (5.11a) or Equation (5.11d).

3.3. For all $\boldsymbol{x} \in \mathcal{L}_{\Omega}$, compute the macroscopic density $\rho(\boldsymbol{x}, t)$ and velocity $\boldsymbol{v}(\boldsymbol{x}, t)$ using Equations (5.5) and (5.6).

3.4. Handle boundary conditions for all relevant lattice sites in $\mathcal{L}_{\overline{\Omega}}$.

3.5. Collision step: compute $f_q^*(\boldsymbol{x}, t)$ for all $\boldsymbol{x} \in \mathcal{L}_{\Omega}$ and $q \in \{1, \ldots, Q\}$ using Equation (5.9a) (this step is the same in all streaming patterns).

3.6. Post-collision step according to the streaming pattern – for all $\boldsymbol{x} \in \mathcal{L}_\Omega$ and $q \in \{1, \ldots, Q\}$:

- A-B push scheme: write $f_q^*(\boldsymbol{x}, t)$ to the `B` array according to Equation (5.9b).
- A-B pull scheme: write $f_q^*(\boldsymbol{x}, t)$ to its natural location in the `B` array.
- A-A pattern: write $f_q^*(\boldsymbol{x}, t)$ to the `A` array according to Equation (5.11c) or Equation (5.11f).

3.7. Write the previously computed values of ρ and $\boldsymbol{v}$ to the global memory. When applicable, compute all other predetermined macroscopic quantities for visualization.

3.8. Go to the next time step: $t := t + \Delta t$. If the A-B pattern is used, swap the `A` and `B` arrays.

All steps in Algorithm 3 involving the iteration over lattice sites provide a natural way to parallelize the algorithm. While each such step could be implemented in a separate parallel loop (e.g., launching a separate CUDA kernel), it is desirable to merge all steps in the time loop into a single parallel loop to avoid a performance penalty due to saving and loading the intermediate results. This is already indicated in Algorithm 3 which includes explicit read and write operations and it is assumed that intermediate results are readily available to the following steps. More formally, Algorithm 4 summarizes the steps with an explicitly highlighted parallel loop.

Algorithm 4 (Parallel LBM)

1. Initialization: perform the step 1 in Algorithm 3, but in parallel for all $\boldsymbol{x} \in \mathcal{L}_{\overline{\Omega}}$.
2. Set $t = 0$.
3. While $t < t_{\max}$:

3.1. For all $\boldsymbol{x} \in \mathcal{L}_{\overline{\Omega}}$ in parallel:

- Perform the steps 3.1 to 3.7 from Algorithm 3, but only for the given lattice site $\boldsymbol{x}$.

3.2. Go to the next time step (step 3.8 in Algorithm 3).

Algorithm 4 is applicable to parallel architectures that utilize a single shared memory, such as multicore CPUs within a single node or a single GPU accelerator. In the latter case, the algorithm needs to be extended with data copies to and from the device memory due to input and output of relevant macroscopic quantities, but these steps are omitted in Algorithm 4 for clarity.

In order to utilize multiple GPUs or even multiple nodes for computing in a distributed fashion, an additional level of coarse-grained parallelism has to be introduced based on the domain decomposition approach. The global lattice $\mathcal{L}_{\overline{\Omega}}$ is split into subdomains $\mathcal{L}_{\overline{\Omega},j}$, $j \in \{1, \ldots, N_{\text{parts}}\}$ which are processed independently using $N_{\text{ranks}} \leq N_{\text{parts}}$ independent processes (MPI ranks). Each subdomain is assigned to one MPI rank that is mapped to one coarse-grain computational unit, such as a CPU core, a CPU socket, or a GPU accelerator. Each MPI rank is responsible for processing computations related to one or more subdomains that were assigned to it. In order to obtain consistent results, it must be ensured that the values of discrete density distribution functions are propagated correctly across the interfaces between subdomains. However, due to the distributed memory system, this cannot happen automatically as part of the streaming steps in Algorithm 4. The solution adopted in our implementation is to extend each subdomain $\mathcal{L}_{\overline{\Omega},j}$ with *ghost lattice sites* which are excluded from parallel processing, but provide additional storage in the `A` and `B` arrays where other lattice sites will write and/or read values that will be later streamed to/from the neighboring subdomains. After the parallel loop in each time step, parts of the `A` (or `B`) array corresponding to the ghost lattice sites are exchanged via MPI among neighboring subdomains. This approach is summarized below in Algorithm 5.

Algorithm 5 (Distributed LBM)

1. For each subdomain $\mathcal{L}_{\overline{\Omega},j}$:

- Perform initialization for all $\boldsymbol{x} \in \mathcal{L}_{\overline{\Omega},j}$ in parallel, excluding the ghost lattice sites (step 1 in Algorithm 3).

2. Copy distribution functions in the A array on the interfaces between neighboring subdomains.
3. Set $t = 0$.
4. While $t < t_{\max}$:

4.1. For each subdomain and for all $\boldsymbol{x} \in \mathcal{L}_{\overline{\Omega},j}$ (excluding the ghost lattice sites) in parallel:

- Perform the steps 3.1 to 3.7 from Algorithm 3, but only for the given lattice site $\boldsymbol{x}$.

4.2. Copy distribution functions in the output array (A or B) on the interfaces between neighboring subdomains.

4.3. Go to the next time step (step 3.8 in Algorithm 3).

First of all, notice that for the LBM algorithm it is not necessary to exchange the full set of Q density distribution functions between each pair of neighboring processes. Taking the geometric interpretation of the velocity set into account, the communication size can be significantly reduced: for example, for $\boldsymbol{\xi}_q = [1, 0, 0]^T$, the values of f_q need to be transferred to the right neighbor, but not to the left. In case of the D3Q27 velocity set and a 1D domain decomposition where each subdomain has at most two neighbors (one to the left and one to the right), only 9 out of 27 density distribution functions need to be streamed from one subdomain to another.

However, even with the aforementioned optimization, a naive implementation of Algorithm 5 does not scale with increasing number of MPI ranks. In order to obtain a scalable distributed LBM algorithm, an additional optimization is needed: it is necessary to *overlap* computation with communication in order to hide the extra latency which was not present in Algorithm 4. This can be done by processing the boundary lattice sites of each subdomain separately from the interior lattice sites, copying the data between subdomains as soon as the computation on the boundary has finished, and processing the interior lattice sites independently while all MPI ranks exchange their boundary data. This approach is summarized in Algorithm 6 using *asynchronous operations*.

Algorithm 6 (Distributed LBM with overlapped computation and communication)

1. Initialization (step 1 in Algorithm 5).
2. Copy distribution functions in the A array on the interfaces between neighboring subdomains.
3. Set $t = 0$.
4. While $t < t_{\max}$:

4.1. For all subdomains $\mathcal{L}_{\overline{\Omega},j}$ in parallel:

4.1.1. Start processing *lattice sites next to the interfaces with other subdomains*.

4.1.2. Start processing *remaining lattice sites*.

4.2. Wait until the operations started in step 4.1.1 are finished for all subdomains.

4.3. Copy distribution functions in the output array (A or B) on the interfaces between neighboring subdomains.

4.4. Wait until the operations started in step 4.1.2 are finished for all subdomains.

4.5. Go to the next time step (step 3.8 in Algorithm 3).

The steps 4.1.1 and 4.1.2 in Algorithm 6 involve the same local operations as the step 3.1 in Algorithm 4, but on different sets of lattice sites. Likewise, all steps 2 and 4.2 in Algorithm 5, and steps 2 and 4.3 in Algorithm 6, perform exactly the same operations that are explained further in Section 5.6.

5.5 Implementation remarks

This section covers practical aspects related to the LBM implementation in our code base [O21]. It focuses primarily on the data structures needed in LBM and on the templated object-oriented design of our LBM implementation. Performance optimizations are described in Section 5.6.

Several multidimensional arrays are needed to store relevant values in the LBM implementation:

- *Distribution functions* are stored in a 4-dimensional array with the size $Q \times N_x \times N_y \times N_z$. Depending on the streaming scheme, either one (`A`) or two (`A` and `B`) such arrays are necessary.
- *Macroscopic quantities* are stored in a 4-dimensional array with the size $N_m \times N_x \times N_y \times N_z$, where N_m is the number of macroscopic quantities stored during the simulation. In case of the Navier–Stokes equations, at least 4 quantities (density and three components of velocity) need to be stored, but additional user-defined quantities can be added.
- *Lattice site tags* are stored in a 3-dimensional array with the size $N_x \times N_y \times N_z$ and the type of elements `short int`. The values in this array determine the *type* of each lattice site and they are used to identify different boundary conditions and geometry of immersed bodies.

The data structure used for all of these arrays, `TNL::Containers::DistributedNDArray`, has been described in Section 2.1. Recall that the global array corresponding to the lattice $\mathcal{L}_{\overline{\Omega}}$ must be decomposed in a *structured conforming* manner, but each rank can be assigned multiple subdomains and the rank numbering is arbitrary. The decomposition of the aforementioned arrays in LBM must be consistent. Hence, we collect all distributed data structures in a class called `LBM_BLOCK` that contains all data associated to one subdomain: local subarrays, subdomain sizes and offsets, indices of neighboring blocks and ranks that own them.

To facilitate computations on GPU accelerators, two objects for each array are needed to represent data in different memory spaces. One object allocates the array data in the host memory where it can be processed by the CPU, and the other object allocates the array data in the device memory where it can be processed by the GPU accelerator. The main part of the LBM (i.e., steps 4.1.1 and 4.1.2 in Algorithm 6) is computed on GPU accelerators, but CPU processing is used for initialization and output of the results. During the computation of the LBM algorithm, relevant arrays must be copied from one memory space to another whenever processing is switched from CPU to GPU or vice versa. Furthermore, additional attribute represented by the `LBM_DATA` class is added to `LBM_BLOCK`, which contains the data that need to be passed to the CUDA kernel when the GPU starts processing the block. This includes pointers to the arrays allocated in the device memory, sizes of the lattice, and scalar parameters (e.g., viscosity and parameters related to boundary conditions).

The relation between the `LBM_DATA` and `LBM_BLOCK` classes is highlighted in Figure 5.5 along with additional classes that will be described later. The `LBM_BLOCK` objects representing individual subdomains managed by the current MPI rank are aggregated in an `std::vector` container in the class named `LBM`. The `LBM` class provides member functions to manipulate all managed blocks collectively, as well as additional functionality related to LBM simulations, such as variables for time-stepping, or conversion between physical and lattice units.

From a high level point of view, each MPI rank creates one object of the `State` class, which combines all components needed to run a simulation. The default implementation of `State` contains one object of the `LBM` class to represent one set of discrete distribution functions for a Navier–Stokes problem. The `State` class is designed to be extended via inheritance, so users can define their own subclass for a particular case and adjust the behavior of the solver by adding attributes and overriding various virtual member functions. Hence, it is possible to implement more complicated solvers for coupled problems, such as the Navier–Stokes–Fourier problem of fluid flow coupled with thermal transport [A52, A155],

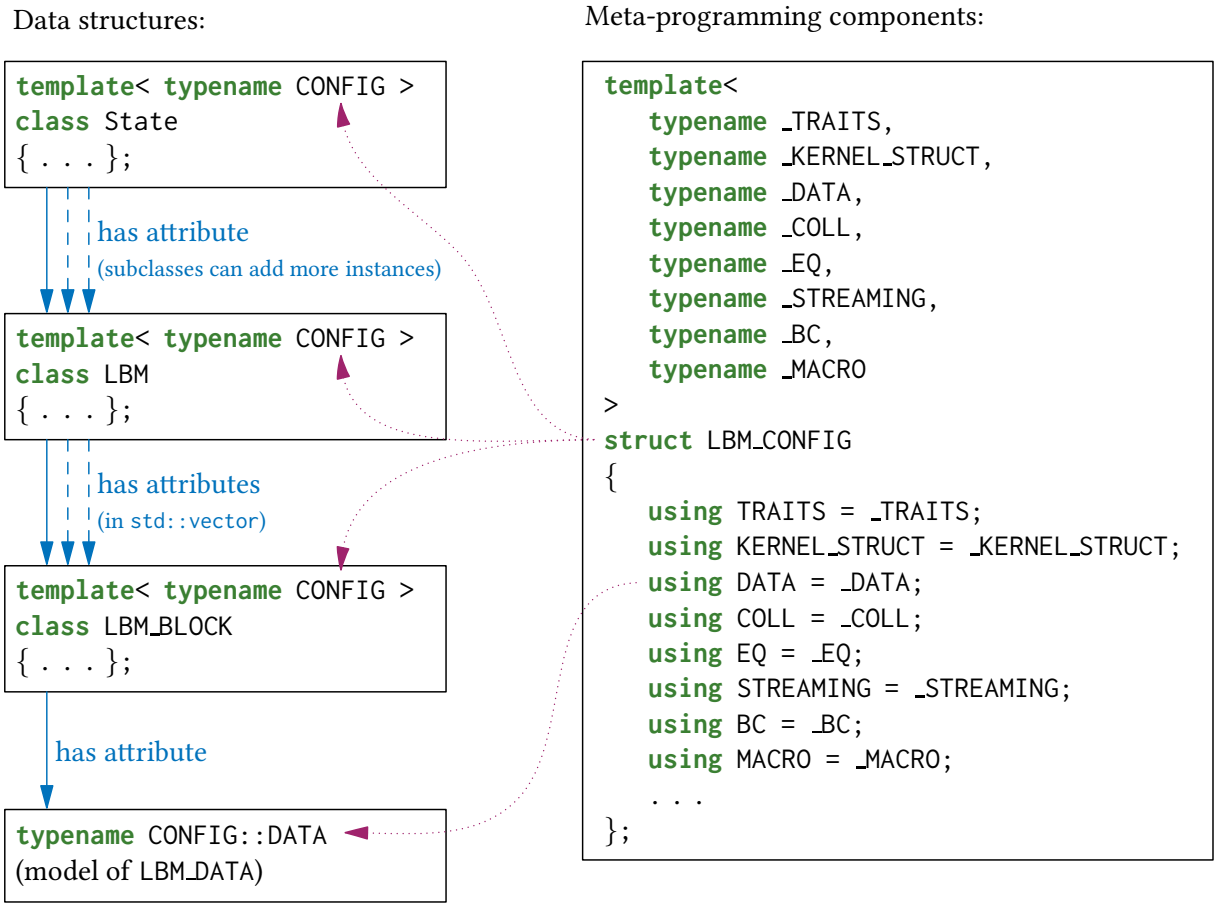


Figure 5.5: Schematic diagram showing the relations between the main C++ classes and meta-programming components in the LBM implementation.

which can be solved using multiple sets of discrete distribution functions represented by objects of the `LBM` class.

Up to now, the description of the implementation revolved around data structures and classes, leading to the `State` class that provides extensibility via subclassing. This approach is suitable for high-level adaptions that reuse the existing low-level building blocks. However, it is often necessary to modify even the low-level details, such as local per-lattice-site computations in the LBM algorithm. For example, it may be desirable to switch to a different collision operator or to define additional macroscopic quantities. In order to sustain high-performance computing on GPU accelerators that have limited support for virtual functions [M19], using compile-time techniques rather than dynamic polymorphism is more appropriate. Therefore, our LBM implementation is based on several template meta-programming components that allow to configure the low-level details while preserving the generation of optimal CUDA kernels for each configuration.

The starting element of template meta-programming used in the code is the `LBM_CONFIG` template class that represents a particular configuration for LBM. As shown in Figure 5.5, it combines the following components that are set via template parameters:

- `TRAITS` is a structure that holds the definition of fundamental types used in the solver, such as the floating-point type, index type, permutations of the multidimensional arrays, etc.

- `KERNEL_STRUCT` is a structure that is used in the CUDA kernel to store local data related to the given lattice site. In particular, it contains an array of Q discrete distribution functions that are read and written during the streaming steps.
- `DATA` is a structure that refers to the `LBM_DATA` class, or its user-defined subclass. Its purpose is to pass data from the host to the CUDA kernel. Besides the attributes contained in `LBM_DATA`, it may be used to provide data for user-defined extensions of the following components.
- `COLL` is a structure that implements a particular collision operator. The action of the operator is computed in its static member function `collision` using the values in an instance of `KERNEL_STRUCT`.
- `EQ` is a structure that implements a particular discrete equilibrium function.
- `STREAMING` is a structure that implements a particular streaming scheme.
- `BC` is a structure that defines tags that may be assigned to lattice sites and implements actions to be taken in the CUDA kernel based on these tags. It is intended for user modification in order to specify boundary conditions for a particular simulation.
- `MACRO` is a structure that implements the computation of macroscopic quantities.

Only the `KERNEL_STRUCT` and `DATA` components are intended to represent data, the other components are never instantiated and serve merely to define types or algorithms in terms of static member functions. The code base provides multiple models for each component with a sensible default behavior. Users can subclass each model, implement their modifications and pass their type to `LBM_CONFIG`.

All the aforementioned classes `LBM_BLOCK`, `LBM`, and `State` are in fact templates with one parameter, `CONFIG`, as shown in Figure 5.5. The modifications of the components such as `BC` or `MACRO` may need to be designed together with the extensions of `State`, especially when custom attributes are added to the `DATA` component.

5.6 Optimization remarks

The most important optimization for distributed LBM computations has already been outlined in Algorithm 6: processing interior lattice sites must be overlapped with data synchronization on the interfaces between neighboring subdomains. This technique is implemented by launching multiple CUDA kernels in different CUDA streams and using stream-level synchronization functions [M19]. A separate CUDA stream is used for each interface with a neighboring subdomain and for the interior lattice sites. The streams for interfaces are configured with the highest priority so that the operations in these streams are executed as soon as possible. The stream for the interior is configured with the lowest priority as it is used for the longest-running kernel that is waited for in the end of the time step.

Following the processing of lattice sites at the interfaces, data synchronization among the neighboring ranks starts. An object of the `DistributedNDArraySynchronizer` class that was described in Section 2.1.3 is created for each subdomain and each of the Q discrete distribution functions. Each synchronizer object is responsible for synchronizing one subarray to and from the neighbors in the direction of the corresponding vector from the velocity set. For example, when the D3Q27 velocity set is used and the lattice is distributed in a 1D manner such that each subdomain has at most two neighbors, one to the left and one to the right, 9 distribution functions (corresponding to discrete velocities with +1 in the first component) are synchronized from left to right on each interface, and different 9 distribution functions (corresponding to discrete velocities with −1 in the first component) are synchronized from right to left. Note that in case of a 2D or 3D decomposition, the communication pattern is more complicated as it is necessary to also synchronize relevant distribution functions across edges and corners of the interfaces between subdomains.

The synchronizer objects are used in Algorithm 6 in the steps 2 and 4.3. The synchronization procedure corresponds to Algorithm 1 on Page 27 with *pipelining* among all synchronizers as mentioned in Section 2.1.3. More formally, the pipelined synchronization as executed by one MPI rank is summarized in Algorithm 7.

Algorithm 7 (Pipelined synchronization of distributed multidimensional arrays)

1. For all synchronizers of the current MPI rank:
   - Allocate all send and receive buffers (relevant only in the first iteration).
   - Start copying data from the local array to the send buffers.
2. For all synchronizers of the current MPI rank:
   - Wait until all data is copied to the send buffers.
   - Start MPI communications with all neighbors via `MPI_Isend` and `MPI_Irecv`.
3. For all synchronizers of the current MPI rank:
   - Wait for all MPI communications with the neighbors to complete.
   - Start copying data from the receive buffers to the local array.
4. For all synchronizers of the current MPI rank:
   - Wait until all data is copied to the local array.

Note that the implementation relies on the so-called *CUDA-aware* support [O18] from the MPI library in order to avoid having to explicitly copy the send and receive buffers in the steps 1 and 3 to the host memory. However, the MPI library may still need to internally copy data between the device and host memory when handling the `MPI_Isend` and `MPI_Irecv` calls. This was actually the case in all simulations performed for this thesis, because we did not have access to any system with functional NVIDIA GPUDirect technology [M20].

Each copy operation started in the steps 1 and 3 involves launching a CUDA kernel that is placed into the same high-priority CUDA stream that was previously used for the processing of the corresponding lattice sites. However, as mentioned in Section 2.1.3, these copy operations can be avoided when the data is already stored in contiguous blocks of memory. In general, this can be ensured only with a 1D distribution of the lattice and with an appropriate storage layout for the underlying four-dimensional $Q \times N_x \times N_y \times N_z$ array:

- The Q-dimension is ordered first such that three-dimensional $N_x \times N_y \times N_z$ subarrays for each discrete distribution function are stored contiguously.
- The x-, y-, and z-dimensions are ordered depending on the dimension along which the lattice is decomposed. If it is decomposed along the x-dimension, i.e., each subdomain has at most two neighbors, one to the left and one to the right, the array is ordered such that the $N_y \times N_z$ slices of the array for each x-coordinate are stored contiguously. Furthermore, each 1D slice of the length N_y along the y-dimension is stored contiguously.

Based on the layout of the four-dimensional array storing the values of discrete distribution functions and assuming the aforementioned 1D decomposition along the x-dimension, the CUDA thread block size for LBM computations is selected as $(1, B_y, B_z)$, where the parameters B_y and B_z are determined by Algorithm 8 based on the lattice sizes N_y and N_z to achieve coalesced memory accesses [M19]. The algorithm does not impose any constraints on N_y and N_z and assumes that the CUDA kernel

checks for each thread if it has a corresponding lattice site or is out of bounds. The thread block size is selected such that B_y is a multiple of 32 (i.e., the *warp size* on contemporary NVIDIA GPUs [M19]) and $B_y B_z \leq$ `max_threads`, a limit that we set empirically as `max_threads` = 256 for a single-precision data type and `max_threads` = 128 for a double-precision data type.

Algorithm 8 (CUDA thread block size selection for LBM)

1. Set $B_y :=$ `max_threads`, $B_z := 1$.
2. While $B_y > 32$ and $N_y \leq B_y/2$:
2.1. Set $B_y := B_y/2$.
2.2. If $B_z < N_z$, set $B_z := 2B_z$.
3. Return $(1, B_y, B_z)$.

5.7 Computational benchmark results

Since the verification and validation of the LBM implementation was performed primarily by other research team members [A23, A63, A64, A69], this section presents only the performance evaluation, which is the author's original work.

The benchmark problem used in this section simulates a laminar flow in a rectangular duct for which an analytical solution can be derived [B34]. The domain $\Omega = (0, L_x) \times (0, L_y) \times (0, L_z)$ has an inflow boundary on the left hand side (i.e., $x = 0$), outflow boundary on the right hand side (i.e., $x = L_x$), and solid walls with the no-slip boundary condition on the remaining sides. The inflow velocity profile is prescribed as $\boldsymbol{v}(0, y, z) = [v_{\mathrm{an},x}(y, z), 0, 0]^T$, where $v_{\mathrm{an},x}$ denotes the x-component of the velocity in the analytical solution. The kinematic viscosity is set as $\nu = 1.5 \times 10^{-5}\,\mathrm{m^2\,s^{-1}}$ and the final simulation time is $t_{\max} = 100\,\mathrm{s}$.

The typical metric for performance evaluation of LBM is GLUPS (i.e., giga-LUPS, *billions of lattice updates per second*). The value of GLUPS is calculated by taking the number of lattice sites $N_x \times N_y \times N_z$, multiplying it by a fixed number of iterations, and dividing it by the computational time needed for the simulation to complete the given number of steps. Running values of GLUPS can be calculated during the simulation by measuring the time needed to complete a predefined part of the simulation (e.g., a given number of iterations). The values reported in this section were calculated by taking the total number of iterations in the whole simulation and the corresponding computational time (i.e., without the time spent in initialization, data output, etc.). The GLUPS values are comparable among different lattice resolutions and hardware platforms used for the computations, but they are bound to the underlying configuration of LBM. Most notably, the results reported in this section are specific to the D3Q27 velocity set and the cumulant collision operator.

5.7.1 Comparison of streaming patterns

As the first result, we compare the performance of two streaming patterns in our implementation: the A-B pull scheme and the A-A pattern (see Section 5.3). The benchmark was computed on a fixed domain with $L_x = L_y = L_z = 0.25\,\mathrm{m}$, using single or double precision, and without MPI distribution (i.e., using one MPI rank). The same simulation was computed on several NVIDIA GPU accelerators from the three latest architectures (Pascal, Volta/Turing, Ampère) and two product lines (GeForce, Tesla). The solver was compiled with the `nvcc` compiler version 11.8 using the flags `-std=c++17 -O3 --use_fast_math` and the host compiler `g++` version 11.3.0.

The results included in Table 5.1 show that in all cases, the difference between the A-B pull scheme and the A-A pattern in terms of performance is negligible. For the GeForce line, the values of GLUPS in single precision are approximately proportional to the maximum global memory throughput of the

Table 5.1: Performance comparison of the A-B pull scheme with the A-A streaming pattern in single and double precision on various NVIDIA GPU architectures. The lattice size is $256 \times 256 \times 256$ in all cases.

	GLUPS in single precision		GLUPS in double precision	
GPU	A-B pull scheme	A-A pattern	A-B pull scheme	A-A pattern
NVIDIA GeForce GTX 1080 Ti	1.397	1.453	0.173	0.173
NVIDIA GeForce RTX 2070	1.380	1.529	0.134	0.134
NVIDIA GeForce RTX 3060	1.411	1.411	0.096	0.096
NVIDIA Tesla P100	1.836	2.021	1.028	1.011
NVIDIA Tesla V100	2.853	3.053	1.529	1.467
NVIDIA Tesla A100	5.163	5.163	2.726	2.775

Table 5.2: Hardware specifications of accelerated compute nodes in the Karolina supercomputer [O15], operated by IT4Innovations (https://www.it4i.cz/).

Number of nodes	72
Processors per node	2
CPU model	AMD EPYC 7763 (64 cores, 2.45-3.5 GHz)
Memory per node	1024 GB DDR4 3200 MT/s
Accelerators per node	8
GPU model	NVIDIA A100 (40 GB HBM2 memory)
Intra-node connection	NVLink 3.0 (12 sub-links, 25 GB/s per sub-link per direction)
Inter-node connection	4× 200 Gb/s InfiniBand ports

GPU, while in double precision they are approximately proportional to the computational performance (FLOPS) of the GPU. This suggests that LBM in single precision is a memory-bound application on these platforms, while LBM in double precision is compute-bound (note that GPUs in the GeForce line have severely limited performance in double precision [M19]). For the Tesla line, where the ratio between the performance in single and double precision is 2 for each GPU, the GLUPS values for single as well as double precision are approximately proportional to the maximum memory bandwidth. For single precision, the proportion matches even the GeForce line. LBM is therefore a memory-bound application on Tesla GPU accelerators in single as well as double precision.

Overall, based on the results in Table 5.1, it is evident that the A-A pattern provides an efficient way to reduce the memory requirements of LBM with negligible impact on the computational performance. Hence, the A-A pattern will be used in all LBM simulations presented hereafter.

5.7.2 Scaling on the Karolina supercomputer

The following results show the performance scaling of distributed LBM computations on the Karolina supercomputer [O15], operated by IT4Innovations in Ostrava. The system comprises 72 accelerated compute nodes with hardware specifications listed in Table 5.2. The solver was compiled with the `nvcc` compiler version 11.6 using the flags `-std=c++17 -O3 --use_fast_math` and the host compiler `nvc++` version 22.2. The MPI implementation was OpenMPI version 4.1.2. All software components were loaded using the environment module `OpenMPI/4.1.2-NVHPC-22.2-CUDA-11.6.0`.

We performed a strong scaling analysis and two types of weak scaling analysis (explained below) using a 1D decomposition of the domain with $N_{\text{ranks}} = N_{\text{parts}}$ (i.e., each MPI rank managed one subdomain). The number of GPUs used in the studies varied from 1 to 128 (i.e., 1 to 16 nodes) and

Table 5.3: Strong scaling in single and double precision on the Karolina supercomputer. The lattice size is $512 \times 512 \times 512$ in all cases. Each MPI rank was mapped to its own GPU accelerator.

		single precision			double precision		
N_{nodes}	N_{ranks}	GLUPS	Sp	Eff	GLUPS	Sp	Eff
1	1	5.2	1.0	1.00	2.8	1.0	1.00
1	2	10.2	2.0	0.98	5.5	2.0	1.00
1	4	20.4	3.9	0.98	11.1	4.0	1.01
1	8	41.1	7.9	0.99	22.3	8.1	1.01
2	16	80.4	15.5	0.97	44.1	16.0	1.00
4	32	145.2	28.0	0.87	85.5	31.0	0.97
8	64	258.6	49.8	0.78	153.7	55.7	0.87
16	128	301.1	58.0	0.45	225.1	81.6	0.64

each rank always used its own separate GPU. The scaling is evaluated based on the GLUPS metric as defined above. For parallel computations using multiple GPUs, we define the speed-up Sp as the ratio between GLUPS using N_{ranks} and 1 GPUs, and parallel efficiency Eff, which quantifies the scalability of the computation and is defined as the speed-up divided by the number of GPUs, i.e.

$$Eff = \frac{\text{GLUPS for } N_{\text{ranks}} \text{ GPUs}}{N_{\text{ranks}} \times (\text{GLUPS for 1 GPU})}.$$

In the strong scaling analysis, the global lattice size is kept constant and with increasing N_{ranks}, the subdomains become smaller. Table 5.3 shows the results on a $512 \times 512 \times 512$ lattice where the best strong scaling efficiency was achieved. The corresponding problem size is approximately 16 GB in single precision and 32 GB in double precision, so it cannot be increased much further in order to fit into the 40 GB memory of a single GPU. The physical dimensions of the domain are $L_x = L_y = L_z = 0.25\,\text{m}$. According to the results from Table 5.3, the computations are strongly scalable with perfect efficiency up to two nodes and then the efficiency starts to drop. When only GPUs from a single node are used, all communication goes through NVLink, which is the fastest interconnection between GPUs that is available on the Karolina supercomputer. When two nodes are used, two ranks (with IDs 7 and 8) must exchange data over the InfiniBand interconnection between nodes. When more than two nodes are used, some nodes must exchange the same amount of data with multiple nodes. For example, with four nodes (with IDs 0, 1, 2, 3) and eight ranks per node (with IDs 0 to 31), ranks 8 and 15 from node 1 exchange data with rank 7 from node 0 and rank 16 from node 2, respectively. Also note that due to the 1D decomposition used in this strong scaling study, the computation-to-communication ratio decreases with increasing number of ranks, since the amount of communication per rank is constant, but the subdomains become smaller. Hence, the communication cost becomes dominant as it eventually cannot be overlapped completely with the computation using Algorithm 6 on Page 88. In the last computations using 128 ranks, the size of each subdomain is $4 \times 512 \times 512$ and the amount of communication corresponds to $1/2$ of the subdomain for all ranks except the first and the last where it corresponds to $1/4$. The resulting parallel efficiency reported in Table 5.3 is 0.45 for single precision and 0.64 for double precision. We suppose that this degenerate case could be significantly improved with a multidimensional decomposition of the domain.

In the first weak scaling analysis, the lattice size of each subdomain is kept constant and with increasing N_{ranks}, the whole domain becomes larger. Furthermore, the computation-to-communication ratio is kept constant as well. Table 5.4 demonstrates almost perfect weak scaling for the case where the size of each subdomain is $256 \times 256 \times 256$ and the subdomains are arranged along the x-axis, resulting in global lattice with the size $256\,N_{\text{ranks}} \times 256 \times 256$. We have also achieved the same parallel efficiency

Table 5.4: Weak scaling with 1D domain expansion in single and double precision on the Karolina supercomputer. The lattice size is scaled as $256\,N_{\text{ranks}} \times 256 \times 256$. Each MPI rank was mapped to its own GPU accelerator.

		single precision			double precision		
N_{nodes}	N_{ranks}	GLUPS	Sp	Eff	GLUPS	Sp	Eff
1	1	5.2	1.0	1.00	2.8	1.0	1.00
1	2	10.2	2.0	0.99	5.5	2.0	0.99
1	4	20.4	4.0	0.99	11.0	4.0	0.99
1	8	40.9	7.9	0.99	22.0	8.0	0.99
2	16	81.8	15.8	0.99	44.1	15.9	0.99
4	32	163.4	31.7	0.99	88.2	31.8	0.99
8	64	326.8	63.4	0.99	176.4	63.6	0.99
16	128	653.9	126.7	0.99	352.8	127.3	0.99

Table 5.5: Weak scaling with 3D domain expansion in single and double precision on the Karolina supercomputer. The lattice size is scaled as $N_x = N_y = N_z = \left\lfloor 512\sqrt[3]{N_{\text{ranks}}} \right\rceil$ in single precision and $N_x = N_y = N_z = \left\lfloor 256\sqrt[3]{N_{\text{ranks}}} \right\rceil$ in double precision. Each MPI rank was mapped to its own GPU accelerator.

		single precision				double precision			
N_{nodes}	N_{ranks}	N_x	GLUPS	Sp	Eff	N_x	GLUPS	Sp	Eff
1	1	512	5.2	1.0	1.00	256	2.8	1.0	1.00
1	2	645	9.5	1.8	0.91	323	5.3	1.9	0.95
1	4	813	19.0	3.7	0.92	406	10.6	3.8	0.96
1	8	1024	40.9	7.9	0.98	512	22.4	8.1	1.01
2	16	1290	74.3	14.3	0.89	645	40.4	14.6	0.91
4	32	1625	153.8	29.6	0.93	813	82.9	30.0	0.94
8	64	2048	324.0	62.4	0.98	1024	176.2	63.7	1.00
16	128	2580	604.9	116.5	0.91	1290	301.3	108.9	0.85

even with larger subdomains (such as $512 \times 512 \times 512$), but not with smaller subdomains (such as $128 \times 128 \times 128$) that are presumably too small to saturate dozens of GPU accelerators. However, note that such weak scaling analysis is not practical, since the physical size of the domain is different in each case (here $L_x = 0.25 N_{\text{ranks}}$ m and $L_y = L_z = 0.25$ m). A more realistic weak scaling study is described below.

In the second weak scaling analysis, the primary requirement is to keep the physical size of the domain constant. In this study, we fix $L_x = L_y = L_z = 0.25$ m, same as in the strong scaling study. The global discrete problem size is scaled in all three spatial dimensions such that the lattice size $N_x \times N_y \times N_z$ is (approximately) proportional to the number of ranks. The lattice size is scaled according to the formula

$$N_x = N_y = N_z = \left\lfloor s\sqrt[3]{N_{\text{ranks}}} \right\rceil, \tag{5.12}$$

where $\lfloor \cdot \rceil$ denotes rounding to the nearest integer and s is a scaling parameter. For the results presented in Table 5.5, we chose $s = 512$ for single precision in order to obtain the largest possible base lattice ($512 \times 512 \times 512$ for one rank) where the best weak scaling efficiency was achieved. For double precision, we had to use a smaller base lattice with $s = 256$ due to memory limitations. The reason is that in this

study, the communication size per rank is increasing (it is proportional to $N_y \times N_z$) and storage for data received from neighboring ranks must be allocated, so with $s = 512$ and $N_{\text{ranks}} \geq 32$ the amount of memory needed per rank would be more than the GPU accelerators have available. It can be noticed in Table 5.5 that *Eff* does not behave monotonically: for 1, 8, and 64 ranks it is close to 1, but otherwise it is significantly lower. However, this problem is not due to the communication cost—the communication is completely overlapped with computation, but the computation itself is slower than it should be. The CUDA thread block size selected by Algorithm 8 is $(1, B_y, 1)$ for all lattice sizes used in this study, where $B_y = 256$ for single precision and $B_y = 128$ for double precision. It can be noticed that the performance of the LBM algorithm is decreased for cases where N_y is not a multiple of B_y due to inactive threads compared to the optimal case where N_y is a multiple of B_y.

Note that all results presented in this section were computed without utilizing the NVIDIA GPUDirect technology [M20] which is still not fully functional on the Karolina supercomputer. Hence, the effective bandwidth of inter-node communication was limited, because data could not be transferred from the GPU memory directly to the InfiniBand network interface and had to be buffered in the operating system memory instead. Another problem is that in each computation, InfiniBand was used in multi-rail configuration [O38] with two ports, but the same ports were shared by all ranks on a node and the other two InfiniBand ports remained completely unused. We are not certain if this is a hardware or software problem and how it should be addressed. The observed maximum bandwidth between nodes was approximately 15 GB/s per direction (i.e., 30 GB/s bidirectional).

Chapter 6

Coupled LBM-MHFEM Computational Approach

As described in Chapter 5, the lattice Boltzmann method is an effective tool for numerical fluid flow simulations and its coupling with other methods is still subject of intensive research in order to develop solvers for complex multiphysics models [A52, A69, A76, A94, A131, A135, A136]. In this work, we investigate a novel computational approach based on the coupling of LBM with the *NumDwarf* scheme described in Chapter 4, which is based on the mixed-hybrid finite element method. As the initial step towards the development of a flexible multiphysics solver, a rather simple model coupling the Navier–Stokes equations with a linear advection–diffusion equation is considered. The content of this chapter deals with numerical details of the coupled approach based on the paper [A111] and represents original work of the author. An application of the developed approach is described in the next chapter.

The chapter is organized as follows. Section 6.1 formulates the general problem and its special case with an analytical solution for convergence analysis. Then, Sections 6.2 to 6.4 provide details related to the coupled computational approach and its implementation. The final Section 6.5 describes the results of the experimental convergence analysis using the benchmark problem from the first section.

6.1 Problem formulation

The flow of an incompressible fluid is governed by the Navier–Stokes equations

$$\nabla \cdot \boldsymbol{v} = 0, \tag{6.1a}$$

$$\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \nabla \boldsymbol{v} = -\frac{1}{\rho} \nabla p + \nu \Delta \boldsymbol{v}, \tag{6.1b}$$

where $\boldsymbol{v} = [v_x, v_y, v_z]$ [m s^{-1}] is the fluid velocity, p [Pa] is the pressure, ρ [kg m^{-3}] is the fluid density, and ν [m^2 s^{-1}] is the kinematic viscosity. All these quantities are functions of spatial coordinates $\boldsymbol{x} = [x, y, z] \in \Omega_1 \subset \mathbb{R}^3$ and time $t \in (0, t_{\max})$.

The mass and momentum conservation laws (6.1) are coupled with a generic advection–diffusion equation without sources or sinks that may be written in the conservative form

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\phi \boldsymbol{v} - D_0 \nabla \phi) = 0, \tag{6.2a}$$

where ϕ is a variable that may be associated to the physical quantity transported by the fluid (e.g., molar or mass concentration for mass transport, or temperature for heat transport) and D_0 [m^2 s^{-1}] is the diffusion coefficient. Combining Equations (6.1a) and (6.2a) leads to the non-conservative form

$$\frac{\partial \phi}{\partial t} + \boldsymbol{v} \cdot \nabla \phi - \nabla \cdot (D_0 \nabla \phi) = 0. \tag{6.2b}$$

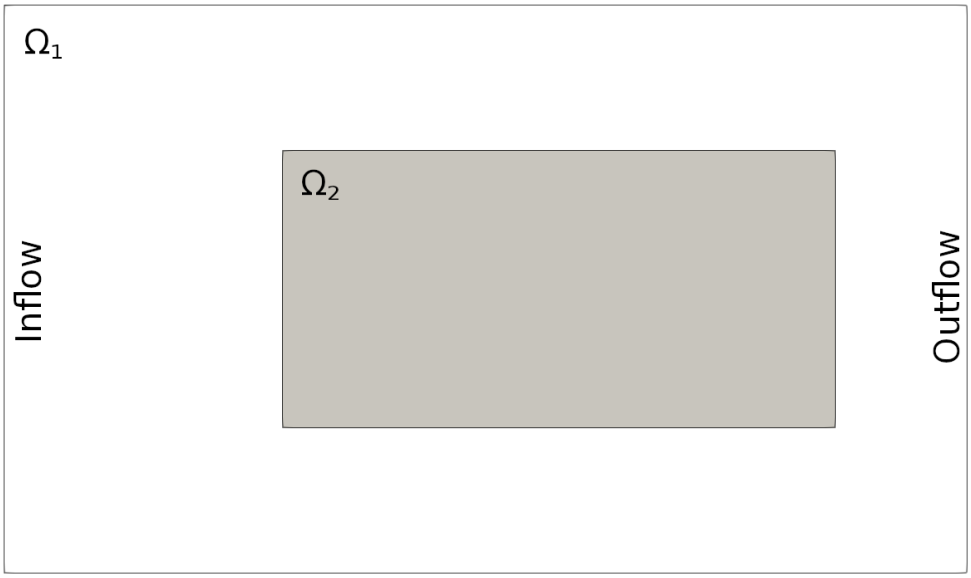


Figure 6.1: Schematic configuration of the computational domains Ω_1 and Ω_2 (2D cross-section of a 3D cuboidal channel along the plane $y = 0$).

Equations (6.2a) and (6.2b) are assumed to hold in $\Omega_2 \times (0, t_{\max})$, where $\Omega_2 \subset \Omega_1$. Since Equations (6.1) and (6.2) are coupled only via the velocity field $\boldsymbol{v}$, Equation (6.1) can be solved without Equation (6.2) as the values of ϕ do not influence the flow field.

In this work, we are interested in modeling constituent transport in a channel flow. As a benchmark problem, we consider the situation illustrated in Figure 6.1 where $\Omega_1 = [0, 1.75] \times [0, 1] \times [0, 1]$ (dimensions are in meters) is a cuboidal channel with inflow, outflow, and solid wall boundaries. The constituent transport is tracked in the subdomain $\Omega_2 = [0.5, 1.5] \times [0.25, 0.75] \times [0.25, 0.75]$ (in meters). For simplicity, we assume that Ω_2 is completely immersed in the domain Ω_1 (i.e., none of the domain boundaries coincide: $\partial\Omega_1 \cap \partial\Omega_2 = \emptyset$). The kinematic viscosity $\nu = 15.52 \times 10^{-6}\ \mathrm{m^2\ s^{-1}}$ and the diffusion coefficient $D_0 = 25.52 \times 10^{-6}\ \mathrm{m^2\ s^{-1}}$ are set the same as in Chapter 7 (see Table 7.2).

Both Equations (6.1) and (6.2) must be supplemented by suitable initial and boundary conditions. For simplicity, the velocity field is initialized by zero ($\boldsymbol{v}(\boldsymbol{x}, 0) = \boldsymbol{0}$ for $x \in \Omega_1$) and the variable ϕ is initialized by one ($\phi(\boldsymbol{x}, 0) = 1$ for $\boldsymbol{x} \in \Omega_2$). The boundary conditions on $\partial\Omega_1$ are posed as follows:

- Solid walls (i.e., the top, bottom, front, and back sides of Ω_1) are modeled using the standard no-slip boundary condition.
- A fixed value of pressure and a zero velocity gradient in the normal direction are prescribed on the outflow side (i.e., right hand side) of domain Ω_1.
- Fixed values of velocity are prescribed on the inflow side (i.e., left hand side) of Ω_1. Details will be described later.

The conditions for the variable ϕ on the subdomain boundary $\partial\Omega_2$ are:

- A Dirichlet-type condition is used to prescribe fixed values on the inflow side (i.e., left hand side) of Ω_2.
- A Neumann-type condition is used to prescribe zero gradient in the normal direction on all remaining sides of Ω_2.

In order to study the differences between the conservative and non-conservative formulations, the boundary conditions are specified with the following profiles. Turbulent flow in Ω_1 is induced by prescribing a fluctuating velocity profile on the inflow boundary of Ω_1 using the algorithm described in Appendix G with the following parameters: mean velocity $\overline{\boldsymbol{v}}_{\text{in}} = [1, 0, 0]^T\ \text{m s}^{-1}$, integral length scale $\mathcal{L}_{\text{int}} = 0.05\ \text{m}$, turbulent kinetic energy $k = 10^{-2}\ \text{m}^2\ \text{s}^{-2}$, and number of discrete modes $N_{\text{modes}} = 3000$. On the inflow boundary of Ω_2 ($x = 0.5\ \text{m}$), we prescribe a fixed value $\phi_{\text{in}} = 1$. Given a velocity field $\boldsymbol{v}(\boldsymbol{x}, t)$ satisfying the divergence-free condition Equation (6.1a), this initial–boundary–value problem has a trivial analytical solution $\phi(\boldsymbol{x}, t) = 1$ for all $\boldsymbol{x} \in \Omega_2$ and $t > 0$.

6.2 Computational algorithm and time adaptivity

The Navier–Stokes equations (6.1) are solved numerically by the lattice Boltzmann method described in Chapter 5. The transport equation (6.2) is incorporated into the mathematical framework of the *NumDwarf* solver described in Chapter 4. Both Equation (6.2a) and Equation (6.2b) can be converted to the form of Equation (4.1) on Page 57 by taking $n = 1$, $\boldsymbol{Z} = [\phi]$, $\mathbf{N} = [1]$, $\boldsymbol{m} = [1]$, $\boldsymbol{D} = [D_0]$, $\boldsymbol{w} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$, $\boldsymbol{f} = [0]$, and depending on the equation:

- $\boldsymbol{u} = \mathbf{0}$, $\boldsymbol{a} = [\boldsymbol{v}]$ for Equation (6.2a) (conservative form),
- $\boldsymbol{u} = [\boldsymbol{v}]$, $\boldsymbol{a} = \mathbf{0}$ for Equation (6.2b) (non-conservative form).

The resulting solver couples the computational algorithms of MHFEM (Section 4.2.10) and LBM (Section 5.4). The initialization of the solver includes setting all physical parameters, discretization of the domain Ω_1 by the regular lattice $\mathcal{L}_{\overline{\Omega}_1}$ (see Section 5.2.1), discretization of the domain Ω_2 by the conforming unstructured mesh $\mathcal{K}_h$ (see Section 4.2), and decomposition of the lattice and mesh into subdomains for distributed computing (described later in Section 6.4). Then the solver allocates all data structures, applies the initial conditions and starts the main time-loop. To optimize the efficiency of the solver, we developed an adaptive time-stepping algorithm based on a "CFL-like" condition. The time-stepping part of the computational algorithm is summarized in Algorithm 9.

Algorithm 9 (Time-stepping in the coupled LBM-MHFEM scheme)

1. Set $t_L := 0\ \text{s}$ and $t_M := 0\ \text{s}$, physical time step δ_t [s] and final time $t_{\max}$ [s].
2. While $t_L < t_{\max}$:

2.1. After every 1000 iterations, recompute inflow velocity fluctuations that will be used in the next 1000 iterations. See Appendix G for details.

2.2. Perform one iteration of LBM on the lattice $\mathcal{L}_{\overline{\Omega}_1}$ (i.e., perform the steps 3.1 to 3.7 from Algorithm 3 on Page 86).

2.3. Set $t_L := t_L + \delta_t$.

2.4. If $t_M < t_L$:

2.4.1. Interpolate the velocity field from the regular lattice to the unstructured mesh. See Section 6.3 for details.

2.4.2. Compute $C = \max_E\{|\hat{\boldsymbol{v}}_E|\delta_t/h_E\}$, where $E \in \mathcal{E}_h$ goes over all faces of the mesh, $|\hat{\boldsymbol{v}}_E|$ is the interpolated velocity magnitude on face E and h_E [m] is the mesh size on face E.

2.4.3. Set the time step for MHFEM: $\delta_{t,M} := \delta_t \lfloor C_{\max}/C \rfloor$ if $C \le C_{\max}$, else $\delta_{t,M} := \delta_t / \lceil C/C_{\max} \rceil$.

2.4.4. Set the number of MHFEM iterations: $n_M := 1$ if $C \le C_{\max}$, else $n_M := \lceil C/C_{\max} \rceil$.

2.4.5. Perform n_M iterations of MHFEM using Algorithm 2 with the time step $\delta_{t,M}$.

2.4.6. Set $t_M := t_M + n_M \delta_{t,M}$.

2.5. If t_L or t_M reached a pre-defined time period, make a snapshot of the current data for visualization/post-processing.

Initially, two separate time tracking variables t_L and t_M are created, one for computations on the lattice and the other for computations on the mesh. The lattice variable t_L is the main one which is checked in the time loop condition (step 2 above). The variable t_M is incremented separately in the step 2.4 that is responsible for the coupling between computations on the lattice and the mesh. Here the velocity field is interpolated from the lattice to the mesh (the procedure is described later in Section 6.3) and the time-step control factor C is computed. The time step $\delta_{t,M}$ for the mesh computations can be either longer or shorter than δ_t depending on the value of C. If $C \leq C_{\max}$, where $C_{\max}$ is a pre-defined constant, the time step $\delta_{t,M}$ is set as an integral multiple of δ_t and the time variable t_M is pushed forward to the new time level by just one iteration of MHFEM. Hence, several following LBM iterations can be performed successively until the lattice time t_L overruns t_M and the condition 2.4 is satisfied. On the other hand, if $C > C_{\max}$, the MHFEM solver needs to perform $n_M > 1$ iterations to move to the new time level $t_M + \delta_t$ with a shorter time step $\delta_{t,M} = \delta_t / n_M$.

In practice, we found empirically that limiting the time step $\delta_{t,M}$ with $C_{\max} = 1/2$ is necessary to ensure the stability of the coupled solver. The overall performance of the solver depends on the concrete values of $\delta_{t,M}$ selected by the adaptive algorithm, which are influenced by the quantities needed to compute the time-step control factor C, i.e., local velocity magnitude, lattice time step δ_t (which is related to the lattice space step δ_x), and the local space step h_E of the unstructured mesh.

An important choice related to the solver performance is the selection of the algorithm for the solution of large systems of linear equations arising from the MHFEM discretization. We achieved the best performance using the BiCGstab method combined with the Jacobi preconditioner, both implemented in TNL [A142]. In all simulations presented in this thesis, the BiCGstab method took at most 4 iterations to converge in most of the time steps, so improved performance cannot be expected from stronger preconditioners.

6.3 Interpolation of the velocity field

Since Equations (6.1) and (6.2) are coupled by the velocity field $\boldsymbol{v}(\boldsymbol{x}, t)$, the numerical approach relies on the interpolation of the approximate velocity field computed by LBM and its projection into the finite element space used by MHFEM. Note that the spatial discretization of the domain $\Omega_2 \subset \Omega_1$ is generally different than the equidistant lattice on Ω_1; it may be a regular grid with different space steps or even an unstructured mesh.

The interpolation of the velocity field can be requested at any point $\boldsymbol{x} \in \Omega_2$. The surrounding lattice points $\hat{\boldsymbol{x}} \in \mathcal{L}_{\overline{\Omega}_1}$ can be easily found and the linear or cubic interpolation in $\mathbb{R}^3$ can be used to obtain the velocity at $\boldsymbol{x}$ from the velocities at $\hat{\boldsymbol{x}}$. Note that linear interpolation can be implemented more efficiently than cubic interpolation as it uses fewer input data points. Our implementation of the cubic interpolation is not efficient and may cause the whole solver to run multiple times slower compared to the linear interpolation. The impact of using the linear or cubic interpolation on the accuracy of the numerical solution is investigated in Section 6.5.

The finite element space used by MHFEM imposes requirements on the interpolation of the velocity field. In this work, we use the Raviart–Thomas–Nédélec space of the lowest order $\mathbf{RTN}_0(\mathcal{K}_h)$ for the finite element–approximation of the velocity field. According to the definition in Equation (4.7) on Page 59, functions $\boldsymbol{\omega}$ belonging to the space $\mathbf{RTN}_0(\mathcal{K}_h)$ must satisfy:

1. for any element $K \in \mathcal{K}_h$, the restriction of $\boldsymbol{\omega}$ to K, $\boldsymbol{\omega}|_K$, must belong to the finite element space $\mathbf{RTN}_0(K)$ on the element K,

2. the normal trace of $\boldsymbol{\omega}$ must be continuous on all interior faces of the mesh, i.e., $\int_E \boldsymbol{\omega}|_{K_1} \cdot \boldsymbol{n}_{K_1,E} + \int_E \boldsymbol{\omega}|_{K_2} \cdot \boldsymbol{n}_{K_2,E} = 0$ for all $E \in \mathcal{E}_h^{\text{int}}$, $E \in \mathcal{E}_{K_1} \cap \mathcal{E}_{K_2}$, where $\boldsymbol{n}_{K_\ell,E}$ is the unit normal vector on the face E oriented outward from the element K_ℓ, $\ell = 1, 2$.

An interpolation strategy compatible with these requirements is as follows. First, velocity is evaluated at the element face centers $\boldsymbol{x}_E$ for all $E \in \mathcal{E}_h$. This can be done at any time level t yielding the approximate velocity values $\hat{\boldsymbol{v}}_E \equiv \hat{\boldsymbol{v}}(\boldsymbol{x}_E, t)$ which are then used for the projection into the $\mathbf{RTN}_0(\mathcal{K}_h)$ space. The discrete velocity field is assumed to be piecewise constant on the element sides $\mathcal{E}_h$ and the values $\hat{\boldsymbol{v}}_E$ define the components corresponding to the face E in the finite element spaces $\mathbf{RTN}_0(K_j)$ of the elements K_j adjacent to the face E.

Finally, it is important to note that the MHFEM schemes for Equations (6.2a) and (6.2b) do not behave equivalently with a general discrete velocity field interpolated to the mesh. This is because the discrete velocity field computed by LBM may not satisfy Equation (6.1a) exactly and even if it did, the interpolation scheme combines values from different locations in the flow field on a single element. Hence, the field interpolated to the unstructured mesh may be locally non-conservative, i.e., the discrete approximation of the velocity divergence $\sum_{E \in \mathcal{E}_K} \hat{\boldsymbol{v}}_E \cdot \boldsymbol{n}_{K,E}$ on element $K \in \mathcal{K}_h$ may be non-zero. The accuracy of the numerical scheme applied to the conservative form of Equation (6.2a) and non-conservative form of Equation (6.2b) is investigated in Section 6.5 on a benchmark problem.

6.4 Domain decomposition for overlapped lattice and mesh

The combination of a lattice overlapped with an unstructured mesh requires special attention when the solver is run in a distributed fashion, e.g. utilizing multiple GPU accelerators. Both the lattice and the mesh have to be decomposed into subdomains and each assigned to an MPI rank. Sufficiently wide overlapping regions on the lattice subdomains have to be generated to ensure that each rank can interpolate the velocity field from its lattice subdomains to its mesh subdomains. Furthermore, since computations on the lattice and the mesh are never executed concurrently, it is desirable to balance the sizes of the subdomains in order to achieve good computational efficiency.

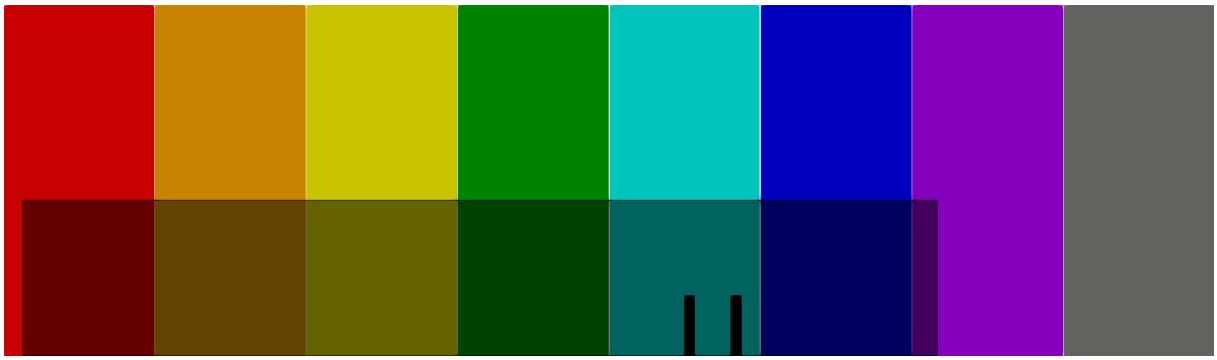
Figure 6.2 illustrates the problems with decomposition on an example involving a non-uniform cuboidal mesh that is refined around the two small black rectangles (they correspond to the two synthetic plants in the configuration EX-1 that will be described in Chapter 7). Due to a limitation of our LBM implementation, only 1D decompositions (i.e., such that all interfaces between two lattice subdomains are planes perpendicular to the x-axis) can be considered. Figure 6.2a shows a naive approach with uniformly sized lattice subdomains (highlighted with rainbow-colored rectangles), which leads to highly non-uniform distribution of mesh cell counts in each subdomain (indicated by percentages below the figure). In order to solve this balancing problem, we implemented a decomposition strategy that optimizes the lattice as well as mesh subdomains such that each MPI rank is assigned approximately the same number of lattice sites as well as mesh cells. The essential idea is to first determine the part of the domain where the lattice and mesh overlap, perform its decomposition such that an optimal mesh decomposition is achieved, and then decompose the remaining parts of the lattice (which do not overlap with the mesh) to add up to the optimal number of lattice sites assigned to each rank.

For a given regular lattice and an unstructured mesh covering the domain Ω_1 and its subdomain Ω_2, respectively, the decomposition procedure (with N_{ranks} denoting the number of MPI ranks used in the computation and N_{cells} denoting the total number of mesh cells) is summarized in Algorithm 10.

Algorithm 10 (Decomposition of lattice overlapped with unstructured mesh)

1. For all x-coordinates of the lattice sites, count the number of mesh cells whose centroid is located left of this x-coordinate. Use linear interpolation to obtain a continuous interpolant function $F(x)$ that is increasing from 0 to N_{cells}.
2. Find the smallest interval $[x_0, x_{N_{\text{ranks}}}]$ such that $F(x_{N_{\text{ranks}}}) - F(x_0) = N_{\text{cells}}$. This interval identifies the part of the lattice that is overlapped by the mesh, i.e., the dark transparent rectangle in Figure 6.2.

(a) Uniform lattice decomposition



12% 14% 14% 14% 24% 19% 3% 0%

(b) Non-uniform lattice decomposition

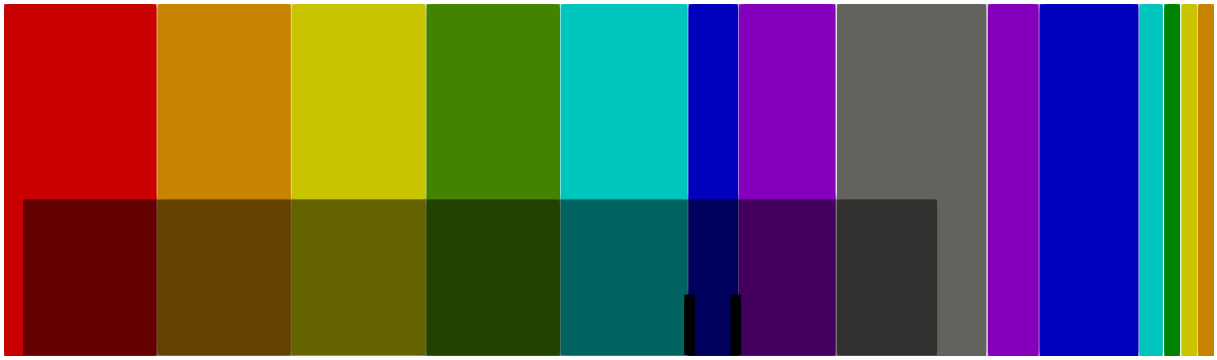


Figure 6.2: Domain decompositions of a regular lattice (rainbow-colored subdomains) overlapped with an unstructured mesh (dark transparent rectangle) that is refined around the two small black rectangles (they correspond to the synthetic plants from Chapter 7). The percentages below the case (a) indicate the portion of the total number of mesh cells included in the corresponding lattice subdomain. All lattice subdomains in the case (b) include 1/8 of the total number of mesh cells.

3. Find a partition $\{x_0, x_1, \ldots, x_{N_{\text{ranks}}-1}, x_{N_{\text{ranks}}}\}$ of the interval $[x_0, x_{N_{\text{ranks}}}]$ such that each subinterval contains approximately $N_{\text{cells}}/N_{\text{ranks}}$ mesh cells:

3.1. Define the objective function $f(x_1, \ldots, x_{N_{\text{ranks}}-1})$ which measures the imbalance of mesh cells included in each subinterval based on the function F.

3.2. Minimize the objective function using the gradient descent method and the uniform interval partition as initial condition.

3.3. Round the solution from $\mathbb{R}$ to the lattice coordinates (i.e., from `double` to `int`). As the rounding does not ensure the optimal result in integer precision, we additionally minimize the objective function in integer precision. We try to iteratively increment/decrement each component of the solution as long as it improves the partition.

4. Decompose the remaining parts of the lattice which do not overlap with the mesh. Note that these parts of the lattice are decomposed separately in reversed order (i.e., from right to left) in order to allow merging the non-overlapping subdomains with the adjacent mesh-overlapping subdomains (see the red and gray subdomains in Figure 6.2b).

The result of this decomposition procedure is illustrated in Figure 6.2b. Overall, the decomposition algorithm optimizes the computational cost and memory requirements of each MPI rank at the cost of increased communication due to increased number of lattice subdomains.

Table 6.1: Characteristics of lattice and grid resolutions used for the experimental convergence analysis.

	RES-A1	RES-A2	RES-A3
Lattice space step	8.06 mm	3.97 mm	1.97 mm
Lattice dimensions	$217 \times 128 \times 128$	$441 \times 256 \times 256$	$889 \times 512 \times 512$
MHFEM grid dimensions	$128 \times 64 \times 64$	$256 \times 128 \times 128$	$512 \times 256 \times 256$
No. of lattice sites	approx. $3.5 \cdot 10^6$	approx. $29 \cdot 10^6$	approx. $233 \cdot 10^6$
No. of grid cells	approx. $0.5 \cdot 10^6$	approx. $4 \cdot 10^6$	approx. $33 \cdot 10^6$
Base time step Δt	1.39×10^{-3} s	3.38×10^{-4} s	8.32×10^{-5} s
Average no. of LBM iters per MHFEM step ($\lfloor C_{\max}/C \rfloor$)	2	4	9

6.5 Experimental convergence analysis

In this section, we study the convergence of the coupled LBM-MHFEM scheme using a numerical experiment based on an artificial benchmark problem described in Section 6.1. The aim of this section is to study the differences between the conservative and non-conservative formulations of the advection–diffusion equation (6.2).

Several variants of the MHFEM scheme from Chapter 4 were used, namely explicit or implicit upwind (see Section 4.2.7), and linear or cubic interpolation of the velocity field (see Section 6.3). Each variant was computed in three resolutions denoted as RES-A1, RES-A2, and RES-A3, see Table 6.1. Note that single precision was used in the LBM part for fluid flow and double precision was used in the MHFEM part. To illustrate the turbulent flow field in Ω_1, Figure 6.3 shows the horizontal velocity (v_x) field in the final time $t_{\max} = 10$ s. Figure 6.4 shows qualitative differences between the fields of the transported variable ϕ that were computed using different variants of the MHFEM scheme. Since the fields obtained using any variant with the non-conservative formulation were visually indistinguishable from the constant analytical solution on the scale used in Figure 6.4, only the conservative formulation variants are shown in the figure. Note that for given resolution, the velocity field is the same in all variants of the MHFEM scheme. Quantitative comparison is presented in Table 6.2 in terms of L^p norms of the differences between the analytical solution $\phi = 1$ and each numerical solution ϕ_h.

Both qualitative and quantitative results in Figure 6.4 and Table 6.2 indicate that for the conservative formulation, changing linear interpolation to cubic, as well as changing the explicit upwind discretization to implicit upwind, leads to smoother and more accurate results. Furthermore, all these variants converge to the analytical solution as the lattice and grid are refined. However, even the most accurate numerical solution obtained using the conservative formulation exhibits an error that is larger by orders of magnitude compared to the non-conservative formulation, even in the coarsest resolution. The only difference between the discretizations of the non-conservative and conservative formulations is in Equation (4.23) on Page 62 where the former contains the term $\sum_{E \in \mathcal{E}_K} u_{i,j,K,E}$ corresponding to the discrete divergence of velocity. The results indicate that this extra term can be understood as a compensation for the non-zero divergence of the discrete velocity field interpolated on the mesh. Furthermore, it can be noticed in Table 6.2 that changing the interpolation or upwind scheme does not have a significant effect on the error when the non-conservative formulation is used. In the finest resolution RES-A3, using the linear interpolation and explicit upwind is not only advantageous for the performance of the solver, but also leads to a smaller error.

The presented results show that solving the transport equation in the conservative form of Equation (6.2a) with a highly turbulent velocity field may lead to large deviations in the numerical solution, whereas solving the non-conservative form of Equation (6.2b) with the same velocity field results in significantly more accurate solution. An alternative approach to address the problem of non-zero

Table 6.2: Results of the experimental convergence analysis for different formulations and variants of the MHFEM scheme.

			conservative		non-conservative	
interp.	upwind	resolution	$\|\phi - \phi_h\|_1$	$\|\phi - \phi_h\|_2$	$\|\phi - \phi_h\|_1$	$\|\phi - \phi_h\|_2$
linear	explicit	RES-A1	4.01×10^{-3}	1.11×10^{-2}	1.21×10^{-14}	3.70×10^{-13}
		RES-A2	2.01×10^{-3}	5.71×10^{-3}	5.05×10^{-15}	2.85×10^{-13}
		RES-A3	7.82×10^{-4}	2.28×10^{-3}	8.00×10^{-15}	1.34×10^{-12}
	implicit	RES-A1	3.24×10^{-3}	8.95×10^{-3}	3.62×10^{-13}	2.40×10^{-12}
		RES-A2	1.62×10^{-3}	4.64×10^{-3}	5.80×10^{-14}	2.62×10^{-13}
		RES-A3	6.23×10^{-4}	1.82×10^{-3}	3.97×10^{-14}	2.19×10^{-13}
cubic	explicit	RES-A1	3.25×10^{-3}	8.75×10^{-3}	1.19×10^{-14}	3.56×10^{-13}
		RES-A2	1.31×10^{-3}	3.63×10^{-3}	7.44×10^{-15}	5.01×10^{-13}
		RES-A3	3.98×10^{-4}	1.09×10^{-3}	8.68×10^{-15}	1.45×10^{-12}
	implicit	RES-A1	2.63×10^{-3}	7.08×10^{-3}	5.28×10^{-13}	2.46×10^{-12}
		RES-A2	1.07×10^{-3}	2.96×10^{-3}	5.73×10^{-14}	2.50×10^{-13}
		RES-A3	3.24×10^{-4}	8.83×10^{-4}	3.37×10^{-14}	1.91×10^{-13}

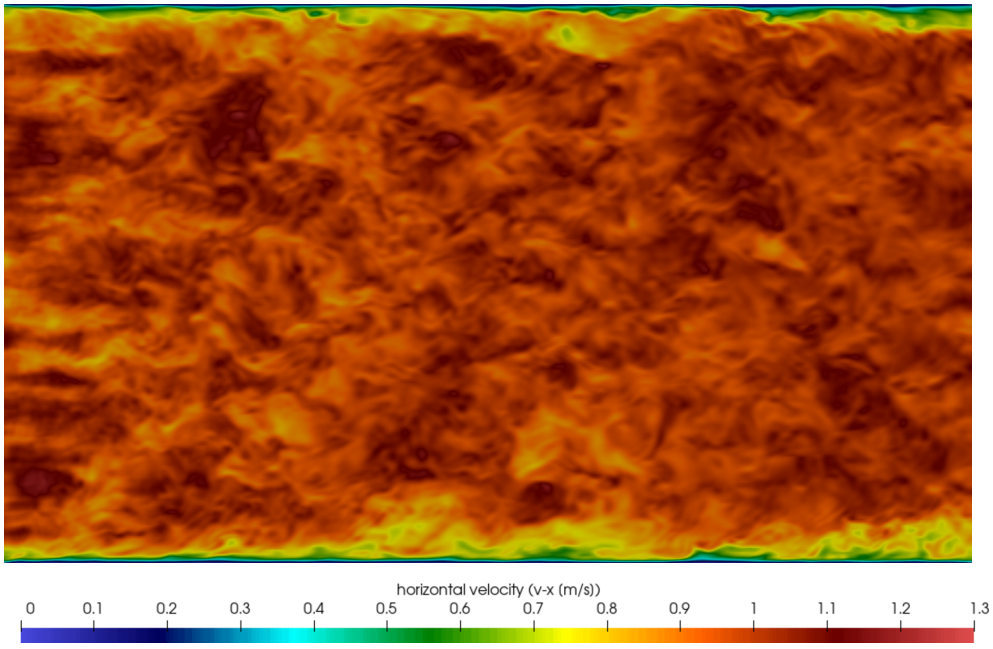
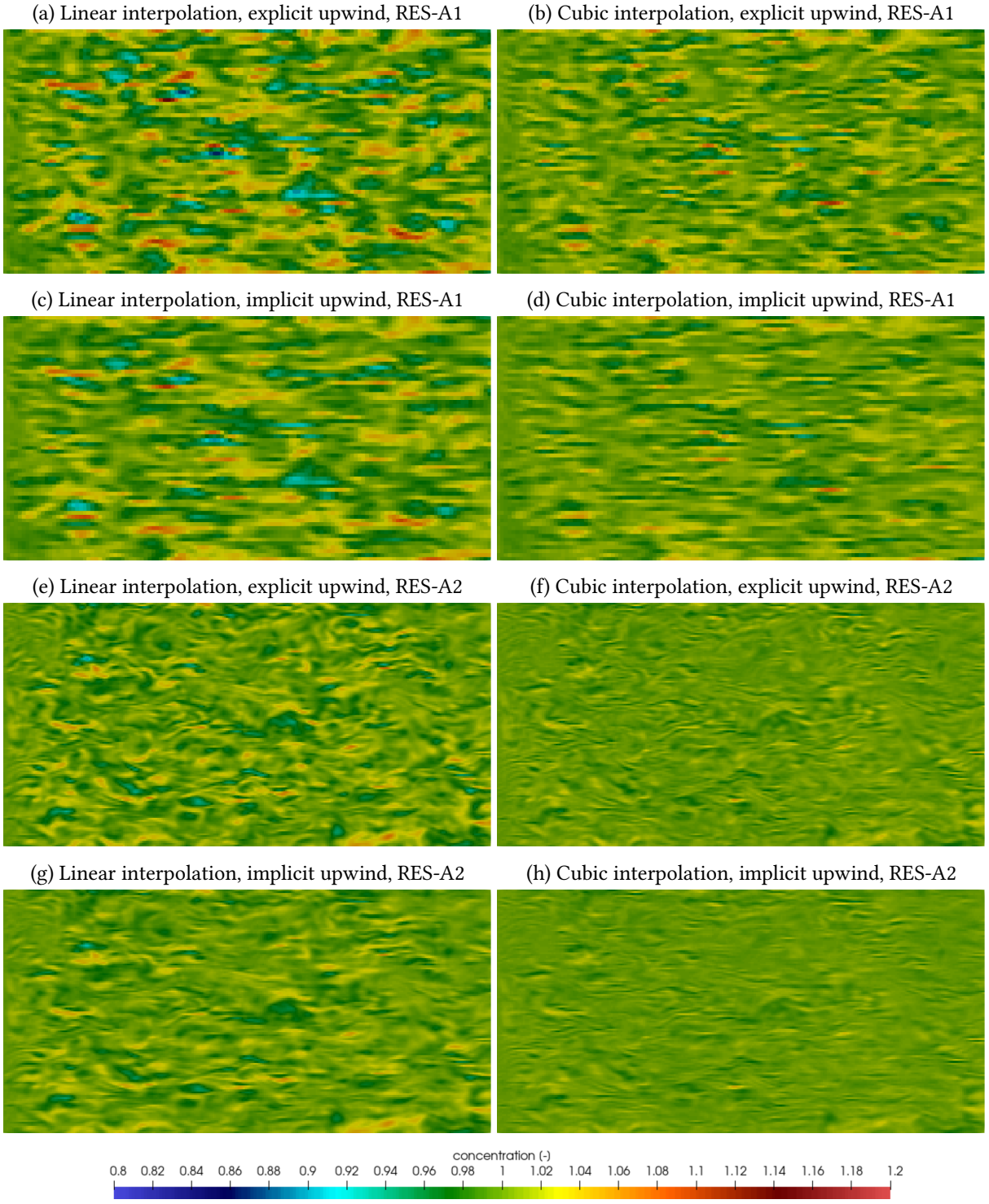


Figure 6.3: Horizontal velocity field (v_x) along the plane $y = 0$ in Ω_1, computed in resolution RES-A2.

divergence of the discrete velocity field might be to use a post-processing algorithm to recover the discrete divergence-free condition on the given mesh. The problem of compatibility between flow schemes producing a discrete velocity field and transport schemes using the interpolated velocity was extensively researched and several velocity post-processing algorithms were developed [A108, A120, A164]. However, such post-processing would incur additional cost to the computational algorithm and the approach is not investigated further in this thesis.



(a) Linear interpolation, explicit upwind, RES-A1

(b) Cubic interpolation, explicit upwind, RES-A1

(c) Linear interpolation, implicit upwind, RES-A1

(d) Cubic interpolation, implicit upwind, RES-A1

(e) Linear interpolation, explicit upwind, RES-A2

(f) Cubic interpolation, explicit upwind, RES-A2

(g) Linear interpolation, implicit upwind, RES-A2

(h) Cubic interpolation, implicit upwind, RES-A2

Figure 6.4: Simulated fields of the transported variable ϕ along the plane $y = 0$ in Ω_2 in the benchmark problem using the *conservative formulation* of the transport equation (6.2a). Several configurations of the numerical scheme are compared: linear and cubic interpolation of the velocity from LBM to MHFEM, and discretization of the advection term in the MHFEM scheme based on explicit and implicit upwind. Only the first two resolutions RES-A1 and RES-A2 are shown here.

Chapter 7

Mathematical Modeling of Vapor Transport in Turbulent Air Flow

In this chapter, we use the coupled LBM-MHFEM computational approach developed in Chapter 6 to simulate vapor transport in turbulent air flow above a soil surface. The content is based on the paper [A111]. The author developed the mathematical model and computational methodology, performed all simulations and compared the results with experimental data. The experimental methodology was developed by Andrew C. Trautz and Tissa H. Illangasekare, the analysis of the results and overall integration of mathematical modeling and experimental work were performed collectively by all co-authors of the paper.

The chapter is organized as follows. First, the motivation and introduction to the mathematical modeling of vapor transport in air is described in Section 7.1. The following Section 7.2 provides a general description of the experiments, mathematical model and boundary conditions. Then, Section 7.3 gives specific details of the computational methodology using the coupled LBM-MHFEM solver and the final Section 7.4 presents the validation results of our model. The model is compared both qualitatively and quantitatively to experimental data measured in three configurations resulting in different flow regimes.

7.1 Introduction

Numerous proprietary or open-source computational tools are available for solving partial differential equations originating from mathematical modeling of various biological, environmental, or industrial problems. In particular, computational software such as deal.II [A17], DUNE [C10], OpenFOAM [C26], TOUGH2 [M24], MFiX [M27], ANSYS Fluent [M3] or COMSOL Multiphysics [M8] are suitable for complex multiphysics simulations involving multiphase or compositional flows. However, each software has limitations: the underlying numerical methods may restrict the applicability of the software; the approach for code execution may cause limited or no advantage of using high-performance architectures, in particular graphical processing units (GPUs), for the acceleration of computations; and the software design in general might make it difficult to combine different tools for solving coupled problems. Furthermore, extending large software packages such as the aforementioned ones with novel mathematical approaches and numerical methods is challenging and unfeasible for most external users.

We have developed a novel computational approach based on a combination of the lattice Boltzmann method and the mixed-hybrid finite element method for simulating component transport within single-phase free flow. The lattice Boltzmann method (LBM) [B21] is a modern and efficient numerical method capable of simulating numerous problems in computational fluid dynamics (CFD), including turbulent fluid flows. The mixed-hybrid finite element method (MHFEM) [B6] is a well established general numerical method for solving partial differential equations. In this work, we use the MHFEM

formulation presented in [A72] and extend it with the coupling to LBM. There are multiple reasons why we pursue this approach:

- Both LBM and MHFEM individually have advantages compared to traditional numerical methods. LBM is based on the mesoscopic description of the fluid and the computational algorithm avoids the solution of Poisson equation for pressure [B21], which is the most time consuming part of algorithms based on finite difference methods (FDM) or finite volume methods (FVM). On the other hand, MHFEM is a general method that combines different finite element approximations of scalar as well as vector functions. For some problems, it may provide higher accuracy and robustness compared to standard FEM or FVM approaches [A149, A187].

- Both LBM and MHFEM individually can be efficiently parallelized and implemented for modern high-performance architectures, including GPU accelerators. All computations in the coupled LBM-MHFEM solver can be executed entirely on a GPU accelerator in order to utilize its computational power and avoid the hardware limitations caused by slow communication between the GPU and CPU over the PCIe bus.

- The coupled solver can utilize vast computational resources available on typical supercomputers by decomposing the domain and dividing the computation between multiple workers (GPUs) which communicate over the Message Passing Interface (MPI) [M15].

Verification of the developed approach against exact solutions is not possible, since there are no known exact analytical solutions for a coupled system of Navier–Stokes and transport equations. We therefore turn to controlled experimental data for model validation purposes, specifically posing this effort in terms of the problem of bare-soil evaporation, a key component of Earth's hydrologic, carbon, and energy balances [B8]. In this context, airflow has been shown to be a strong forcing parameter responsible for driving the process evaporation from bare soils or any porous medium [A102, A166]. Given the complexity of near-surface airflow, evaporation estimates in practice rely on simplified flow parameterizations or sparse wind measurements. We pose that the development of the combined LBM-MHFEM flow and transport model herein could support these issues by providing a way to simulate near-surface flow fields and subsequent above-ground vapor transport with a high level of fidelity, in turn improving evaporation estimates [A167].

The data used for validation in this thesis include velocity and relative humidity profiles measured in a low speed, climate controlled wind tunnel where 3D turbulent flow above a soil surface with cuboidal bluff bodies were investigated in [A168]. The wind tunnel used for these measurements was designed specifically for the investigation of coupled soil–plant–atmosphere processes as soil-test beds containing any desired soil(s) and vegetation can be interfaced at a sufficiently large scale along a 7.4 m long test-section.

This chapter focuses on the validation of the coupled numerical scheme developed in Chapter 6 using experimental data obtained in controlled environments and simplified flow geometry. While we present this investigation in terms of a bare-soil evaporation problem, it is important to note that with relatively minor modifications this numerical solver could be expanded to support a wide range of applications currently being investigated by the authors, including: heat transfer in the context of disturbed environments, training AI/ML super-resolution algorithms, dust forecasting and mitigation efforts, and greenhouse gas loading. More importantly, the work presented herein should be viewed as the initial steps toward the development of a flexible numerical solver that can be used to support fundamental process understanding and exploration, guide sensitivity analysis for measurements in regions with high uncertainty, inform sampling in future experimental effort or help fill gaps in datasets where it was not possible to make measurements.

7.2 Problem formulation

In this section, the background for numerical simulations is introduced, namely the experimental facility and procedures are described, and the mathematical model is formulated.

7.2.1 Experimental setup and methodology

All experimental data used in this thesis and the paper [A111] were generated in the Center for Experimental Study of Subsurface Environmental Processes (CESEP) wind tunnel–porous media test facility now located at the US Army Engineer Research and Development Center (ERDC) Synthetic Environment for Near-Surface Sensing and Experimentation (SENSE) Research Facility. The facility is centered around a closed-circuit, climate-controlled, low-speed wind tunnel that can be interfaced with soil test-beds of varying size. The test facility was designed specifically for the investigation of coupled soil–plant–atmosphere processes, including air flow and heat and mass transport at a 1:1 scale; the wind tunnel meets similarity criteria and is therefore suitable for momentum scaling studies as well. A brief description of the components relevant to this work is presented below for the convenience of the reader. Details concerning the test facility can be found in [A166, A169].

The wind tunnel was interfaced with a 7.15 m long and 0.3 m wide soil test-bed along the centerline of its 7.4 m long, 1 m wide, and 1 m tall test-section. The experimental datasets of [A168] were chosen for model validation due to their use of synthetic plants (i.e., porous limestone blocks) instead of living vegetation. This approach significantly simplified the airflow component of the problem (i.e., the plant can be treated as a traditional bluff body) while still retaining key hydrodynamic characteristics. The dimensions of each block were nominally $(3.15 \times 3.15 \times 29.5) \pm 0.1$ cm and they were planted in a vertical position (10.0 ± 0.3) cm deep, leaving the 19.5 cm high part above ground.

A total of three experimental configurations featuring two synthetic plants were explored by [A168]. Each configuration was characterized in terms of the spacing between the two synthetic plants: 15 cm spacing (EX-1), 45 cm spacing (EX-2), and 105 cm spacing (EX-3). The positions of the synthetic plants in these configurations are given by:

- EX-1 (15 cm spacing): $x_{\mathrm{I}} = -5.197$ m, $x_{\mathrm{II}} = -5.047$ m,
- EX-2 (45 cm spacing): $x_{\mathrm{I}} = -5.647$ m, $x_{\mathrm{II}} = -5.197$ m,
- EX-3 (105 cm spacing): $x_{\mathrm{I}} = -6.247$ m, $x_{\mathrm{II}} = -5.197$ m.

Note that in the coordinate system used hereafter, $x = 0$ m corresponds to the upstream entrance of the test-section of the wind tunnel (right hand side), $x = -7.4$ m corresponds to the downstream exit of the test section (left hand side), $z = 0$ m corresponds to the ground surface and $y = 0$ m corresponds to the centerline of the test-section. At the time that the datasets associated with [A168] were generated, the authors did not have access to a LiDAR system that could scan the exact position of the bluff bodies. Uncertainty of the placement of the bluff bodies is thus given as $\delta x_{\mathrm{I}} = \delta x_{\mathrm{II}} = \delta y = \pm 5$ mm. The maximum deviatory angle of attack of the bluff bodies relative to the direction of the flow is similarly given as 5°.

During each experiment, the mean wind speed, air temperature, and relative humidity of the air entering the test-section was controlled and continuously monitored. Given the coupling of the test-section with the partially saturated soil test-bed, a variable temperature and vapor gradient was observed above the soil tank. The soil temperature was controlled by varying the exterior temperature of the soil test-bed and the soil moisture was hydrostatically distributed and allowed to freely evaporate. Table 7.1 provides a summary of the average soil, surface and air temperatures in the experiments. Variability around the mean value was caused by the cycling of the individual climate control systems (i.e., heater, chiller, humidifier, dehumidifier) throughout the duration of the experiments. The exterior

Table 7.1: Experimental climate conditions.

		EX-1	EX-2	EX-3
Mean Air Temperature	[℃]	25.83 ± 1.52	25.70 ± 1.06	23.98 ± 0.21
Mean Surface Temperature	[℃]	22.10 ± 0.23	23.50 ± 0.80	22.07 ± 0.03
Mean Soil Temperature	[℃]	20.72 ± 0.31	22.40 ± 0.18	20.92 ± 0.11

temperature of the soil test-bed and barometric pressure were also measured through out the experiments; the reader is referred to [A168, A169] for more details.

The measurements of [A168, A169] are available as a public dataset [O36]. Note that only a subset of these measurements is relevant for the purpose of this chapter, namely the airflow properties (velocity, RMS, Reynolds stress) and relative humidity above the soil surface. Given the size of the domain and the experimental setup, the number of airflow and relative humidity measurements were necessarily constrained. The locations of airflow and relative humidity were varied between the three configurations based on the spacing distance between the synthetic plants and the resulting flow regime created. The measurement locations are highlighted in the figures in Section 7.4. The laser used to make the flow measurements was mounted on an automated traverse located outside the wind tunnel test-section. Uncertainty in the exact location where the measurements were made can be given as $\delta x = \delta y = \pm 5\,\mathrm{mm}$ and $\delta z = \pm 1\,\mathrm{mm}$. The sensor used to measure relative humidity was similarly mounted on an automated traverse located within the test-section; uncertainty associated with this system is given as $\delta x = \pm 10\,\mathrm{mm}$ and $\delta y = \delta z = \pm 5\,\mathrm{mm}$.

Air flow statistics (i.e., velocity, turbulence intensity, Reynolds stress) above the soil surface were measured using two-dimensional laser Doppler velocimetry [A169], providing high frequency data with an accuracy of 5 %. Relative humidity was measured with an accuracy of ±0.03 using a relative humidity–temperature (RHT) sensor constructed by the University Corporation of Atmospheric Research. Compared with the laser, these RHT sensors have a significantly lower sampling rate (≈ 1 Hz); data collected over a 30 second window were averaged at each measurement location [A169].

Furthermore, water loss from the wetted surface of the synthetic plants was measured in a separate small scale experiment [A169]. These data were used to calculate an average mass flux of $\Phi_\star = 0.128\,\mathrm{g\,cm^{-2}\,d^{-1}}$; this value was assumed to be applicable to all three configurations given similarity in applied climate conditions.

7.2.2 Mathematical model

The air flow in the free space above the soil surface is governed by the Navier–Stokes equations. As the model targets low Mach number situations (Ma ≈ 0.003 in the wind tunnel), the fluid is considered to be incompressible [A60, A127]. For convenience, we copy Equation (6.1) here from Chapter 6. The momentum and mass conservation laws for the air are therefore represented by

$$\nabla \cdot \boldsymbol{v} = 0, \tag{7.1a}$$

$$\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \nabla \boldsymbol{v} = -\frac{1}{\rho}\nabla p + \nu \Delta \boldsymbol{v}, \tag{7.1b}$$

where $\boldsymbol{v} = [v_x, v_y, v_z]$ $[\mathrm{m\,s^{-1}}]$ is the air velocity, p [Pa] is the pressure, ρ $[\mathrm{kg\,m^{-3}}]$ is the air density, and ν $[\mathrm{m^2\,s^{-1}}]$ is the kinematic viscosity of the air. In general, these quantities are given as functions of spatial coordinates $\boldsymbol{x} = [x, y, z] \in \Omega_1 \subset \mathbb{R}^3$ and time $t \in (0, t_{\max})$.

The mass conservation law for a component α within a gas mixture in $\Omega_2 \times (0, t_{\max})$ can be derived [B4] and written as

$$\frac{\partial \rho_\alpha}{\partial t} + \nabla \cdot (\rho_\alpha \boldsymbol{v} + \boldsymbol{J}_\alpha) = 0, \tag{7.2}$$

Table 7.2: Model parameters for air under standard atmospheric conditions (25 °C and pressure of 1 bar).

Density ρ [O7]	$1.184\,\mathrm{kg\,m^{-3}}$
Kinematic viscosity ν [O7]	$15.52 \times 10^{-6}\,\mathrm{m^2\,s^{-1}}$
Molecular diffusivity D_{H_2O} of water vapor in air [A132]	$25.52 \times 10^{-6}\,\mathrm{m^2\,s^{-1}}$

where ρ_α [$\mathrm{kg\,m^{-3}}$] is the density of the component α and $\boldsymbol{J}_\alpha$ [$\mathrm{kg\,m^{-2}\,s^{-1}}$] is the diffusive flux. As in Equation (7.1a), no sources/sinks are considered in Equation (7.2). The diffusive flux is given by the Fick's law [B4] as

$$\boldsymbol{J}_\alpha = -\rho D_\alpha \nabla \omega_\alpha, \tag{7.3}$$

where D_α [$\mathrm{m^2\,s^{-1}}$] denotes the molecular diffusivity coefficient of the component α in the mixture and $\omega_\alpha = \rho_\alpha / \rho$ [-] is the mass fraction of the component α in the mixture.

In this work, we consider a single component $\alpha \equiv \mathrm{H_2O}$ representing water vapor dispersed in air. The component density ρ_α corresponds to the absolute humidity (i.e., the mass of the water vapor per unit volume) and the mass fraction ω_α corresponds to the specific humidity. The relative humidity ϕ [-] is defined as

$$\phi = \frac{p_\alpha}{p_\alpha^*}, \tag{7.4}$$

where p_α [Pa] is the partial pressure of water vapor in the mixture and p_α^* [Pa] is the equilibrium vapor pressure of water over a flat surface of pure water at a given temperature. Assuming that the mixture behaves as an ideal gas at constant temperature T [K], the ideal gas law $p_\alpha = \rho_\alpha R_\alpha T$ with the specific gas constant R_α [$\mathrm{J\,kg^{-1}\,K^{-1}}$] means that the partial pressure p_α is proportional to the absolute humidity ρ_α and thus the relative humidity equals

$$\phi = \frac{\rho_\alpha}{\rho_\alpha^*}, \tag{7.5}$$

where ρ_α^* [$\mathrm{kg\,m^{-3}}$] is the saturated absolute humidity corresponding to p_α^*. Using $\rho_\alpha = \phi \rho_\alpha^*$, Equation (7.3) and the assumption of constant density ρ, Equation (7.2) transforms to the conservative transport equation

$$\frac{\partial \phi}{\partial t} + \nabla \cdot \left(\phi \boldsymbol{v} - D_{H_2O} \nabla \phi \right) = 0. \tag{7.6a}$$

Based on the experience from Chapter 6, we also combine Equations (7.1a) and (7.6a) to derive the non-conservative form

$$\frac{\partial \phi}{\partial t} + \boldsymbol{v} \cdot \nabla \phi - \nabla \cdot \left(D_{H_2O} \nabla \phi \right) = 0. \tag{7.6b}$$

Note that Equation (7.6) looks formally the same as Equation (6.2), but here the quantity ϕ has a physical interpretation.

Although a slightly variable temperature distribution above the soil tank was observed during the experiments [A168], we assume its impact on the density, kinematic viscosity, molecular diffusivity, and relative humidity to be negligible compared to the sensor accuracies. The isothermal model given by Equations (7.1) and (7.6) is used with constant parameters ρ, ν, and D_{H_2O}. Furthermore, the fluid density ρ is assumed to be independent of the relative humidity ϕ and the effect of gravity is neglected due to the dimensions of the experimental facility. Model parameters for air under standard atmospheric conditions are given in Table 7.2.

For the purpose of this thesis we are interested in simulating only the free flow region, where the soil-atmosphere and synthetic plant (bluff body)–atmosphere interfaces are treated using boundary conditions (described in Section 7.2.3). Hence, the computational domains Ω_1 and Ω_2 are considered as shown in Figure 7.1. Based on the experimental setup described in Section 7.2.1, the computational domain Ω_1 is defined as an inset of the whole test-section starting at a downstream distance of −3.507 m

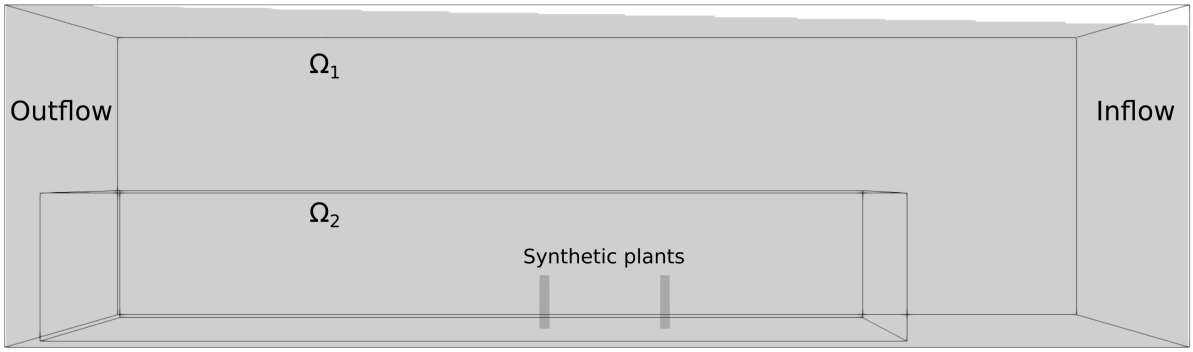


Figure 7.1: Illustration of the computational domains Ω_1 and Ω_2 for Equations (7.1) and (7.6). Note that Ω_1 conforms to the actual dimensions (width and height) along a 3.89 m length of the wind tunnel test-section where experimental measurements were made.

from the test-section inlet. The total dimensions of Ω_1 are approximately $3.89\,\mathrm{m} \times 1\,\mathrm{m} \times 1.13\,\mathrm{m}$. The upper side of the domain Ω_1 coincides with the inclined ceiling of the wind tunnel; the back and front sides coincide with the test-section walls. The bottom boundaries of both domains Ω_1 and Ω_2 coincide with the soil surface in which the two synthetic plants were planted as illustrated in Figure 7.1. The dimensions of the subdomain Ω_2 are $2.94\,\mathrm{m} \times 0.7\,\mathrm{m} \times 0.5\,\mathrm{m}$.

Note that Equation (7.1) is solved in domain Ω_1 and Equation (7.6) is solved in domain Ω_2. Since Equations (7.1) and (7.6) are coupled only via the velocity field $\boldsymbol{v}$, Equation (7.1) can be solved without Equation (7.6) and the latter can be solved only in a subdomain of Ω_1, i.e., $\Omega_2 \subset \Omega_1$.

Both Equations (7.1) and (7.6) must be supplemented by initial and boundary conditions suitable for this problem. In all simulations presented in this chapter, the velocity field was initialized by zero ($\boldsymbol{v}(\boldsymbol{x}, 0) = \boldsymbol{0}$) and the initial relative humidity profile varied with height as $\phi(\boldsymbol{x}, 0) = \phi_{\mathrm{in}}(z)$, where $\phi_{\mathrm{in}}(z)$ determines the inflow boundary condition for relative humidity (see Subsection “Inflow: relative humidity profile” on Page 114). Boundary conditions for the simulations are discussed in the following section.

7.2.3 Boundary conditions

Various boundary conditions are used throughout the simulations. The sides, ceiling, and floor (including soil) of the test-section are impermeable and thus modeled using the standard no-slip boundary condition. A simple outflow condition based on a fixed pressure value and a zero velocity gradient in the normal direction is used on the left hand side of the domain in Figure 7.1. On the right hand side of the domain Ω_1 in Figure 7.1, the inflow boundary condition for velocity $\boldsymbol{v}$ is prescribed. Two types of inflow boundary conditions are considered in this work with details provided in the following subsections.

The domain Ω_2 in Figure 7.1 has an inflow boundary on the right hand side where a height-dependent profile $\phi_{\mathrm{in}}(z)$ for the relative humidity ϕ is prescribed; see Subsection “Inflow: relative humidity profile” below for details. Furthermore, the value $\phi_{\mathrm{in}}(0)$ is used also for the fixed-value boundary condition at the bottom side of the domain Ω_2 that coincides with the interface between the soil tank and free space. On all remaining sides of the domain Ω_2 (i.e., on the free-stream boundaries), a zero gradient of the relative humidity ϕ is prescribed.

The final Subsection “Synthetic plants” below addresses boundary conditions applied on the synthetic plants to model ”transpiration”.

Inflow: mean velocity profile

Using this boundary condition, a time-constant inflow velocity profile approximating the mean free flow velocity measured in the wind tunnel is prescribed. The inflow velocity is set to $\boldsymbol{v}_{\text{in}} = [-v_{\text{in},x}(z), 0, 0]$, where the component $v_{\text{in},x}$ is specified as a function of height z [m] based on the 1/7-th power law [B34]:

$$v_{\text{in},x}(z) = \begin{cases} v_{\max}\left(\dfrac{z}{z_\delta}\right)^{\frac{1}{7}} & \text{if } z \leq z_\delta, \\ v_{\max}\left(\dfrac{z_{\max} - z}{z_\delta}\right)^{\frac{1}{7}} & \text{if } z_{\max} - z \leq z_\delta, \\ v_{\max} & \text{otherwise,} \end{cases} \tag{7.7}$$

where $z_{\max} = 1.0624\,\text{m}$ is the height of the computational domain at the inflow boundary, $z_\delta = 0.1\,\text{m}$ is the estimated boundary layer height and $v_{\max} = 0.8\,\text{m}\,\text{s}^{-1}$ corresponds to the mean free stream velocity in the wind tunnel. The parameters were chosen based on the experimentally measured velocity profiles and the same values are used in all three spacing configurations.

Inflow: velocity fluctuations

Direct numerical simulations (DNS) of turbulent flow in a regular domain require sufficiently high resolution and large domain size, otherwise the turbulent boundary layer may not fully develop and the simulation may give wrong results. As will be seen in Section 7.4, prescribing a time-constant velocity profile at the inflow boundary may lead to non-physical results, because the simulated flow field may remain laminar until it reaches the first obstacle placed in the domain. An alternative is to turn away from the DNS approach and add synthetic fluctuations to the velocity profile prescribed on the inflow boundary, which should help induce turbulent flow and let the boundary layer develop faster.

The inflow velocity $\boldsymbol{v}_{\text{in}} = \boldsymbol{v}_{\text{in}}(\boldsymbol{x}, t)$ is decomposed as

$$\boldsymbol{v}_{\text{in}}(\boldsymbol{x}, t) = \overline{\boldsymbol{v}}_{\text{in}}(\boldsymbol{x}) + \boldsymbol{v}'_{\text{in}}(\boldsymbol{x}, t), \tag{7.8}$$

where $\overline{\boldsymbol{v}}_{\text{in}}(\boldsymbol{x})$ is the mean (time-averaged) value given in Subsection "Inflow: mean velocity profile" above, and $\boldsymbol{v}'_{\text{in}}(\boldsymbol{x}, t)$ is the velocity fluctuation. The field $\boldsymbol{v}'_{\text{in}}(\boldsymbol{x}, t)$ is generated using the algorithm described in Appendix G with the following parameters. The turbulent length scale is set to one half of the inflow boundary layer z_δ used in Equation (7.7), i.e., $\mathcal{L}_{\text{int}} = 0.05\,\text{m}$. The number of discrete modes is $N_{\text{modes}} = 3000$, the turbulent kinetic energy is $k_{\text{in}} = 10^{-2}\,\text{m}^2\,\text{s}^{-2}$, and the kinematic viscosity ν is given in Table 7.2. Time correlation is introduced with the turbulent integral time scale $\mathcal{T}_{\text{int}} = \mathcal{L}_{\text{int}}/v_{\max}$, where $v_{\max}$ is the mean free stream velocity specified in Subsection "Inflow: mean velocity profile".

Note that the fluctuations generated with the aforementioned procedure are isotropic, which is an acknowledged simplification of anisotropic real-world turbulence. The procedure could be improved based on a specified anisotropic Reynolds stress tensor [O6], however, even using the inflow condition based on isotropic synthetic turbulence lead to improved results in this work. Experimentally, the problem was treated as 2D, i.e., the transverse flow statistics, such as the components $\overline{v'_y v'_y}$, $\overline{v'_x v'_y}$, and $\overline{v'_y v'_z}$ of the Reynolds stress tensor, were not quantified.

Inflow: relative humidity profile

In case of the subdomain Ω_2, the inflow boundary condition for relative humidity ϕ is specified as a function $\phi_{\text{in}}(z)$ of height z [m]:

$$\phi_{\text{in}}(z) = \begin{cases} \phi_{\max} + \phi_{\text{lin}} \dfrac{z_{\text{pow}} - z}{z_{\text{pow}}} & \text{if } z \le z_{\text{pow}}, \\ \phi_{\max} - (\phi_{\max} - \phi_{\min}) \left(\dfrac{z - z_{\text{pow}}}{z_{\text{const}} - z_{\text{pow}}} \right)^{\gamma} & \text{if } z > z_{\text{pow}} \text{ and } z \le z_{\text{const}}, \\ \phi_{\min} & \text{if } z > z_{\text{const}}, \end{cases} \tag{7.9}$$

where the common parameters $\phi_{\text{lin}} = 0.01$ and $z_{\text{pow}} = 0.005\,\text{m}$ describe the profile below the minimal height for measurements, $z_{\text{const}} = 0.195\,\text{m}$ corresponds to the height of the synthetic plants, and the remaining parameters $\phi_{\min}$, $\phi_{\max}$, γ are fitted values to the experimental data (see Table 7.3 and Figure 7.2).

Table 7.3: Values of fitted parameters $\phi_{\min}$, $\phi_{\max}$, γ for the relative humidity inflow boundary condition (7.9).

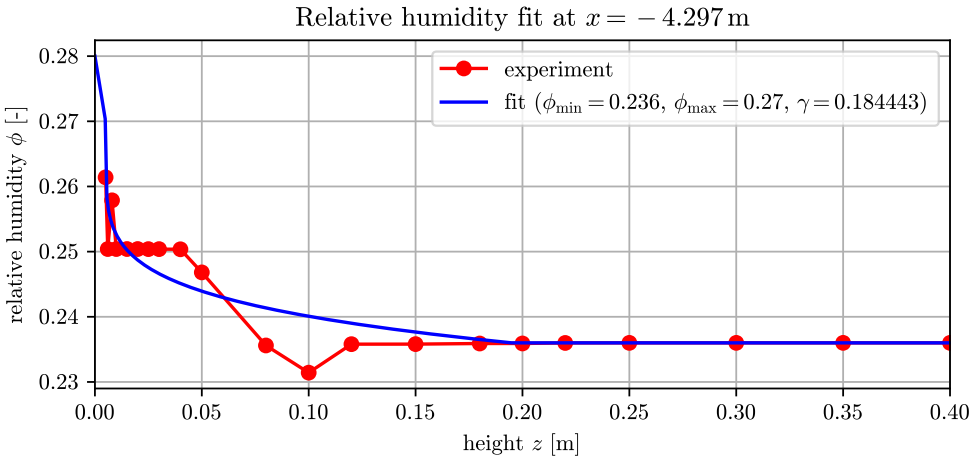
	EX-1	EX-2	EX-3
$\phi_{\min}$	0.236	0.230	0.217
$\phi_{\max}$	0.270	0.275	0.265
γ	0.184443	0.262402	0.402284

Synthetic plants

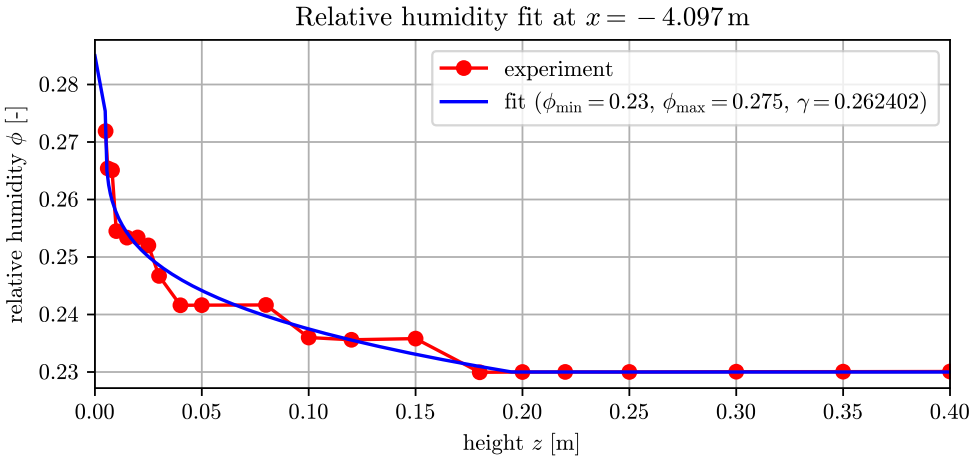
Water loss does not occur homogeneously on the surface of the synthetic plants since a capillary fringe of fully wetted surface was observed at the bottom of the plants. Therefore, water vapor is released with higher concentration near the soil surface and the top part of the plants is in equilibrium with the ambient environment. This is modeled in the simulations by prescribing a height-variable relative humidity profile on the sides of the synthetic plants as depicted in Figure 7.3. Based on the experiments, the capillary fringe height is approximately $z_{\min} = 3.5\,\text{cm}$. The prescribed relative humidity is $\phi_{\max} = 1$ below $z_{\min}$ and $\phi_{\min} = \phi_{\text{in}}(0.195)$ above $2z_{\min}$, where the ambient relative humidity $\phi_{\min}$ is set to the value prescribed on the inflow in the height of the synthetic plants. We also assume a transition part between $z_{\min}$ and $2z_{\min}$ where the surface is not fully wetted and the prescribed relative humidity decays linearly from $\phi_{\max}$ to $\phi_{\min}$.

To match the experimentally measured mass flux $\Phi_\star$ (see Section 7.2.1), we also need to prescribe a non-zero velocity in the normal direction on the sides of the plants. For example, given the mean relative humidity $\phi_{\text{mean}} = 0.437$ from Figure 7.3b and assuming that saturated air has absolute humidity of $23\,\text{g}\,\text{m}^{-3}$ at $25\,^{\circ}\text{C}$, the mass flux $\Phi_\star$ corresponds to the velocity $V_\star = 1.5\,\text{mm}\,\text{s}^{-1}$. The velocity $V_\star$ is prescribed only on the downstream side of each synthetic plant as a constant profile $v_x = -V_\star$, which coincides with the direction of the mean flow. On the remaining sides of the plants, the standard no-slip condition is used as we do not model advective transpiration on these sides for simplicity. On the upstream side, the velocity would have to be prescribed in the direction opposite to the mean flow, and on the sides parallel to the mean flow it would have to be prescribed in the tangential direction. However, water vapor is still released diffusively from the sides where the no-slip condition is used.

(a) EX-1



(b) EX-2



(c) EX-3

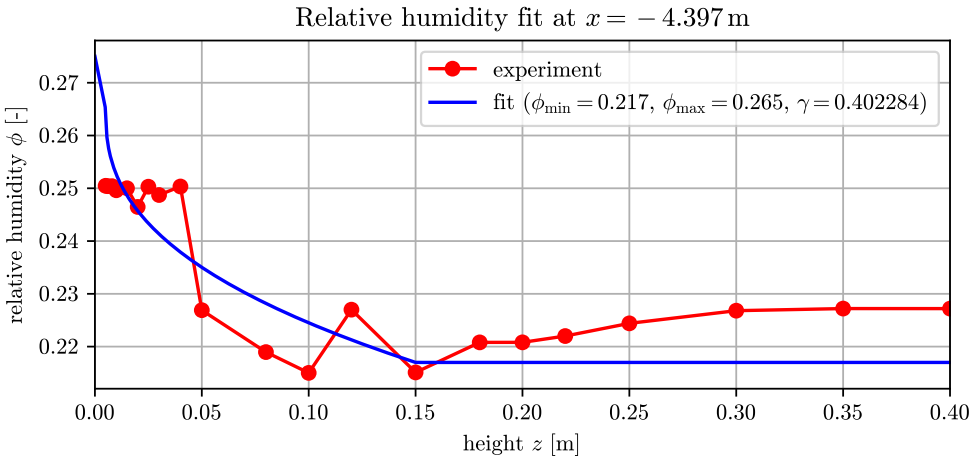
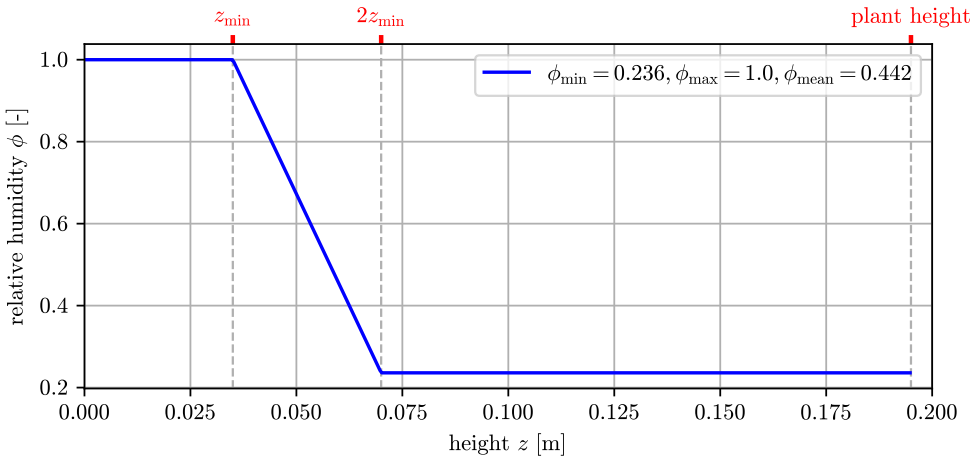
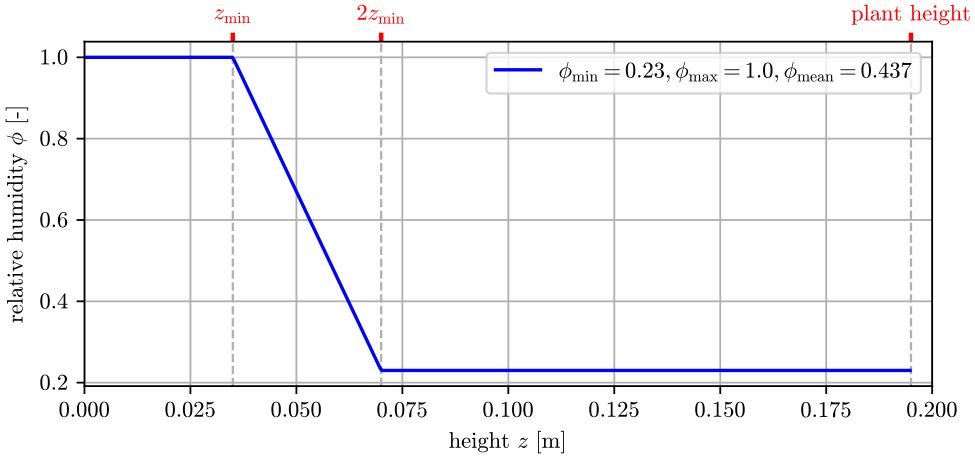


Figure 7.2: Relative humidity profiles prescribed at the inflow boundary in the spacing configurations EX-1, EX-2, and EX-3.

(a) EX-1



(b) EX-2



(c) EX-3

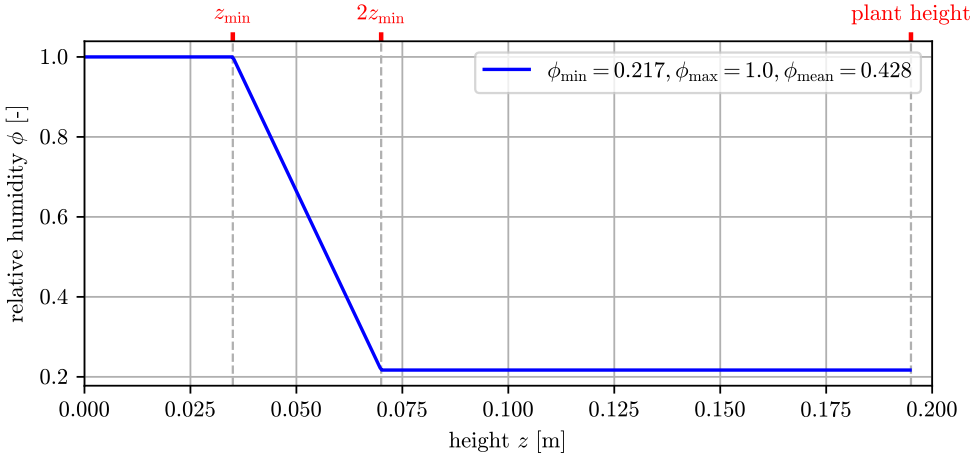


Figure 7.3: Relative humidity profiles prescribed as the boundary condition on the synthetic plants in the spacing configurations EX-1, EX-2, and EX-3.

7.3 Computational methodology

The coupled LBM-MHFEM solver described in Chapter 6 is used for the numerical solution of Equations (7.1) and (7.6b). Based on the experience from the previous chapter, the non-conservative form of the transport equation is used rather than the conservative form. The use of a regular lattice for the flow domain discretization is currently the main limiting factor for the flexibility of the solver, because extra care must be taken when setting up a simulation to ensure proper alignment of immersed boundaries such as the synthetic plants used in this chapter. This could be improved by using interpolated boundary conditions for LBM [B21], they are however not yet implemented in our solver. On the other hand, the MHFEM part of the solver can be used on complex domain geometries with unstructured mesh discretizations. In this chapter, we use conforming unstructured cuboidal meshes that are refined around the synthetic plants immersed in the domain.

The configuration of the solver is similar or the same as described in the previous chapter. Based on the results from Section 6.5, we use the linear interpolation of the velocity field and the explicit upwind scheme in MHFEM. Discrete boundary conditions are applied based on the continuous description from Section 7.2.3. In the MHFEM scheme, Dirichlet-type conditions are used to prescribe fixed values of relative humidity and the Neumann-type condition for the diffusive flux is used to prescribe zero gradient in the normal direction, see Section 4.2.6 for details. The following approaches are used the LBM part. We use the full-way bounce-back boundary condition [B21] to prescribe the no-slip condition on impermeable walls. The boundary condition for velocity on the downstream faces of synthetic plants (see Subsection "Synthetic plants" on Page 114) is realized via the modified bounce-back condition [B21] by specifying zero tangential and small non-zero normal velocity of the moving wall. On the inflow, the discrete distribution functions are approximated by the discrete equilibrium functions evaluated from the known macroscopic variables [A93, A121, A139]. On the outflow, the extrapolation outflow boundary condition is used to approximate the discrete distribution functions.

All three experimental configurations were simulated up to the final time $t_{\max} = 100\,\mathrm{s}$ using single precision for air flow and double precision for vapor transport in three different resolutions, hereafter denoted as RES-1, RES-2, and RES-3. A reference lattice and mesh were generated for the initial resolution RES-1 and the space step is halved with each subsequent resolution. The simulations were computed on the cluster operated by the *Research Center for Informatics* (`http://rci.cvut.cz/`). The compute node was equipped with two AMD EPYC 7763 processors and eight NVIDIA Tesla A100 GPU accelerators with NVLink interconnection. The decomposition algorithm for overlapped lattice and mesh described in Section 6.4 was used. See Table 7.4 for the characteristics of each resolution and computational resources needed for the simulations.

To match the experimental methodology described in Section 7.2.1, time-averaging was employed to produce statistical quantities such as the mean and variance of the flow velocity and relative humidity. Time-averaging is implemented as part of the simulation code where statistical quantities are updated in every time step using the Welford's online algorithm [A45, A126, A179]. Note that no space-averaging is applied to the simulation results.

7.3.1 Computational performance analysis

To demonstrate the computational efficiency of the implemented LBM-MHFEM solver, we performed a strong scaling study whose results are shown in Table 7.5 for the experimental configuration EX-1 in the resolution RES-2. Due to limited availability of the computational resources, the performance scaling analysis could be performed with only 8 GPUs on one node of the cluster. Same as in Chapter 5, the performance is evaluated in GLUPS (giga-LUPS, *billions of lattice updates per second*). Note that for the coupled solver, GLUPS is calculated by taking only the lattice size as the amount of work and dividing it by the total computational time (i.e., LBM plus MHFEM). While the metric with ignored mesh size does not fully express the performance of the coupled solver, it allows to directly compare the slowdown

Table 7.4: Characteristics of each resolution used for presented simulations. The computational times were achieved using 8 NVIDIA Tesla A100 cards with NVLink interconnection. The total computational time is broken down to cumulative contributions from LBM computation, velocity interpolation, and MHFEM computation; the remaining time includes initialization and output of the data.

	RES-1	RES-2	RES-3
Lattice space step	7.88 mm	3.94 mm	1.97 mm
Lattice dimensions	$496 \times 96 \times 144$	$991 \times 224 \times 288$	$1981 \times 480 \times 575$
No. of lattice sites	approx. 7×10^6	approx. 64×10^6	approx. 547×10^6
No. of mesh cells	approx. 1.5×10^6	approx. 12×10^6	approx. 96×10^6
Total memory	2.5 GiB	24 GiB	200 GiB
Base time step Δt	1.33×10^{-3} s	3.33×10^{-4} s	8.32×10^{-5} s
Average no. of LBM iters per MHFEM step ($\lfloor C_{\max}/C \rfloor$)	1	2	4
Computational time	10 min	65 min	15 h 12 min
– LBM computation	1 min	11 min	6 h 8 min
– velocity interpolation	46 s	2 min	30 min
– MHFEM computation	4 min	30 min	7 h 54 min

due to coupling compared to a standalone LBM solver. The performance of the coupled solver achieved in Table 7.5 is about 5-6× lower compared to the results in Table 5.3 for a standalone LBM solver. The efficiency decreases with increasing the number of GPUs used in the computation, which is a typical behavior in strong scaling analyses caused by reduced work per GPU and increased communication-to-work ratio. The coupled solver also requires a more complicated domain decomposition resulting in worse load balance between multiple GPUs and higher computational cost compared to a standalone LBM simulation. Considering that our implementation is limited by the one-dimensional domain decomposition of the lattice, the 80% efficiency achieved on 8 GPUs is a satisfactory result. Higher efficiency can be expected for weak scaling studies where the amount of work is kept proportional to the number of GPUs. However, we were not able to investigate it due to limited availability of the computational resources. Also note that due to the adaptive time stepping strategy used in the coupled solver, it is not straightforward to analyze the weak scaling, because the performance of the solver depends on the number of time steps where MHFEM is executed, which would be different for each problem size.

Table 7.5: Strong scaling of the coupled LBM-MHFEM solver for the scenario EX-1 in the resolution RES-2. N_{GPUs} denotes the number of NVIDIA Tesla A100 GPUs used in the computation, the Time column includes the computational time without initialization, the performance metric GLUPS stands for *billions of lattice updates per second* and *Eff* denotes the parallel efficiency.

N_{GPUs}	Time [min]	GLUPS	*Eff*
1	392	1.0	1.00
2	202	1.9	0.96
4	110	3.7	0.92
8	62	6.4	0.80

7.4 Validation results

Before describing the validation results themselves, let us summarize the methodology used for comparing simulation results with the physical experimental data.

In each experiment, data was collected pointwise in a series of one-dimensional vertical profiles at multiple locations streamwise along the length of the wind tunnel test-section in the $y = 0$ plane. The sparsity of these datasets prevents a reliable reconstruction of the flow field that can be compared quantitatively with the three-dimensional simulation results. To address this issue, simulation data were extracted at the same locations as the experimentally measured vertical profiles for direct quantitative comparison of the results.

While not as informative as a point-by-point comparison, a qualitative comparison of the simulated flow field with interpolated experimental data in 2D is still important for ensuring that the model is able to capture general patterns and behavior. Qualitative comparison is especially important when all of the different forms of experimental uncertainty associated with the measurements are considered. For example, the qualitative comparison via 2D flow fields might reveal whether the data considered for point-by-point quantitative comparison is translated in either the horizontal or vertical direction.

As previously discussed, relative humidity and flow statistics were measured with accuracies of ± 0.03 and 5 %, respectively. It should be noted that there are a large number of variables at play with respect to the experimental setup and methodologies employed by [A168]. Collectively, these are not accounted for in the model and are the likely explanation for any of the observed differences between the experimental and simulation results. A summary of possible factors contributing to experimental uncertainty include:

- The low sampling frequency of relative humidity measurements leads to a smearing effect due to spatio-temporal averaging of the vapor transport in the streamwise direction.
- The reliance on a sensor immersed in the flow to measure relative humidity is furthermore invasive, its presence disturbs the flow field locally and therefore the relative humidity distribution. The impact of the sensor was not quantified by [A168].
- The climate controls (i.e., heater, chiller, humidifier, dehumidifier) continuously fluctuated during the experiments, typically by no more than temperature $\pm 1\,^{\circ}\mathrm{C}$, relative humidity ± 0.03, and velocity $\pm 0.05\,\mathrm{m\,s^{-1}}$. This can lead to momentary increases or decreases in any of the aforementioned atmospheric variables. This variability is accounted for in the validation study by comparing statistical quantities (mean average and root-mean-square).
- Uncertainty in the physical placement of the bluff bodies within the test-section, see Section 7.2.1. The model and physical experiments may therefore differ slightly. Note that the measurements did not explore variability in the transverse direction (i.e., y-axis).
- Uncertainty in the physical locations where the flow and relative humidity measurements were made due to the accuracy of the automated traversing systems. If the laser was not perfectly centered behind a synthetic plant for example, it would measure slightly different flow behavior than it would otherwise. See Section 7.2.1 for more details.
- Uncertainty in the exact angle of the laser beams. If the beams were not exactly perpendicular to the flow, there could be some resulting skewness in the measured flow properties.

Finally, we would like to emphasize that the aim of the validation study is not to exactly reproduce the experimental results of [A168], nor is this even a feasible task since every model is a simplification of its prototype and thus cannot reproduce all of its characteristics [A61]. Our goal is to instead ensure that the model is able to capture the general patterns and behavior observed in the experimental data.

7.4.1 Qualitative comparison via 2D flow fields

Upon investigation of the velocity profiles from simulations using the time-constant (i.e., steady) inflow velocity profile given by Equation (7.7), we noticed that in the configurations EX-2 (45 cm spacing) and EX-3 (105 cm spacing), vortical structures induced by the flow around obstacles were different immediately behind the first and second synthetic plants. The difference is caused by the fact that when steady flow is prescribed at the inflow boundary, the flow immediately before the first plant is still steady, whereas the flow reaching the second plant is already turbulent. This scenario is non-physical, since natural air flow measured in the experiments was turbulent ($\mathrm{Re} > 10^4$). In order to introduce additional realism to the simulations and to address the above issue, the inflow velocity profile was modified through the inclusion of synthetic fluctuations as described in Subsection "Inflow: velocity fluctuations" on Page 113. The effect of small perturbations on the inflow boundary is that they enhance the development of the turbulent boundary layer in simulations with limited spatial resolution and domain size. A qualitative comparison of the mean horizontal velocity (v_x) fields simulated in the highest resolution RES-3 using both steady (time-constant) and unsteady (time-varying) boundary conditions is presented for the configurations EX-2 and EX-3 in Figure 7.4. Vortical structures caused by recirculating flow are observed downstream of both synthetic plants regardless of the applied boundary condition, but the overall size of this recirculating region is affected by the boundary condition. The steady inflow velocity boundary condition leads to the development of a region with low velocity downstream of the first plant in the flow direction that is larger than that observed when the time-varying boundary condition is applied. When these results are compared with the experimental data (i.e., the vertical columns in Figure 7.4), it is clear that the time-varying boundary condition captures the observed behavior better. Hence, only the time-varying inflow boundary condition is considered for the results discussed hereafter.

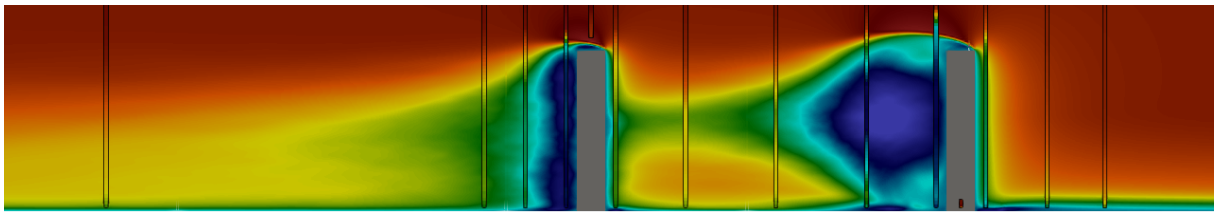
The profiles of mean horizontal and vertical velocity (v_x and v_z), root-mean-square of the turbulent horizontal and vertical velocity (RMS_x and RMS_z), and mean relative humidity (ϕ) along the plane $y = 0$ are shown in Figure 7.5 for EX-1, Figure 7.6 for EX-2, and Figure 7.7 for EX-3. In all three cases, the background color corresponds to the high-resolution simulation (RES-3) and the narrow vertical columns highlight superimposed experimental data measured at the corresponding locations.

The flow field observed around the synthetic plants depends on the spacing between the roughness elements [A182]:

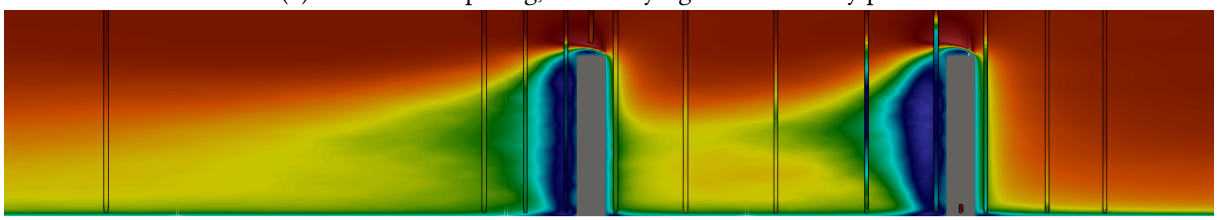
- In the closest spacing configuration (EX-1, Figure 7.5), the downstream synthetic plant is located within the turbulent wake of the upstream plant. In this scenario, stable vortices are formed between the plants and the regime is termed skimming flow.
- In the intermediate spacing configuration (EX-2, Figure 7.6), the wake created by the upstream plant does not fully develop before the flow reaches the downstream plant and affects the flow around it. Thus, this scenario is termed wake interference flow.
- In the widest spacing configuration (EX-3, Figure 7.7), the synthetic plants are spaced sufficiently far apart so as to allow for full development of individual wake zones. This flow regime is termed isolated roughness flow.

These flow regimes can be clearly discerned in the simulated flow fields shown in Figures 7.5 to 7.7. The skimming flow and the isolated roughness flow regimes seem to be captured well by the high-resolution simulation (RES-3). As discussed at the beginning of this section, the horizontal position of the wakes in the wake interference and isolated roughness flow regimes is affected by the inflow boundary condition used in the simulation. While the width of the simulated upstream wake in the isolated roughness flow regime (Figure 7.7) is in good agreement with the experiment, it is underestimated in the wake interference flow regime (Figure 7.6). Aside from this disagreement, the wakes formed behind the

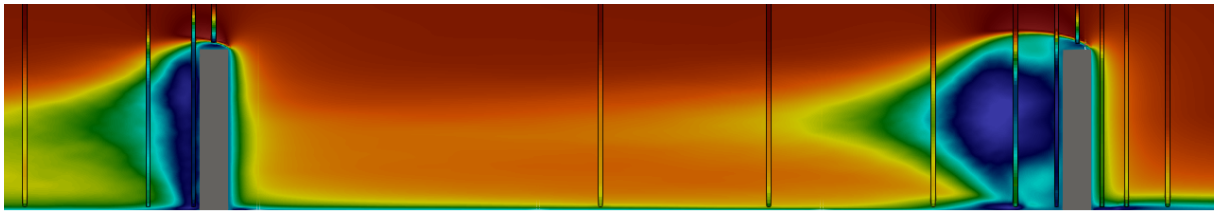
(a) EX-2: 45 cm spacing, time-constant inflow velocity profile



(b) EX-2: 45 cm spacing, time-varying inflow velocity profile



(c) EX-3: 105 cm spacing, time-constant inflow velocity profile



(d) EX-3: 105 cm spacing, time-varying inflow velocity profile

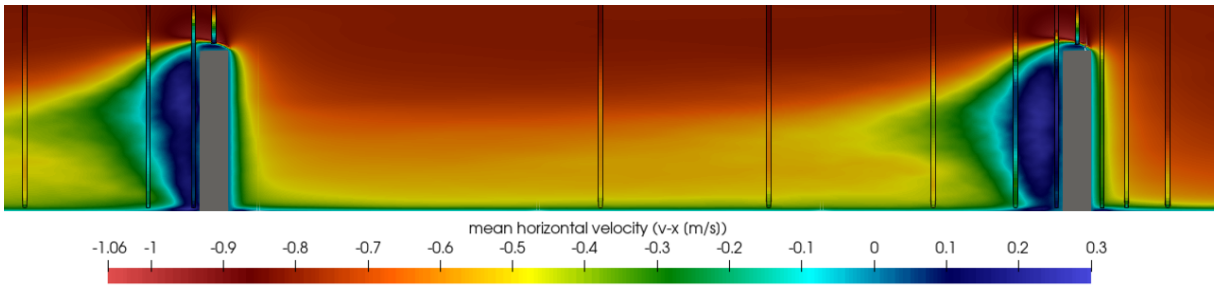


Figure 7.4: Simulated mean horizontal velocity (v_x) fields along the plane $y = 0$ with overlain 1D velocity profiles for the configurations EX-2 (45 cm spacing) and EX-3 (105 cm spacing). The velocity profile prescribed at the inflow boundary was either time-constant (Figures 7.4a and 7.4c) or time-varying (Figures 7.4b and 7.4d). Only a small region of interest around the synthetic plants is shown to improve visibility. The background color corresponds to the high-resolution simulation (RES-3); vertical columns show experimental data.

downstream plant as well as flow profiles far away from the plants are qualitatively in good agreement with the experiments in all three scenarios.

In the case of relative humidity, the greatest differences between experiments and simulations occur in the vertical profiles measured immediately downstream of the synthetic plants (i.e., on the left side of the gray rectangles in the figures). In the following section, we will quantify the differences in the profiles immediately downstream of the first plant and compare them with the measurements accuracy.

Note that all figures presented in this section correspond to simulations performed in the highest resolution RES-3. For completeness, additional 2D flow fields obtained in the lower resolutions RES-1 and RES-2 can be found in the supplementary materials for the paper [A111].

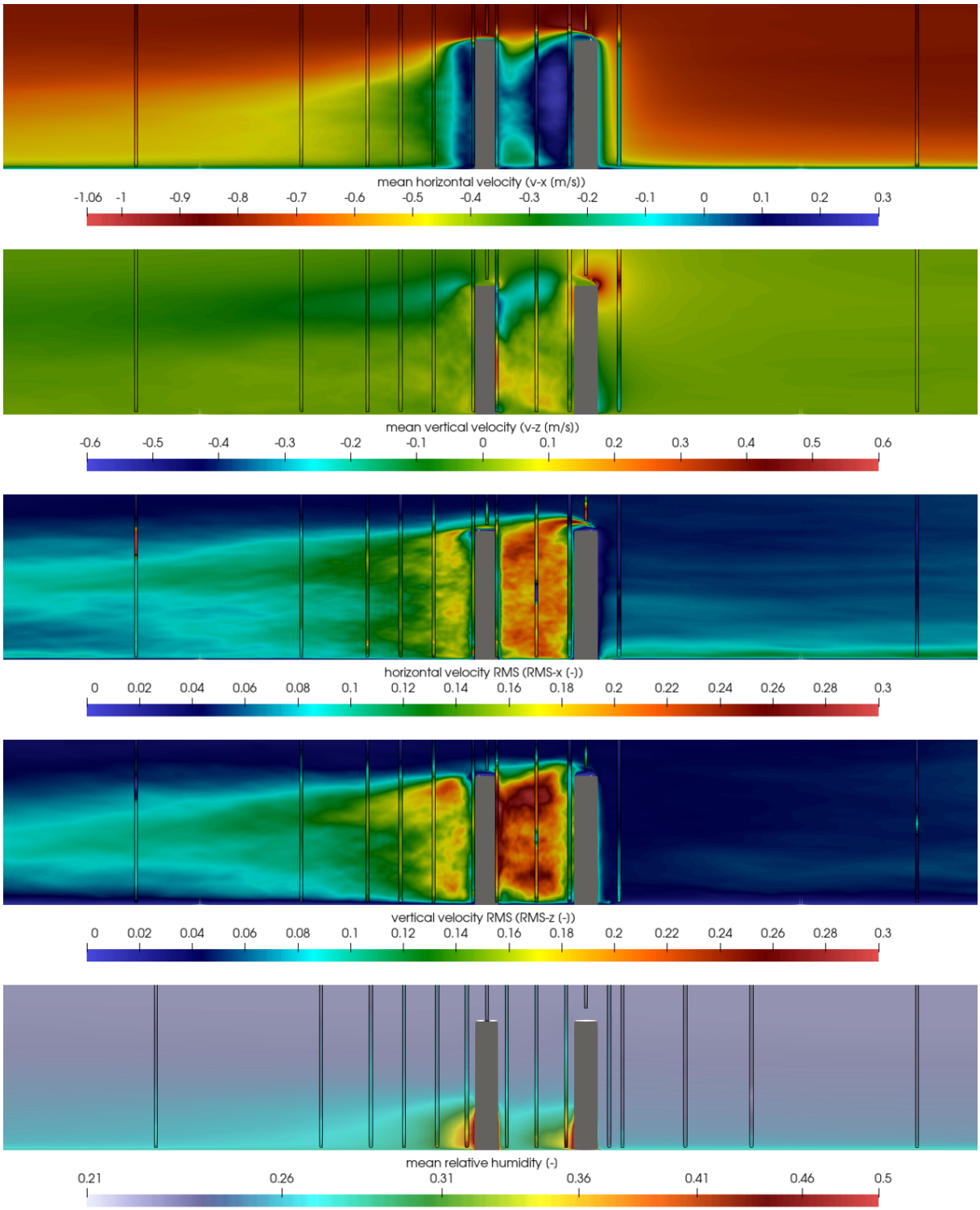


Figure 7.5: Simulated flow fields along the plane $y = 0$ with overlain 1D profiles: mean horizontal and vertical velocity (v_x and v_z), root-mean-square of horizontal and vertical velocity (RMS_x and RMS_z), and mean relative humidity (ϕ) for the configuration with 15 cm spacing between synthetic plants (EX-1). Only a small region of interest around the synthetic plants is shown to improve visibility. The background color corresponds to the high-resolution simulation (RES-3); vertical columns show experimental data.

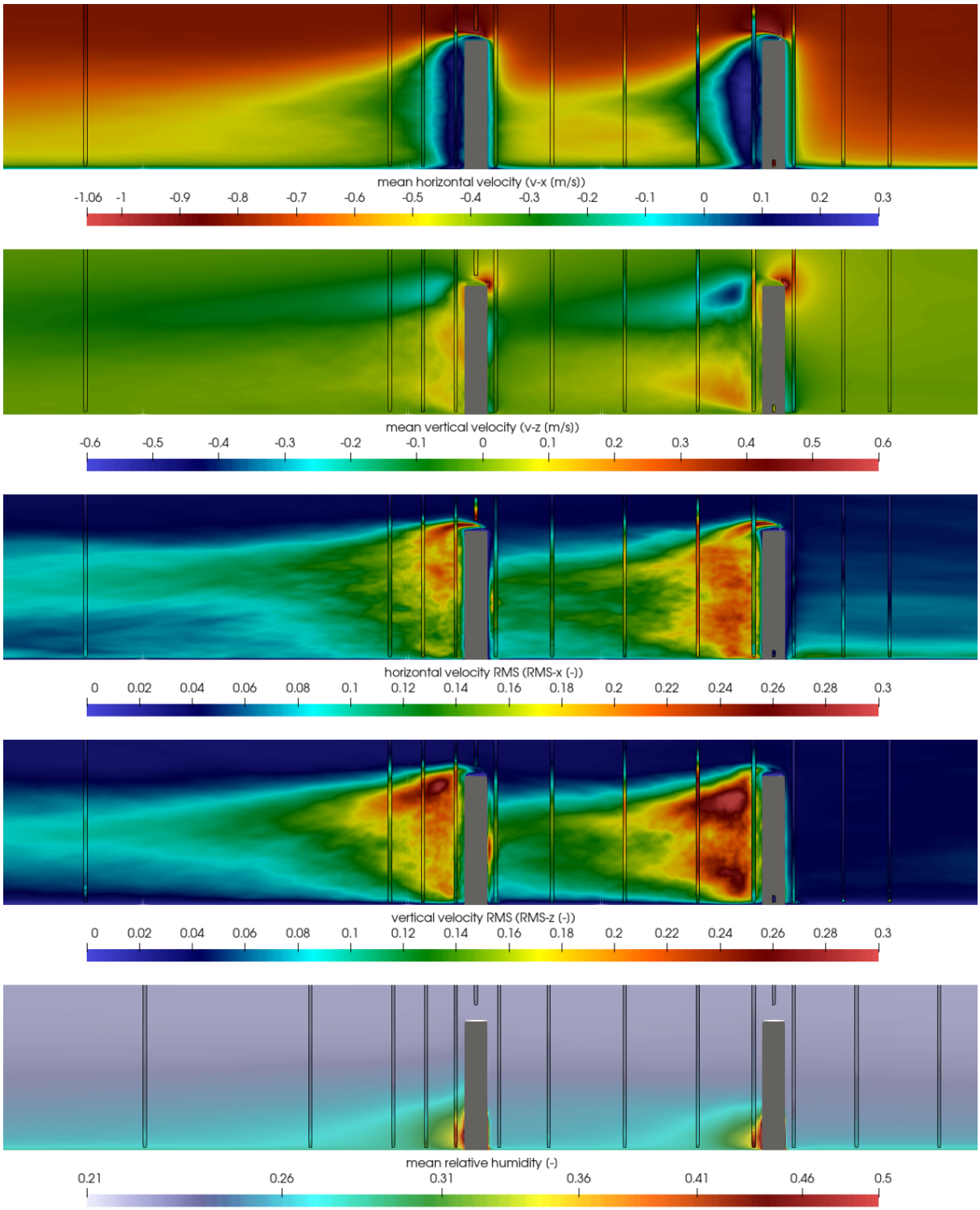


Figure 7.6: Simulated flow fields along the plane $y = 0$ with overlain 1D profiles: mean horizontal and vertical velocity (v_x and v_z), root-mean-square of horizontal and vertical velocity (RMS_x and RMS_z), and mean relative humidity (ϕ) for the configuration with 45 cm spacing between synthetic plants (EX-2). Only a small region of interest around the synthetic plants is shown to improve visibility. The background color corresponds to the high-resolution simulation (RES-3); vertical columns show experimental data.

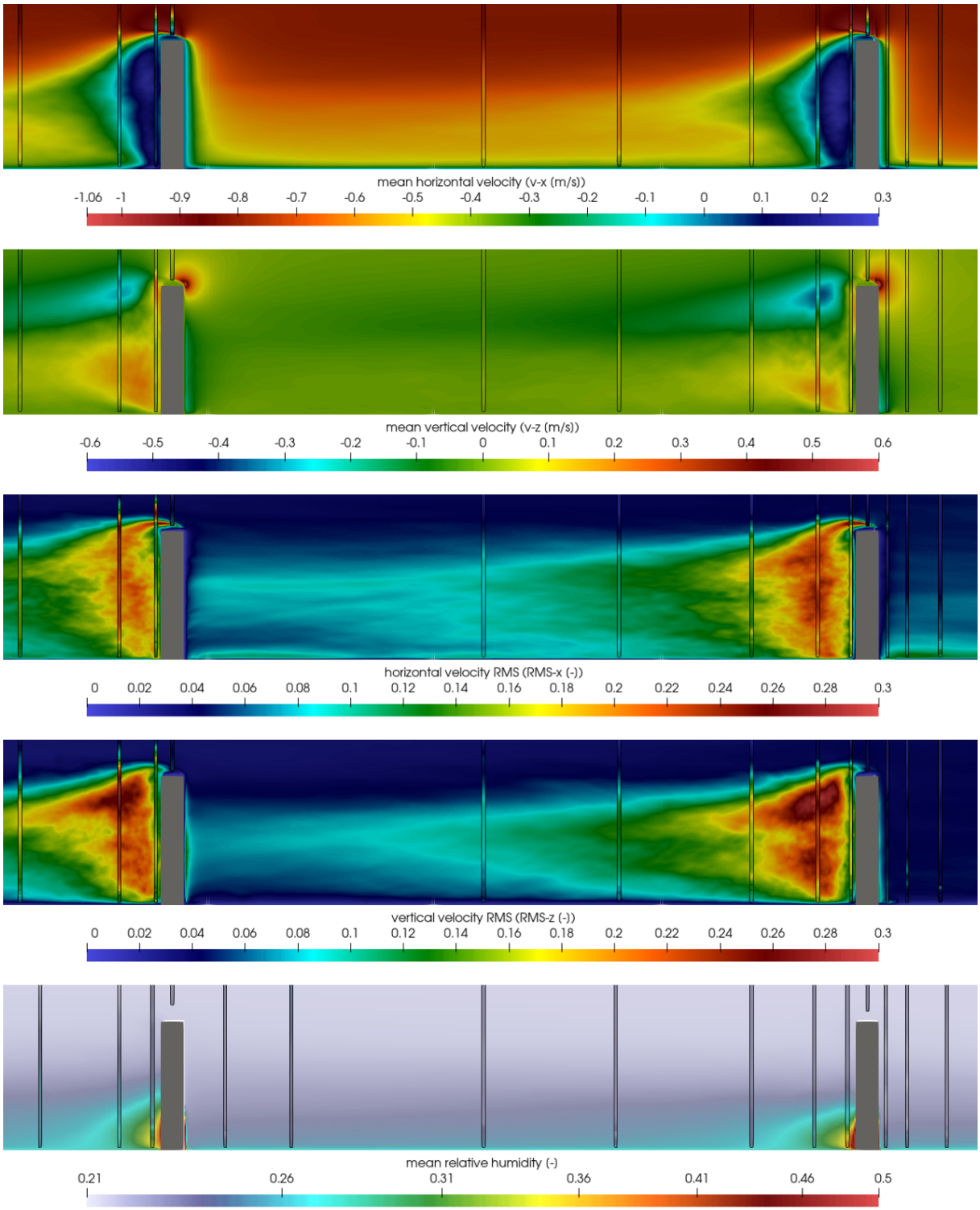


Figure 7.7: Simulated flow fields along the plane $y = 0$ with overlain 1D profiles: mean horizontal and vertical velocity (v_x and v_z), root-mean-square of horizontal and vertical velocity (RMS_x and RMS_z), and mean relative humidity (ϕ) for the configuration with 105 cm spacing between synthetic plants (EX-3). Only a small region of interest around the synthetic plants is shown to improve visibility. The background color corresponds to the high-resolution simulation (RES-3); vertical columns show experimental data.

7.4.2 Quantitative comparison via 1D graphs

In this section, the simulation results and experimental data are compared quantitatively in the vertical columns immediately downstream of the first synthetic plant. For completeness, comparisons in the other measurement locations are included in the supplementary materials for the paper [A111].

The graphs of the mean velocity (v_x and v_z) profiles are shown in Figures 7.8 to 7.10 and the graphs of the relative humidity (ϕ) profiles are shown in Figures 7.11 to 7.13. For convenience, the horizontal position of each profile is indicated schematically by the red bar above each graph on the right hand side. In each graph, the experimental data (red line) and simulation data in three resolutions (dark blue line for RES-1, light blue line for RES-2, orange line for RES-3) are compared. The dashed lines indicate the statistical deviation of the respective simulated quantity from its mean value (full lines). In case of the measured velocity profiles in Figures 7.8 to 7.10, the statistical deviation is indicated by errorbars. On the other hand, the low sampling frequency of the RHT sensors prevents any information on the turbulent transport of the water vapor from being inferred from the data. In this case, the errorbars in Figures 7.11 to 7.13 represent the sensor accuracy rather than statistical deviation of the samples.

The graphs in Figures 7.8 and 7.10 for the cases EX-1 and EX-3, respectively, show the best match between the simulated velocity profiles and experimental data. It can be observed that of the three resolutions compared in the graphs, the low-resolution simulation RES-1 differs the most from the experimental data and the high-resolution simulation RES-3 is closest to the experimental data. For the remaining case EX-2, the graphs in Figure 7.9 show larger differences that correspond to the qualitative disagreement described in the previous section. The primary cause is presumably the uncertainty related to the exact conditions in the experiment. Upon closer examination of the documentation from this experiment, we found that the upstream plant may have been planted not perfectly perpendicular to the ground surface, which would result in different flow behavior around and above the plant. However, there is no data that would allow us to reproduce this scenario in the simulation.

In case of the relative humidity profiles, the largest difference between the experimental data and simulations is observed in Figure 7.11 for EX-1 where the simulated profiles are underestimated. These results suggest that improvements could be made by modifying the boundary condition applied on the plant. The boundary condition described in Subsection “Synthetic plants” on Page 114 is based on a mass flux that was measured in a separate experiment under different atmospheric conditions [A169] than those in the configurations EX-1, EX-2, and EX-3. The mass fluxes used in the model may therefore underestimate the conditions actually present in the investigated scenarios. Furthermore, it should be noted that when the plants are located close together, they may compete for the limited resource [A168] which might result in increased collective evaporation rate compared to the cases where the plants are located far away from each other. In the cases EX-2 and EX-3 featuring different flow regimes, the graphs in Figures 7.12 and 7.13 indicate that the simulation results are well within the accuracy of the RHT sensors.

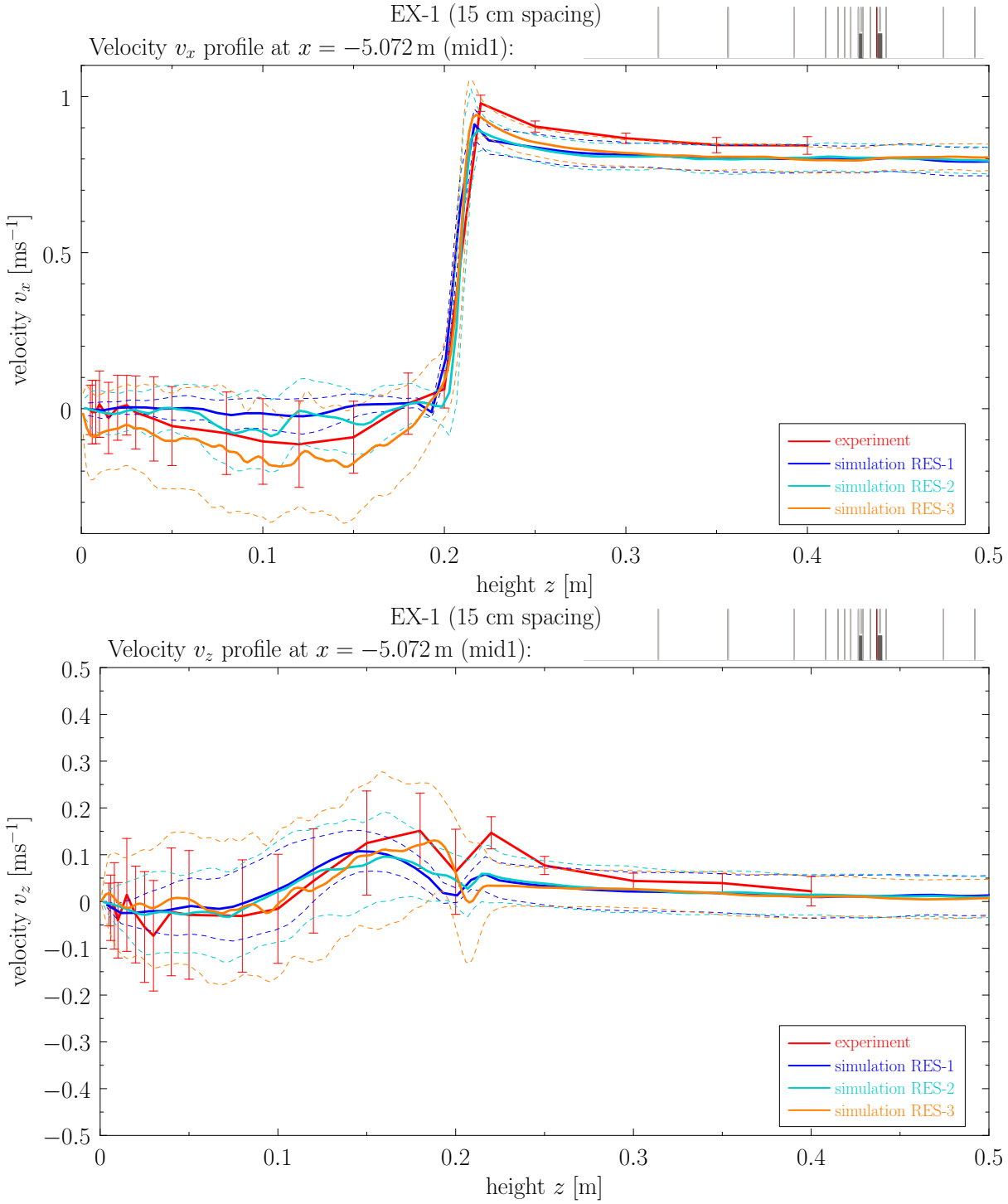


Figure 7.8: Quantitative comparison of horizontal and vertical velocity profiles (v_x and v_z) at the first position downstream of the first synthetic plant in the 15 cm spacing (EX-1). The red bar above each graph highlights the position of the profile relative to the synthetic plants (dark rectangles) and other measurement locations (thin gray bars).

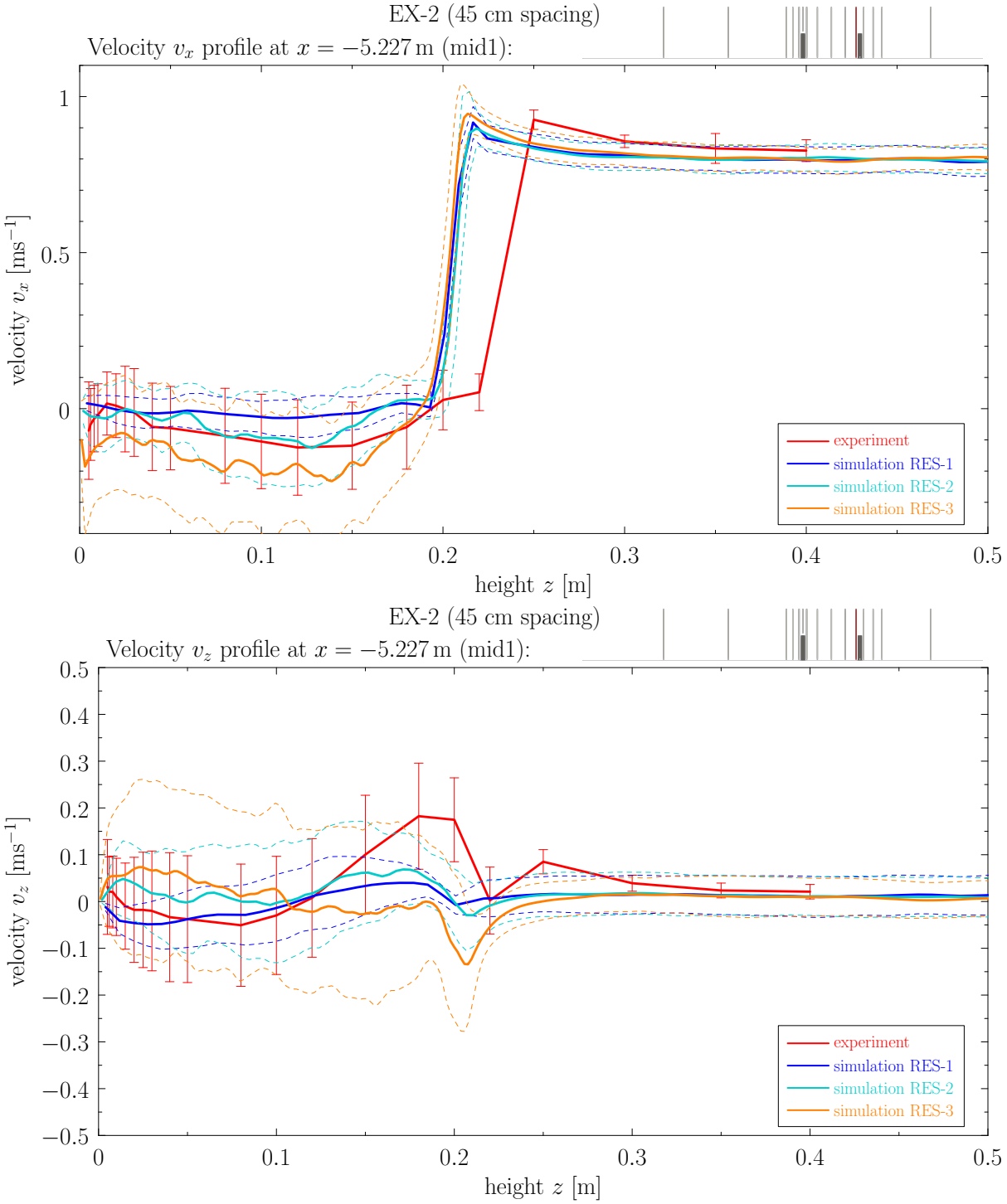


Figure 7.9: Quantitative comparison of horizontal and vertical velocity profiles (v_x and v_z) at the first position downstream of the first synthetic plant in the 45 cm spacing (EX-2). The red bar above each graph highlights the position of the profile relative to the synthetic plants (dark rectangles) and other measurement locations (thin gray bars).

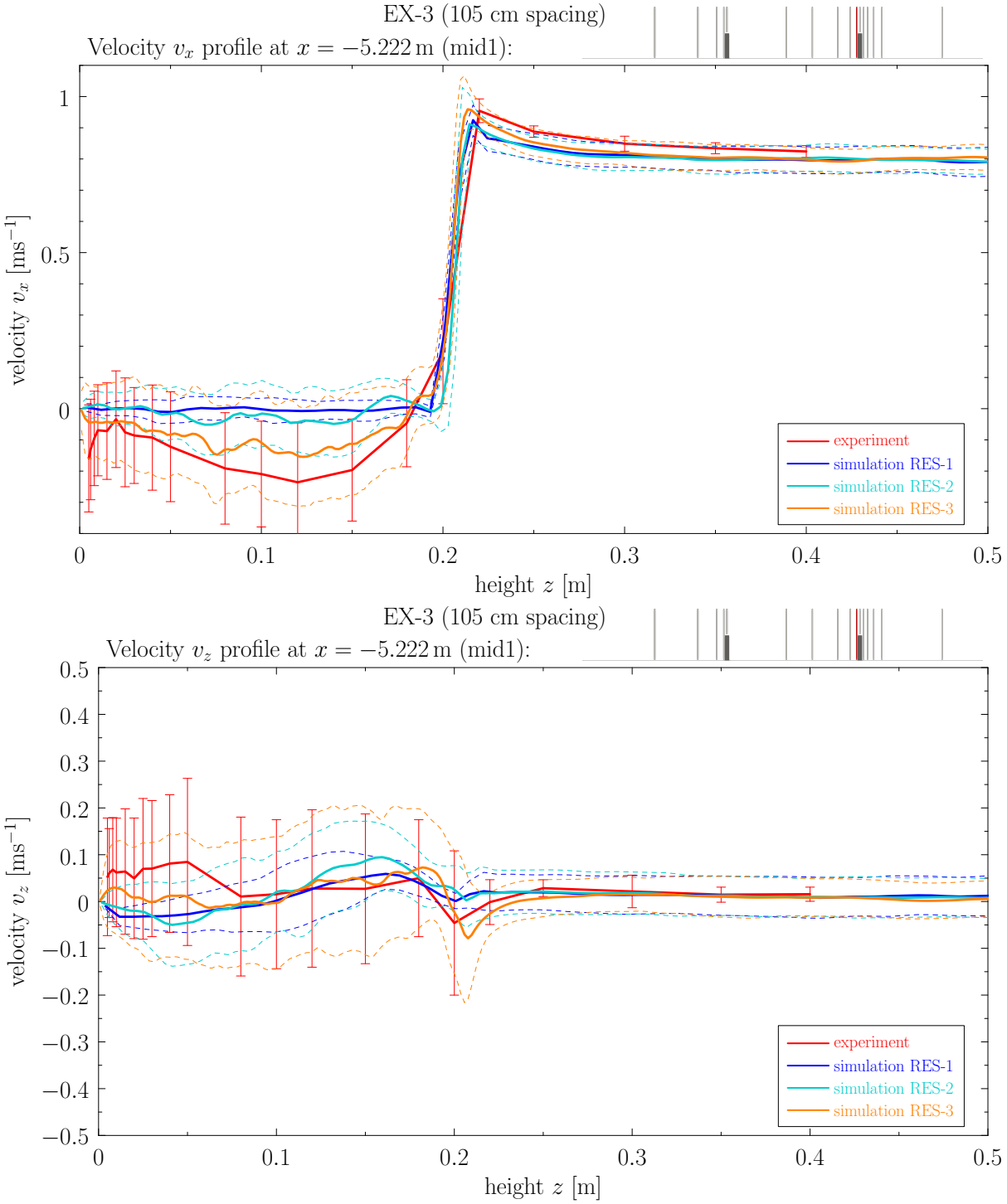


Figure 7.10: Quantitative comparison of horizontal and vertical velocity profiles (v_x and v_z) at the first position downstream of the first synthetic plant in the 105 cm spacing (EX-3). The red bar above each graph highlights the position of the profile relative to the synthetic plants (dark rectangles) and other measurement locations (thin gray bars).

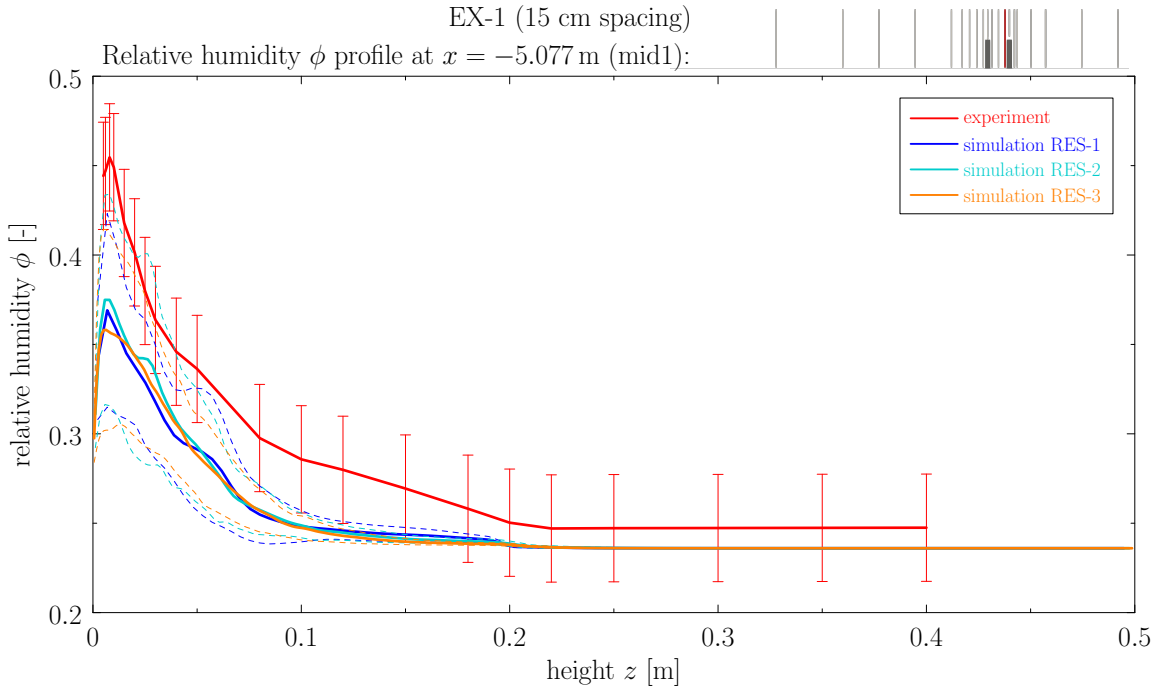


Figure 7.11: Quantitative comparison of relative humidity profiles (ϕ) at the first position downstream of the first synthetic plant in the 15 cm spacing (EX-1). The red bar above the graph highlights the position of the profile relative to the synthetic plants (dark rectangles) and other measurement locations (thin gray bars).

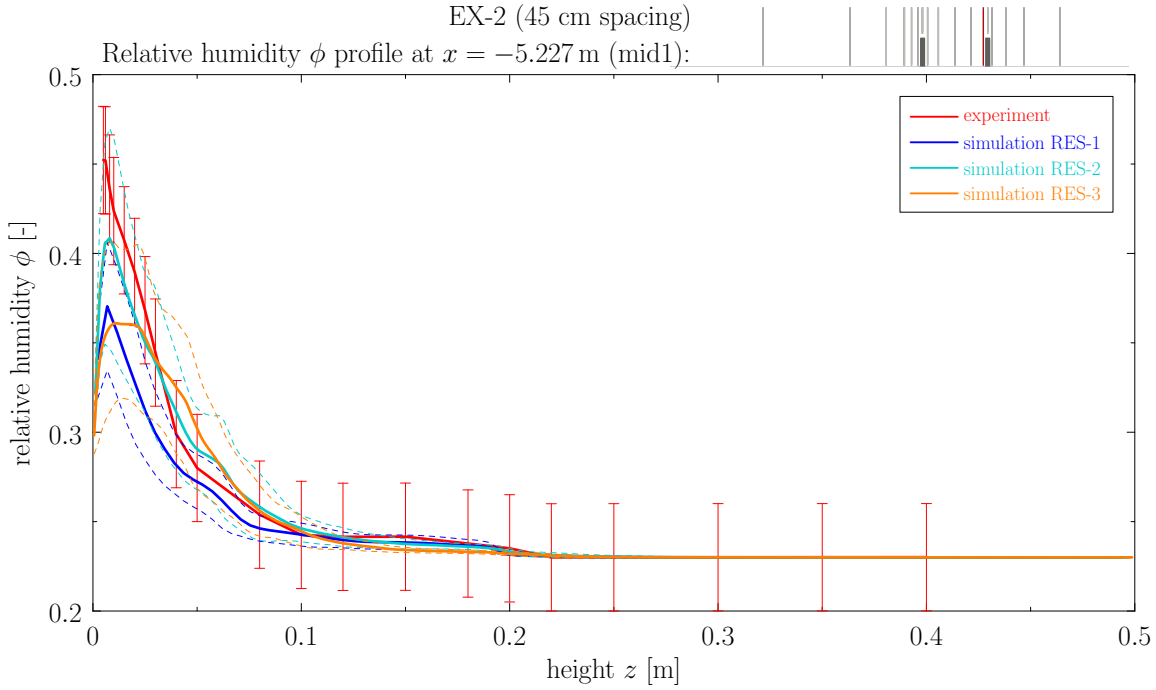


Figure 7.12: Quantitative comparison of relative humidity profiles (ϕ) at the first position downstream of the first synthetic plant in the 45 cm spacing (EX-2). The red bar above the graph highlights the position of the profile relative to the synthetic plants (dark rectangles) and other measurement locations (thin gray bars).

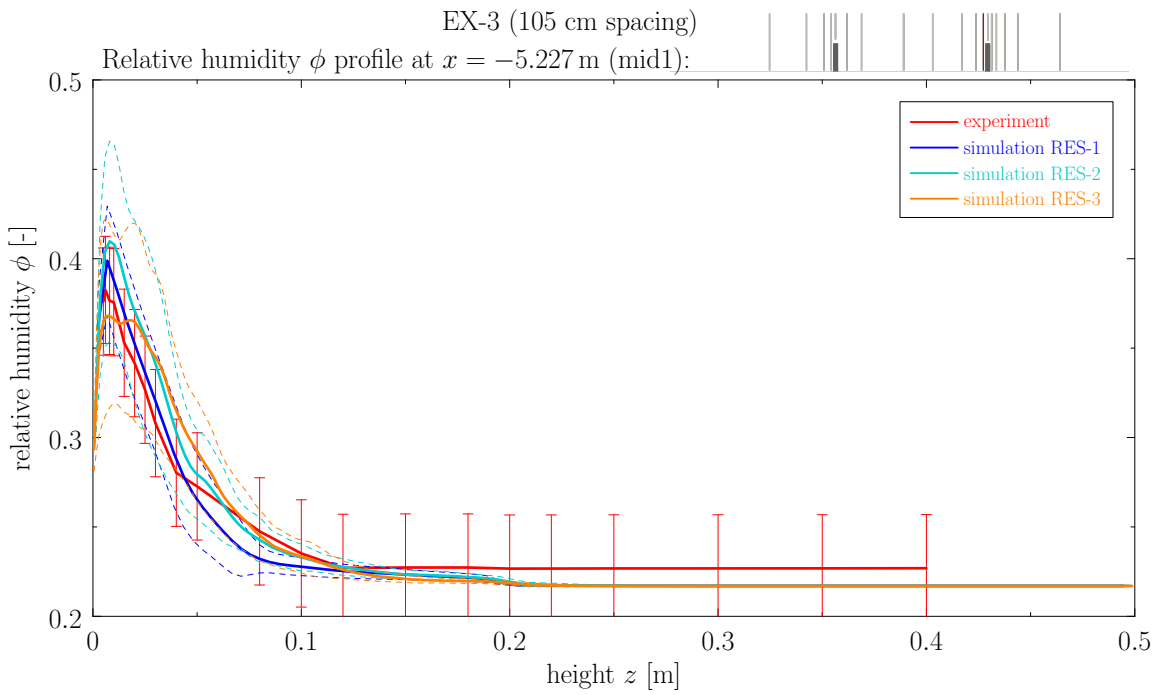


Figure 7.13: Quantitative comparison of relative humidity profiles (ϕ) at the first position downstream of the first synthetic plant in the 105 cm spacing (EX-3). The red bar above the graph highlights the position of the profile relative to the synthetic plants (dark rectangles) and other measurement locations (thin gray bars).

Conclusion

The following sections summarize the results presented in the previous chapters and potential future research directions.

1 Programming techniques for modern parallel architectures

In Chapter 1, we introduced various approaches for programming modern parallel architectures. First, contemporary high-performance computing systems were introduced in a brief summary and the C++ programming language was selected for the work presented in the thesis. Then, we presented an overview of common parallel programming frameworks and illustrated the implementation of a parallel *axpy* operation in each framework. The Message Passing Interface for distributed computing was also briefly described. Subsequently, we described several high-level parallel programming libraries with backend systems that provide performance portability across multiple hardware platforms and/or parallel programming frameworks. Examples of the parallel *axpy* operation in each library are included for comparison with the frameworks. Finally, the Template Numerical Library was introduced and its features, design, and future work were described.

2 Data structures

In Chapter 2, two efficient and configurable data structures from the Template Numerical Library were described: multidimensional arrays and unstructured meshes. The section related to multidimensional arrays amends the implementation described in the author's Master's thesis [B20] and thus it focuses on describing the extension for distributed computing.

The section related to unstructured meshes provides a detailed description of the design and implementation of the data structure published in [A110] and its later extension for polygonal and polyhedral meshes. The data structure is designed around sparse matrix formats for the representation of incidence matrices, supports computations on CPUs as well as GPUs, and supports distributed computing via MPI. Its efficiency was compared using several benchmark problems based on simple parallel algorithms. Compared to the data structure available in the MOAB library, the primary benchmark using the TNL data structure is about 13× faster for triangular meshes, 5× faster for tetrahedral meshes, 10× faster for polygonal meshes, and 6× faster for polyhedral meshes. However, for the alternative benchmark requiring more information from the mesh data structure, the factor rises up to 130× for tetrahedral meshes. Furthermore, the results indicate good GPU utilization for all benchmarks and mesh types, including polygonal and polyhedral.

3 Solution of sparse linear systems

In Chapter 3, we presented a review of iterative methods and preconditioning techniques for the solution of sparse linear systems. Additionally, we presented an overview of software packages

implementing related algorithms and described corresponding features available in the TNL library. Finally, we described details related to the implementations of distributed sparse matrices in the TNL and Hypre libraries.

4 Mixed-hybrid finite element method

In Chapter 4, we described a numerical scheme for the solution of a system of partial differential equations in a general coefficient form, called *NumDwarf*. The scheme is based on the mixed-hybrid finite element method (MHFEM) and the discontinuous Galerkin method for spatial discretization, the Euler method for temporal discretization and the semi-implicit approach of the frozen coefficients method for the linearization in time. The scheme was developed in [A72] originally for multicomponent flow and transport phenomena in porous media. The chapter builds on the paper [A72] and author's Master's thesis [B20], but includes more details and variants of the scheme, such as implicit upwind stabilization for advective terms. The implementation of the scheme relies on the TNL library, especially the data structure for unstructured meshes and all parallel computing capabilities.

The chapter briefly introduces the mathematical model of incompressible two-phase flow in porous media and the generalized McWhorter–Sunada problem, which is used as a benchmark problem to analyze the accuracy and computational performance of the scheme. The verification results show first order of accuracy in the L_1 and L_2 norms in 2D and 3D. The benchmark was computed in several configurations with varying parameters, namely the linear system solver, hardware architecture, and programming framework for CPU parallelization. The results show the importance of a good solver for systems of linear equations, which takes most of the computational time. The BiCGstab method with the BoomerAMG preconditioner from the Hypre library performs several times faster than the Jacobi-preconditioned BiCGstab method on CPU. Both variants have similar strong scalability based on the increasing number of CPU cores. For GPU computations, the difference between the performance of the BoomerAMG and Jacobi preconditioners is not as significant, but the former still performs better and its advantage might be even more considerable for larger meshes.

5 Lattice Boltzmann method

In Chapter 5, we described the implementation of the lattice Boltzmann method and performance optimizations necessary for its efficient use on distributed systems with GPU accelerators. All components of the method are described with the objective to formulate the computational algorithm. Two streaming schemes based on the A-B and A-A patterns are explained in detail and tested in a performance benchmark. The optimizations include overlapping computation and communication, pipelining for operations in the distributed data synchronization, and using CUDA streams and MPI functions efficiently in the implementation.

A small computational benchmark was performed on several NVIDIA GeForce and NVIDIA Tesla GPUs to evaluate the difference between the A-B and A-A streaming patterns. In almost all cases, the A-A pattern performed better or slightly worse in terms of GLUPS, but its main advantage are halved memory requirements for the storage of discrete distribution function values. A larger computational benchmark was performed for the A-A pattern on the Karolina supercomputer using accelerated nodes with 8 NVIDIA A100 GPUs each. The results show good strong scalability up to 8 nodes (64 GPUs) using a $512 \times 512 \times 512$ lattice in single as well as double precision. Two weak scaling studies with 1D and 3D domain expansion show almost ideal scalability up to 16 nodes (128 GPUs), which is the most that were tried in the benchmark.

An important limitation of the implemented solver is the one-dimensional domain decomposition; implementation of a multi-dimensional decomposition algorithm is planned for the future. The per-

formance of the solver in the strong scaling study is further limited due to dysfunctional GPUDirect technology on the supercomputer. Another problem that may be important for modern supercomputers is the optimal mapping of MPI ranks to GPUs, which would consider non-uniform communication costs between each pair of GPUs. This leads to the quadratic assignment problem, which is NP-hard, and the weights approximating communication costs need to be measured experimentally or provided by a network topology–aware hardware introspection tool.

6 Coupled LBM-MHFEM computational approach

In Chapter 6, we presented a coupled computational approach based on the combination of lattice Boltzmann and mixed-hybrid finite element methods for the solution of the Navier–Stokes equations coupled with a general system of advection–diffusion–reaction partial differential equations. The work presented in this chapter explores new possibilities for efficient solution of various multiphysics problems using modern hardware architectures.

Numerical details are provided for the coupled computational algorithm, time adaptivity, and interpolation of the velocity field. Thanks to the TNL library, the solver can utilize modern GPU-based high-performance computing systems. For optimal utilization of computational resources, we designed a domain decomposition algorithm for overlapped lattice and mesh, which allows to optimize the computational cost and memory requirements of each MPI rank at the cost of increased communication due to increased number of lattice subdomains. The decomposition algorithm is essentially one-dimensional and its generalization to improve the scalability on large supercomputers may be subject of future research.

A simple benchmark problem based on a highly turbulent velocity field and a linear advection–diffusion equation with an analytical solution was designed to analyze the accuracy of the coupled numerical scheme. The results show that the numerical schemes for the conservative and non-conservative forms of the advection–diffusion equation do not behave equivalently. Thanks to a term that compensates for non-zero velocity divergence, the accuracy of the non-conservative scheme is better by about 10 orders of magnitude in the benchmark. Furthermore, the benchmark shows small differences between linear and cubic interpolation schemes and between explicit and implicit upwind stabilization for advective terms.

7 Mathematical modeling of vapor transport in air

In Chapter 7, we used the coupled computational approach developed in the previous chapter to simulate vapor transport in the boundary layer above a partially saturated soil. The chapter concludes the multidisciplinary work presented in this thesis with mutual collaboration between experimental and computational methodologies.

The mathematical model was validated with experimental data measured above a flat partially saturated soil layer featuring synthetic plants arranged in several configurations. The experimental dataset used in this study was generated by [A168, A169] and is publicly available in [O36]. The model relies on experimental data for the specification of boundary conditions: the inflow velocity and humidity profiles and the average mass flux of water loss from the plants. Based on the presented validation study, we can draw reasonable predictions about the flow and transport behavior inside the computational domain.

The performance of the coupled solver depends on the selected lattice and mesh sizes (i.e., spatial resolution) and the adaptively selected time steps. The highest-resolution simulations, which compare the best to the experimental data, require about 200 GiB memory and 15.25 h computational time on 8 NVIDIA Tesla A100 cards to simulate 100 s of physical time. The simulations in lower resolutions

are not as accurate, but require less memory and shorter computational time compared to the highest resolution. A strong scaling analysis was performed for a lower resolution giving a parallel efficiency of 80% on 8 NVIDIA Tesla A100 cards. Scalability problems that are likely to occur on large-scale supercomputers were not investigated due to the availability of computational resources.

The presented results suggest several key areas where future experimental efforts could be improved, allowing the analysis of this model's performance to be extended and further explored. For example, extending measurements with flow characteristics in the transverse direction (e.g., v_y, RMS_y, $\overline{v'_x v'_y}$, $\overline{v'_y v'_z}$) would allow us to compare the turbulent kinetic energy and improve the fluctuating inflow velocity condition for the simulations. Another possible improvement is to arrange measurements in horizontal profiles in regions behind the plants, which would allow us to study the convergence of the numerical method (i.e., the effect of mesh resolution) by comparing the horizontal location of the vortical structures. Last but not least, the applicability of the measured evaporative mass flux to the close spacing scenario EX-1 should be investigated. Improving the methodology for measuring the evaporation from the plants would allow for prescribing more appropriate boundary conditions.

The presented simulator for vapor transport in air is just a first application of the coupled LBM-MHFEM approach and could be extended into a more general software tool capable of solving other physical phenomena such as non-isothermal flow, multicomponent flow, land-atmospheric interaction, etc. There are many potential applications in combination with the experimental research, such as developing an efficient tool for a sensitivity analysis of measurements, supplementing sparse experimental datasets in regions where measurements would be too expensive or unfeasible, or predicting the behavior of the studied system in virtual scenarios.

Appendix A

Default Mesh Configuration

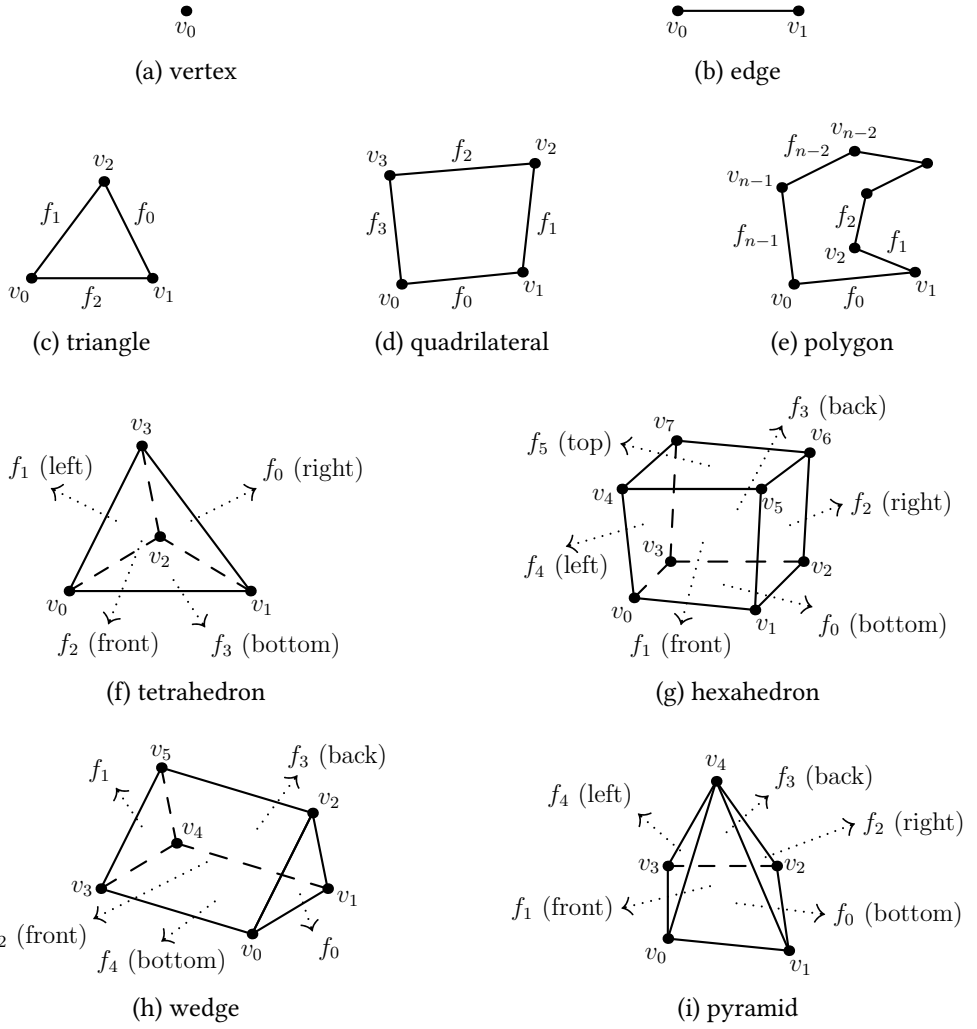
The TNL library provides the following class template as the default mesh configuration:

```
template< typename Cell,
int SpaceDimension = Cell::dimension,
typename Real = double,
typename GlobalIndex = int,
typename LocalIndex = GlobalIndex >
struct DefaultConfig
{
   using CellTopology = Cell;
   using RealType = Real;
   using GlobalIndexType = GlobalIndex;
   using LocalIndexType = LocalIndex;

   static constexpr int spaceDimension = SpaceDimension;
   static constexpr int meshDimension = Cell::dimension;

   // Storage of subentity links.
   static constexpr bool subentityStorage( int entityDimension, int subentityDimension )
   {
      return true;
   }

   // Storage of superentity links.
   static constexpr bool superentityStorage( int entityDimension, int superentityDimension )
   {
      return true;
   }

   // Storage of boundary tags of mesh entities.
   static constexpr bool entityTagsStorage( int entityDimension )
   {
      return superentityStorage( meshDimension - 1, meshDimension ) &&
      ( entityDimension >= meshDimension - 1 || subentityStorage( meshDimension - 1, entityDimension ) );
   }

   // Storage of the dual graph.
   static constexpr bool dualGraphStorage()
   {
      return true;
   }

   // Parameter n_{common} for the generation of the dual graph.
   static constexpr int dualGraphMinCommonVertices = meshDimension;
};
```

Appendix B

Mesh Entity Topologies

In non-polyhedral meshes, cells are specified by their subvertices and every other subentity is determined implicitly by the cell topology and its subvertices. Hence, the subvertices of a cell must follow a specific order which is shown in Figure B1 for the entity topologies currently implemented in TNL (except polyhedron). The ordering of the subvertices is inspired by VTK [B27] and extended with a specific numbering of other subentities which can be utilized by numerical schemes. For example, the faces of a simplex are numbered such that each face has the same local index as the opposite vertex.



(a) vertex (b) edge

(c) triangle (d) quadrilateral (e) polygon

(f) tetrahedron (g) hexahedron

(h) wedge (i) pyramid

Figure B1: Mesh entity topologies in the TNL library and the ordering of subvertices and faces.

Appendix C

Mesh Storage Layers

The inheritance diagram for the `Mesh` class template is shown in Figure C1. The template parameters `Config`, which represents the mesh configuration (see Subsection "Static configuration" on Page 33), and `Device`, which represents the *device* in which memory space the mesh will be allocated (see Section 1.3.2), are propagated to all base class templates in the hierarchy. The first base class template is `StorageLayerFamily` which does not have any data members and its purpose is to provide access via tag dispatching to all individual storage layers. The recursive inheritance is realized in the `StorageLayer` class template which is parametrized by the entity dimension tag, `DimTag<d>`, and comprises all data related to d-dimensional mesh entities. The `StorageLayerFamily` class template starts the inheritance with the dimension 0 and each storage layer inherits from the same class template, but with an incremented dimension tag. Finally, recursion is terminated by the empty $(D+1)$-dimensional partial class template specialization.

The inheritance diagram for the general d-dimensional `StorageLayer` class template is shown in Figure C2 (the inheritance between the instances of `StorageLayer` with the parameters `DimTag<d>` and `DimTag<d+1>` has been explained in Figure C1). The inheritance scheme is similar to Figure C1, now using two separate storage layer families for the data related to subentities and superentities of d-dimensional entities. We introduce the `SubentityStorageLayer` and `SuperentityStorageLayer` class templates which are parametrized by the entity dimensions d and d' and store one specific incidence matrix each: `SubentityStorageLayer` stores $I_{d,d'}$, where $d > d'$, and `SuperentityStorageLayer` stores $I_{d,d'}$, where $d < d'$. By default, both `SubentityStorageLayer` and `SuperentityStorageLayer` also store the array containing the

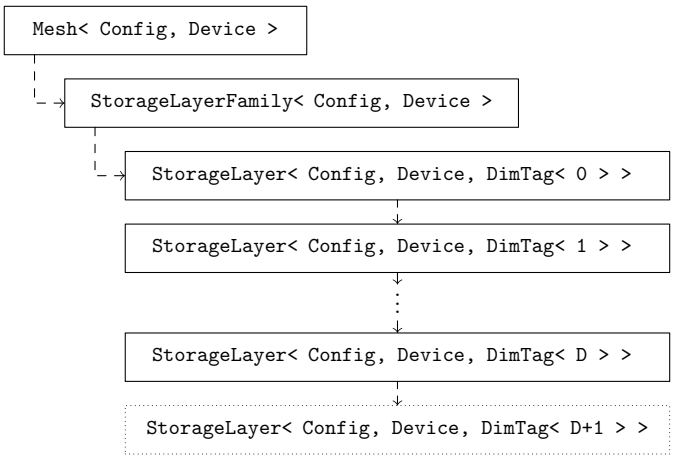


Figure C1: Inheritance scheme of the D-dimensional `Mesh` class template. The $(D+1)$-dimensional partial specialization of the `StorageLayer` class template is an empty class template used to terminate the recursive inheritance.

Figure C2: Inheritance scheme of the d-dimensional specialization of the `StorageLayer` class template for a D-dimensional mesh. The d-dimensional partial specializations of the `SubentityStorageLayer` and `SuperentityStorageLayer` class templates are empty class templates used to terminate the recursive inheritance.

numbers of non-zero elements per row (see Subsection “Internal data structures” on Page 33). The `SubentityStorageLayer` template has a partial class template specialization for static entity topologies, which omits the array with numbers of non-zero elements per row and uses a compile-time constant instead. Note that the inheritance for `SubentityStorageLayer` is started by $d' = 0$ and continues by incrementing d', but for the `SuperentityStorageLayer` it is started by $d' = D$ and continues by decrementing d'. In both cases, the recursive inheritance which is terminated by an empty partial class template specialization when the entity dimension is the same as the subentity or superentity dimension (i.e., $d = d'$).

In the following code examples, we outline the implementation of the scheme from Figure C1 using recursive inheritance and partial class template specialization. The recursive inheritance can be written in a simplified form as follows:

```
1 template< typename Config, typename Device, typename DimTag >
2 class StorageLayer
3 : public StorageLayer< Config, Device, typename DimTag::Increment >
4 {
5     // implementation of member functions is omitted
6 };
```

It is assumed that the `DimTag` parameter is initially supplied a 0-dimensional tag which is incremented by one to inherit the next storage layer. Note that due to a C++ language limitation, we have to use the `DimTag` type template parameter instead of a non-type template parameter such as `int d` to represent the dimension, because expressions involving other type template parameters cannot be used to partially specialize a non-type template parameter. After all storage layers for the given mesh have been used, an empty partial class template specialization is supplied in order to terminate the recursive inheritance. The empty specialization is identified by `Config::meshDimension + 1`, where `Config::meshDimension` stands for the mesh dimension D:

```
template< typename Config, typename Device >
class StorageLayer< Config, Device, DimTag< Config::meshDimension + 1 > >
{};
```

In practice, the complete inheritance is more complicated than the scheme illustrated in Figures C1 and C2, since the `Mesh` class template has additional base classes which are omitted here for clarity. Individual storage layers also store more data in addition to the incidence matrices. For example, there are arrays storing *tags* (such as `BoundaryEntity`, `GhostEntity` or arbitrary user attributes) for each mesh entity. On the other hand, some data members which do not have to be parametrized by the entity dimension (e.g. the array of vertex coordinates) are placed directly in the `Mesh` class template. Finally, note that all data members in the hierarchy of storage layers are always present, even if they are unused according to the configuration. Disabling storage in the configuration skips only dynamic allocation and `static_assert` declarations guard access to the disabled data members.

Appendix D

Raviart–Thomas–Nédélec Basis Functions

The definition of the Raviart–Thomas–Nédélec space $\mathbf{RTN}_0(K)$ of the lowest order on the mesh element $K \in \mathcal{K}_h$ depends on the shape of the element. This chapter provides explicit definitions of the basis functions on common elements that satisfy the properties given by Equation (4.5). See [B6] for details.

D.1 Basis functions for simplex elements in $\mathbb{R}^D$

The simplex $K \in \mathcal{K}_h$ in $\mathbb{R}^D$ is the convex hull of $D+1$ affinely independent points $\boldsymbol{V}_0, \dots, \boldsymbol{V}_D \in \mathbb{R}^D$. We assume that the faces $E_0, \dots, E_D \in \mathcal{E}_K$ of the simplex are numbered such that for all $\ell \in \{0, \dots, D\}$, $\boldsymbol{V}_\ell$ is the opposite vertex of the face E_ℓ. Then the basis functions of the space $\mathbf{RTN}_0(K)$ are

$$\boldsymbol{\omega}_{K,E_\ell}(\boldsymbol{x}) = \frac{1}{D|K|_D}\,(\boldsymbol{x} - \boldsymbol{V}_\ell)\,, \tag{D.1}$$

for all $\ell \in \{0, \dots, D\}$ and $\boldsymbol{x} \in K$.

D.2 Basis functions for rectangles in $\mathbb{R}^2$

For the reference rectangular element $K = (0, h_x) \times (0, h_y)$ in $\mathbb{R}^2$ we choose the basis functions of the space $\mathbf{RTN}_0(K)$ of the form

$$\begin{aligned} \boldsymbol{\omega}_{K,E_1}(\boldsymbol{x}) &= \frac{1}{|K|_2}\begin{pmatrix} x - h_x \\ 0 \end{pmatrix}, & \boldsymbol{\omega}_{K,E_2}(\boldsymbol{x}) &= \frac{1}{|K|_2}\begin{pmatrix} x \\ 0 \end{pmatrix}, \\ \boldsymbol{\omega}_{K,E_3}(\boldsymbol{x}) &= \frac{1}{|K|_2}\begin{pmatrix} 0 \\ y - h_y \end{pmatrix}, & \boldsymbol{\omega}_{K,E_4}(\boldsymbol{x}) &= \frac{1}{|K|_2}\begin{pmatrix} 0 \\ y \end{pmatrix}, \end{aligned} \tag{D.2}$$

where x, y denote the components of the vector $\boldsymbol{x} \in K \subset \mathbb{R}^2$ and the symbols E_1, E_2, E_3, E_4 denote the left, right, bottom, and top edge of the element K, respectively.

D.3 Basis functions for cuboids in $\mathbb{R}^3$

For the reference cuboidal element $K = (0, h_x) \times (0, h_y) \times (0, h_z)$ in $\mathbb{R}^3$ we choose the basis functions of the space $\mathbf{RTN}_0(K)$ of the form

$$\begin{aligned}
\boldsymbol{\omega}_{K,E_1}(\boldsymbol{x}) &= \frac{1}{|K|_3}\begin{pmatrix} x - h_x \\ 0 \\ 0 \end{pmatrix}, & \boldsymbol{\omega}_{K,E_2}(\boldsymbol{x}) &= \frac{1}{|K|_3}\begin{pmatrix} x \\ 0 \\ 0 \end{pmatrix}, \\
\boldsymbol{\omega}_{K,E_3}(\boldsymbol{x}) &= \frac{1}{|K|_3}\begin{pmatrix} 0 \\ y - h_y \\ 0 \end{pmatrix}, & \boldsymbol{\omega}_{K,E_4}(\boldsymbol{x}) &= \frac{1}{|K|_3}\begin{pmatrix} 0 \\ y \\ 0 \end{pmatrix}, \\
\boldsymbol{\omega}_{K,E_5}(\boldsymbol{x}) &= \frac{1}{|K|_3}\begin{pmatrix} 0 \\ 0 \\ z - h_z \end{pmatrix}, & \boldsymbol{\omega}_{K,E_6}(\boldsymbol{x}) &= \frac{1}{|K|_3}\begin{pmatrix} 0 \\ 0 \\ z \end{pmatrix},
\end{aligned} \tag{D.3}$$

where x, y, z denote the components of the vector $\boldsymbol{x} \in K \subset \mathbb{R}^3$ and E_ℓ are the faces of the element K numbered as follows:

- E_1 and E_2 are faces orthogonal to the x-axis, the point $\boldsymbol{x} = \mathbf{0}$ lies on the face E_1,
- E_3 and E_4 are faces orthogonal to the y-axis, the point $\boldsymbol{x} = \mathbf{0}$ lies on the face E_3,
- E_5 and E_6 are faces orthogonal to the z-axis, the point $\boldsymbol{x} = \mathbf{0}$ lies on the face E_5.

Appendix E

Mass Matrices in MHFEM

This chapter completes the numerical scheme described in Section 4.2 with explicit calculations of the local mass matrices $\mathbf{b}_{i,j,K} = [b_{i,j,K,E,F}]_{E,F\in\mathcal{E}_K}$ defined for all $i, j \in \{1,\dots,n\}$ and $K \in \mathcal{K}_h$ in Section 4.2.4. For simplicity, we assume that the tensor $\mathbf{D}_{i,j} = D_{i,j}\mathbf{I}$ describes an isotropic medium and $D_{i,j,K}$ denotes the mean value of $D_{i,j}$ on the element K.

E.1 Mass matrix $\mathbf{b}_{i,j,K}$ for edges

Let us consider the reference element $K = (0, h_x) \subset \mathbb{R}^1$ and let the symbols E_1 and E_2 denote the left and right edge of the element K, respectively. Direct calculation of the defining integrals $B_{i,j,K,E,F} = D_{i,j,K}^{-1}\int_K \boldsymbol{\omega}_{K,E}^T\boldsymbol{\omega}_{K,F}$ for $E, F \in \mathcal{E}_K$ leads to

$$\mathbf{B}_{i,j,K} = \frac{h_x}{6}D_{i,j,K}^{-1}\begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix}. \tag{E.1}$$

The inverse matrix $\mathbf{b}_{i,j,K} = \mathbf{B}_{i,j,K}^{-1}$ is then

$$\mathbf{b}_{i,j,K} = \frac{2}{h_x}D_{i,j,K}\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}. \tag{E.2}$$

In practice, however, this choice of the coefficients $b_{i,j,K,E,F}$ may lead to spurious oscillations in simulations involving heterogeneous media, see [A72, B20, C37]. According to [A186], the problem is that the matrix of the final MHFEM system Equation (4.44) for rectangular and cuboidal elements is not an M-matrix. The solution is to use the *mass-lumping* technique where the matrix entries $B_{i,j,K,E,F}$ are calculated only approximately using numerical integration

$$\int_K \xi(\boldsymbol{x})\mathrm{d}\boldsymbol{x} \approx \frac{|K|}{\#\mathcal{V}_K}\sum_{\iota=1}^{\#\mathcal{V}_K}\xi(\boldsymbol{x}_\iota), \tag{E.3}$$

where ξ is the integrated scalar function and the integration points $\boldsymbol{x}_\iota$ are placed in the vertices of the element K with $\#\mathcal{V}_K$ being the number of the vertices. The resulting matrix $\mathbf{B}_{i,j,K}^{(\ell)}$ is diagonal and has the form

$$\mathbf{B}_{i,j,K} \approx \mathbf{B}_{i,j,K}^{(\ell)} = \frac{h_x}{2}D_{i,j,K}^{-1}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{E.4}$$

The matrix $\mathbf{b}_{i,j,K} = \mathbf{B}_{i,j,K}^{-1}$ is then approximated by the inverse of $\mathbf{B}_{i,j,K}^{(\ell)}$, i.e.

$$\mathbf{b}_{i,j,K} \approx \frac{2}{h_x}D_{i,j,K}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{E.5}$$

E.2 Mass matrix $\mathbf{b}_{i,j,K}$ for rectangles

Let us consider the reference element $K = (0, h_x) \times (0, h_y) \subset \mathbb{R}^2$ and consistently with Appendix D let the symbols E_1, E_2, E_3, E_4 denote the left, right, bottom, and top edge of the element K, respectively. Similarly to the previous section, direct calculation of the defining integrals leads to the matrix $\mathbf{B}_{i,j,K}$ in the form

$$\mathbf{B}_{i,j,K} = \frac{1}{6} D_{i,j,K}^{-1} \begin{pmatrix} 2\frac{h_x}{h_y} & -\frac{h_x}{h_y} & 0 & 0 \\ -\frac{h_x}{h_y} & 2\frac{h_x}{h_y} & 0 & 0 \\ 0 & 0 & 2\frac{h_y}{h_x} & -\frac{h_y}{h_x} \\ 0 & 0 & -\frac{h_y}{h_x} & 2\frac{h_y}{h_x} \end{pmatrix}. \tag{E.6}$$

The inverse matrix $\mathbf{b}_{i,j,K} = \mathbf{B}_{i,j,K}^{-1}$ is then

$$\mathbf{b}_{i,j,K} = 2 D_{i,j,K} \begin{pmatrix} 2\frac{h_y}{h_x} & \frac{h_y}{h_x} & 0 & 0 \\ \frac{h_y}{h_x} & 2\frac{h_y}{h_x} & 0 & 0 \\ 0 & 0 & 2\frac{h_x}{h_y} & \frac{h_x}{h_y} \\ 0 & 0 & \frac{h_x}{h_y} & 2\frac{h_x}{h_y} \end{pmatrix}. \tag{E.7}$$

Same as in $\mathbb{R}^1$, this choice of the coefficients $b_{i,j,K,E,F}$ for rectangular elements in $\mathbb{R}^2$ may cause spurious oscillations. Thus, we again use the mass-lumping technique and use the numerical integration from Equation (E.3) to calculate the matrix entries $B_{i,j,K,E,F}$. The resulting matrix is

$$\mathbf{B}_{i,j,K} \approx \mathbf{B}_{i,j,K}^{(\ell)} = \frac{1}{2} D_{i,j,K}^{-1} \begin{pmatrix} \frac{h_x}{h_y} & 0 & 0 & 0 \\ 0 & \frac{h_x}{h_y} & 0 & 0 \\ 0 & 0 & \frac{h_y}{h_x} & 0 \\ 0 & 0 & 0 & \frac{h_y}{h_x} \end{pmatrix}. \tag{E.8}$$

The matrix $\mathbf{b}_{i,j,K} = \mathbf{B}_{i,j,K}^{-1}$ is then approximated by the inverse of $\mathbf{B}_{i,j,K}^{(\ell)}$, i.e.

$$\mathbf{b}_{i,j,K} \approx 2 D_{i,j,K} \begin{pmatrix} \frac{h_y}{h_x} & 0 & 0 & 0 \\ 0 & \frac{h_y}{h_x} & 0 & 0 \\ 0 & 0 & \frac{h_x}{h_y} & 0 \\ 0 & 0 & 0 & \frac{h_x}{h_y} \end{pmatrix}. \tag{E.9}$$

E.3 Mass matrix $\mathbf{b}_{i,j,K}$ for cuboids

Let us consider the reference element $K = (0, h_x) \times (0, h_y) \times (0, h_z) \subset \mathbb{R}^3$ and let the faces $E_\iota \in \mathcal{E}_K$ be numbered same as in Appendix D.3. Similarly to the previous sections, direct calculation of the defining integrals leads to the matrix $\mathbf{B}_{i,j,K}$ in the form

$$\mathbf{B}_{i,j,K} = \frac{1}{6} D_{i,j,K}^{-1} \begin{pmatrix} 2\frac{h_x}{h_y h_z} & -\frac{h_x}{h_y h_z} & 0 & 0 & 0 & 0 \\ -\frac{h_x}{h_y h_z} & 2\frac{h_x}{h_y h_z} & 0 & 0 & 0 & 0 \\ 0 & 0 & 2\frac{h_y}{h_x h_z} & -\frac{h_y}{h_x h_z} & 0 & 0 \\ 0 & 0 & -\frac{h_y}{h_x h_z} & 2\frac{h_y}{h_x h_z} & 0 & 0 \\ 0 & 0 & 0 & 0 & 2\frac{h_z}{h_x h_y} & -\frac{h_z}{h_x h_y} \\ 0 & 0 & 0 & 0 & -\frac{h_z}{h_x h_y} & 2\frac{h_z}{h_x h_y} \end{pmatrix}. \tag{E.10}$$

The inverse matrix $\mathbf{b}_{i,j,K} = \mathbf{B}_{i,j,K}^{-1}$ is then

$$\mathbf{b}_{i,j,K} = 2 D_{i,j,K} \begin{pmatrix} 2\frac{h_y h_z}{h_x} & \frac{h_y h_z}{h_x} & 0 & 0 & 0 & 0 \\ \frac{h_y h_z}{h_x} & 2\frac{h_y h_z}{h_x} & 0 & 0 & 0 & 0 \\ 0 & 0 & 2\frac{h_x h_z}{h_y} & \frac{h_x h_z}{h_y} & 0 & 0 \\ 0 & 0 & \frac{h_x h_z}{h_y} & 2\frac{h_x h_z}{h_y} & 0 & 0 \\ 0 & 0 & 0 & 0 & 2\frac{h_x h_y}{h_z} & \frac{h_x h_y}{h_z} \\ 0 & 0 & 0 & 0 & \frac{h_x h_y}{h_z} & 2\frac{h_x h_y}{h_z} \end{pmatrix}. \tag{E.11}$$

Same as in $\mathbb{R}^1$, this choice of the coefficients $b_{i,j,K,E,F}$ for cuboidal elements in $\mathbb{R}^3$ may cause spurious oscillations. Thus, we again use the mass-lumping technique and use the numerical integration from Equation (E.3) to calculate the matrix entries $B_{i,j,K,E,F}$. The resulting matrix is

$$\mathbf{B}_{i,j,K} \approx \mathbf{B}_{i,j,K}^{(\ell)} = \frac{1}{2} D_{i,j,K}^{-1} \begin{pmatrix} \frac{h_x}{h_y h_z} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{h_x}{h_y h_z} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{h_y}{h_x h_z} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{h_y}{h_x h_z} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{h_z}{h_y h_z} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{h_z}{h_y h_z} \end{pmatrix}. \tag{E.12}$$

The matrix $\mathbf{b}_{i,j,K} = \mathbf{B}_{i,j,K}^{-1}$ is then approximated by the inverse of $\mathbf{B}_{i,j,K}^{(\ell)}$, i.e.

$$\mathbf{b}_{i,j,K} \approx 2 D_{i,j,K} \begin{pmatrix} \frac{h_y h_z}{h_x} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{h_y h_z}{h_x} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{h_x h_z}{h_y} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{h_x h_z}{h_y} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{h_x h_y}{h_z} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{h_x h_y}{h_z} \end{pmatrix}. \tag{E.13}$$

E.4 Mass matrix $\mathbf{b}_{i,j,K}$ for triangles

Let us consider the triangle $K = [\boldsymbol{V}_0, \boldsymbol{V}_1, \boldsymbol{V}_2]_\kappa \subset \mathbb{R}^2$ with edges $E_\iota \in \mathcal{E}_K$ numbered same as in Appendix D.1. To calculate the integrals $B_{i,j,K,E_e,E_f} = D^{-1}_{i,j,K} \int_K \boldsymbol{\omega}^T_{K,E_e} \boldsymbol{\omega}_{K,E_f}$ for $e, f \in \{0,1,2\}$, we use the barycentric coordinates $\eta_0, \eta_1 \in [0,1]$, $\eta_0 + \eta_1 \leq 1$. Denoting $\boldsymbol{P}_e = \boldsymbol{V}_e - \boldsymbol{V}_2$ for $e \in \{0,1,2\}$, each point $\boldsymbol{x} \in K$ can be expressed as

$$\boldsymbol{x} = \boldsymbol{V}_2 + \eta_0 \boldsymbol{P}_0 + \eta_1 \boldsymbol{P}_1. \tag{E.14}$$

The absolute value of the Jacobian determinant for this transformation is

$$\begin{aligned} |\mathcal{J}| &= \left| \det \begin{pmatrix} P_0^{(1)} & P_1^{(1)} \\ P_0^{(2)} & P_1^{(2)} \end{pmatrix} \right| = \left| P_0^{(1)} P_1^{(2)} - P_0^{(2)} P_1^{(1)} \right| \\ &= \left| (V_0^{(1)} - V_2^{(1)})(V_1^{(2)} - V_2^{(2)}) - (V_0^{(2)} - V_2^{(2)})(V_1^{(1)} - V_2^{(1)}) \right| = 2|K|_2. \end{aligned} \tag{E.15}$$

Using this transformation to calculate the integral leads to

$$\begin{aligned} \int_K \boldsymbol{\omega}^T_{K,E_e} \boldsymbol{\omega}_{K,E_f} \mathrm{d}\boldsymbol{x} &= \frac{1}{4|K|_2^2} \int_K (\boldsymbol{x} - \boldsymbol{V}_e)^T (\boldsymbol{x} - \boldsymbol{V}_f) \mathrm{d}\boldsymbol{x} \\ &= \frac{1}{2|K|_2} \int_0^1 \mathrm{d}\eta_1 \int_0^{1-\eta_1} \mathrm{d}\eta_0 (\eta_0 \boldsymbol{P}_0 + \eta_1 \boldsymbol{P}_1 - \boldsymbol{P}_e)^T (\eta_0 \boldsymbol{P}_0 + \eta_1 \boldsymbol{P}_1 - \boldsymbol{P}_f) \\ &= \frac{1}{2|K|_2} \int_0^1 \mathrm{d}\eta_1 \int_0^{1-\eta_1} \mathrm{d}\eta_0 \Big(\eta_0^2 \boldsymbol{P}_0^T \boldsymbol{P}_0 + \eta_1^2 \boldsymbol{P}_1^T \boldsymbol{P}_1 + \boldsymbol{P}_e^T \boldsymbol{P}_f \\ &\qquad + 2\eta_0\eta_1 \boldsymbol{P}_0^T \boldsymbol{P}_1 - \eta_0 \boldsymbol{P}_0^T (\boldsymbol{P}_e + \boldsymbol{P}_f) - \eta_1 \boldsymbol{P}_1^T (\boldsymbol{P}_e + \boldsymbol{P}_f) \Big) \\ &= \frac{1}{24|K|_2} \Big(\boldsymbol{P}_0^T \boldsymbol{P}_0 + \boldsymbol{P}_1^T \boldsymbol{P}_1 + \boldsymbol{P}_0^T \boldsymbol{P}_1 - 2(\boldsymbol{P}_0 + \boldsymbol{P}_1)^T (\boldsymbol{P}_e + \boldsymbol{P}_f) + 6 \boldsymbol{P}_e \boldsymbol{P}_f \Big). \end{aligned} \tag{E.16}$$

In general, all entries in the matrix $\mathbf{B}_{i,j,K}$ are non-zero. The inverse matrix $\mathbf{b}_{i,j,K}$ is computed using the LU factorization.

E.5 Mass matrix $\mathbf{b}_{i,j,K}$ for tetrahedra

Let us consider the tetrahedron $K = [\boldsymbol{V}_0, \boldsymbol{V}_1, \boldsymbol{V}_2, \boldsymbol{V}_3]_\kappa \subset \mathbb{R}^3$ and its faces $E_\iota \in \mathcal{E}_K$ numbered same as in Appendix D.1. To calculate the integrals $B_{i,j,K,E_e,E_f} = D_{i,j,K}^{-1} \int_K \boldsymbol{\omega}_{K,E_e}^T \boldsymbol{\omega}_{K,E_f}$ for $e, f \in \{0, \dots, 3\}$, we use the barycentric coordinates $\eta_0, \eta_1, \eta_2 \in [0, 1]$, $\eta_0 + \eta_1 + \eta_2 \leq 1$. Denoting $\boldsymbol{P}_e = \boldsymbol{V}_e - \boldsymbol{V}_3$ for $e \in \{0, \dots, 3\}$, each point $\boldsymbol{x} \in K$ can be expressed as

$$\boldsymbol{x} = \boldsymbol{V}_3 + \eta_0 \boldsymbol{P}_0 + \eta_1 \boldsymbol{P}_1 + \eta_2 \boldsymbol{P}_2. \tag{E.17}$$

The absolute value of the Jacobian determinant for this transformation is

$$|\mathcal{J}| = \left| \det \begin{pmatrix} P_0^{(1)} & P_1^{(1)} & P_2^{(1)} \\ P_0^{(2)} & P_1^{(2)} & P_2^{(2)} \\ P_0^{(3)} & P_1^{(3)} & P_2^{(3)} \end{pmatrix} \right| = \left| \boldsymbol{P}_0 \cdot (\boldsymbol{P}_1 \times \boldsymbol{P}_2) \right| = 6|K|_3. \tag{E.18}$$

Using this transformation to calculate the integral leads to

$$\begin{aligned}
\int_K \boldsymbol{\omega}_{K,E_e}^T \boldsymbol{\omega}_{K,E_f} \mathrm{d}\boldsymbol{x} &= \frac{1}{9|K|_3^2} \int_K (\boldsymbol{x} - \boldsymbol{V}_e)^T (\boldsymbol{x} - \boldsymbol{V}_f) \mathrm{d}\boldsymbol{x} \\
&= \frac{2}{3|K|_3} \int_0^1 \mathrm{d}\eta_2 \int_0^{1-\eta_2} \mathrm{d}\eta_1 \int_0^{1-\eta_1-\eta_2} \mathrm{d}\eta_0 \left(\sum_{k=0}^{2} \eta_k \boldsymbol{P}_k - \boldsymbol{P}_e \right)^T \left(\sum_{\ell=0}^{2} \eta_\ell \boldsymbol{P}_\ell - \boldsymbol{P}_f \right) \\
&= \frac{2}{3|K|_3} \int_0^1 \mathrm{d}\eta_2 \int_0^{1-\eta_2} \mathrm{d}\eta_1 \int_0^{1-\eta_1-\eta_2} \mathrm{d}\eta_0 \left(\sum_{k=0}^{2} \left(\eta_k^2 \boldsymbol{P}_k^T \boldsymbol{P}_k - \eta_k \boldsymbol{P}_k^T (\boldsymbol{P}_e + \boldsymbol{P}_f) \right) \right. \\
&\qquad \left. + \sum_{\substack{k,\ell=0 \\ k \neq \ell}}^{2} \eta_k \eta_\ell \boldsymbol{P}_k^T \boldsymbol{P}_\ell + \boldsymbol{P}_e^T \boldsymbol{P}_f \right) \\
&= \frac{1}{180|K|_3} \left(2\boldsymbol{P}_0^T \boldsymbol{P}_0 + 2\boldsymbol{P}_1^T \boldsymbol{P}_1 + 2\boldsymbol{P}_2^T \boldsymbol{P}_2 + 2\boldsymbol{P}_0^T \boldsymbol{P}_1 + 2\boldsymbol{P}_0^T \boldsymbol{P}_2 + 2\boldsymbol{P}_1^T \boldsymbol{P}_2 \right. \\
&\qquad \left. - 5(\boldsymbol{P}_0 + \boldsymbol{P}_1 + \boldsymbol{P}_2)^T (\boldsymbol{P}_e + \boldsymbol{P}_f) + 20 \boldsymbol{P}_e \boldsymbol{P}_f \right).
\end{aligned} \tag{E.19}$$

In general, all entries in the matrix $\mathbf{B}_{i,j,K}$ are non-zero. The inverse matrix $\mathbf{b}_{i,j,K}$ is computed using the LU factorization.

Appendix F

Velocity Sets for LBM

A Python script solving the system of equations that the given velocity set must satisfy. The reference mentioned in the code is [B21, Section 3.4.7.2].

[online version]

```python
#! /usr/bin/env python3
# Reference: [1] Timm Krüger - The Lattice Boltzmann Method, Section 3.4.7.2, Eq. (3.60)
import itertools
from sympy import *
init_printing()

# input parameters
D = 3    # space dimension
Q = 27   # number of discrete velocities
#cs = 1/Rational(2)   # speed of sound (D3Q7)
cs = 1/sqrt(3)        # speed of sound (D1Q3, D2Q5, D2Q9, D3Q7*, D3Q15, D3Q19, D3Q27)

# all discrete velocity vectors
c_1_3 = Matrix([[0, 1, -1]])
c_2_9 = Matrix([
    [0, 1, 0, -1, 0,   1, 1,-1,-1],
    [0, 0, 1, 0, -1,   1,-1, 1,-1],
])
c_3_27 = Matrix([
    [0, 1, 0, 0, -1, 0, 0,   0, 0, 0, 0, 1, 1,-1,-1, 1, 1,-1,-1,   1, 1, 1, 1,-1,-1,-1,-1],
    [0, 0, 1, 0, 0, -1, 0,   1, 1,-1,-1, 1,-1, 1,-1, 0, 0, 0, 0,   1, 1,-1,-1, 1, 1,-1,-1],
    [0, 0, 0, 1, 0, 0, -1,   1,-1, 1,-1, 0, 0, 0, 0, 1,-1, 1,-1,   1,-1, 1,-1, 1,-1, 1,-1],
])

# select the discrete velocity vectors for D and Q
if D == 1:
    c = c_1_3
elif D == 2:
    c = c_2_9.extract(range(D), range(Q))
elif D == 3 and Q == 15:
    c = c_3_27.extract(range(D), [i for i in range(27) if i < 7 or 19 <= i])
elif D == 3:
    c = c_3_27.extract(range(D), range(Q))
else:
    raise NotImplementedError
assert sum(c) == 0
print(f"Matrix of discrete velocity vectors for D{D}Q{Q}")
pprint(c)

def sym_vector(dim, name):
    syms = []
    for i in range(dim):
        syms.append(symbols(f"{name}_{i}"))
    return Matrix(syms)

```

```python
def delta(a, b):
    return 1 if a == b else 0

# solve weights according to [1]
w = sym_vector(Q, "w")
eqs = []
# eq 1
eqs.append(Eq(sum(w), 1))
# eq 2
for d1 in range(D):
    eqs.append(Eq(w.dot(c.row(d1)), 0))
# eq 3
for d1, d2 in itertools.product(range(D), repeat=2):
    rhs = cs ** 2 if d1 == d2 else 0
    s = Add(0)
    for i in range(Q):
        s += w[i] * c[d1, i] * c[d2, i]
    # avoid adding identical equation "0=0" (it would add a boolean constant instead of an expression)
    if s != 0:
        eqs.append(Eq(s, rhs))
# eq 4
for d1, d2, d3 in itertools.product(range(D), repeat=3):
    s = Add(0)
    for i in range(Q):
        s += w[i] * c[d1, i] * c[d2, i] * c[d3, i]
    # avoid adding identical equation "0=0" (it would add a boolean constant instead of an expression)
    if s != 0:
        eqs.append(Eq(s, 0))
# avoid overdetermined system
if not (D == 3 and Q == 7 and cs == 1/Rational(2)):
    # eq 5
    for d1, d2, d3, d4 in itertools.product(range(D), repeat=4):
        rhs = cs ** 4 * (delta(d1,d2) * delta(d3,d4) + delta(d1,d3) * delta(d2,d4) \
                + delta(d1,d4) * delta(d2,d3))
        s = Add(0)
        for i in range(Q):
            s += w[i] * c[d1, i] * c[d2, i] * c[d3, i] * c[d4, i]
        # avoid adding identical equation "0=0" (it would add a boolean constant instead of an expression)
        if s != 0:
            eqs.append(Eq(s, rhs))
    # eq 6
    for d1, d2, d3, d4, d5 in itertools.product(range(D), repeat=5):
        s = Add(0)
        for i in range(Q):
            s += w[i] * c[d1, i] * c[d2, i] * c[d3, i] * c[d4, i] * c[d5, i]
        # avoid adding identical equation "0=0" (it would add a boolean constant instead of an expression)
        if s != 0:
            eqs.append(Eq(s, 0))

print("Total number of equations:", len(eqs))
for i, eq in enumerate(eqs):
    pprint(f"{i}-th equation: {eq}")

solution = linsolve(eqs, [wi for wi in w])
if not solution:
    raise Exception("linsolve did not return a solution")
solution = Matrix(list(solution)[0])

print("solution:")
pprint(Eq(w, solution))

# D3Q27 is underdetermined (see [1])
if D == 3 and Q == 27:
    w26 = 1/Rational(216)
    print(f"assuming {w[-1]}={w26}:")
    pprint(Eq(w, solution.subs({w[-1]: w26})))
```

Appendix G

Synthetic Turbulence Generator

The procedure for generating isotropic synthetic turbulence is based on [O6, C18, A54, A55]. It can be used to generate velocity fluctuations for an initial condition $\boldsymbol{v}(\boldsymbol{x}, 0)$ for $\boldsymbol{x} \in \Omega$, as well as for an inflow boundary condition $\boldsymbol{v}_{\text{in}}(\boldsymbol{x}, t)$ for $\boldsymbol{x} \in \Gamma_{\text{in}} \subset \partial\Omega$, $t \geq 0$. The velocity field $\boldsymbol{v}(\boldsymbol{x}, t)$ is first decomposed as

$$\boldsymbol{v}(\boldsymbol{x}, t) = \overline{\boldsymbol{v}}(\boldsymbol{x}) + \boldsymbol{v}'(\boldsymbol{x}, t), \tag{G.1}$$

where $\overline{\boldsymbol{v}}(\boldsymbol{x})$ is the mean (time-averaged) field and $\boldsymbol{v}'(\boldsymbol{x}, t)$ is the velocity fluctuation. The mean field $\overline{\boldsymbol{v}}$ can be set as desired for the initial or boundary condition and the fluctuation $\boldsymbol{v}'$ must be generated. The algorithm described below is implemented in the C++ language with CUDA acceleration and available as part of the Template Numerical Library that was introduced in Section 1.3.

G.1 Computing the fluctuations

Turbulent fluctuations consist of a wide range of scales and their analysis is based on using Fourier series. A general form of a fluctuating velocity field $\boldsymbol{v}'$ at a fixed time level with zero average ($\overline{\boldsymbol{v}'} = 0$) can be expressed as [O6, Eq. (27.3)]

$$\boldsymbol{v}'(\boldsymbol{x}) = 2 \sum_{n=1}^{N_{\text{modes}}} \hat{v}^n \cos(\boldsymbol{\kappa}^n \cdot \boldsymbol{x} + \psi^n) \boldsymbol{\sigma}^n, \tag{G.2}$$

where $\hat{v}^n$ and ψ^n are the amplitude and phase of the n-th mode, $\boldsymbol{\sigma}^n$ is a unit vector that determines the direction of the n-th mode, and $\boldsymbol{\kappa}^n$ is the wave-number vector of the n-th mode. Note that only the first N_{modes} terms of the full series are considered in the generator.

The wave-number vector $\boldsymbol{\kappa}^n$ is written as

$$\kappa_x^n = \rho^n \sin(\theta^n) \cos(\phi^n), \tag{G.3}$$
$$\kappa_y^n = \rho^n \sin(\theta^n) \sin(\phi^n), \tag{G.4}$$
$$\kappa_z^n = \rho^n \cos(\theta^n), \tag{G.5}$$

where θ^n and ϕ^n are randomly selected angles and ρ^n is the magnitude of the n-th wave number. In order to have $\nabla \cdot \boldsymbol{v}' = 0$, the direction vector $\boldsymbol{\sigma}^n$ must be orthogonal to the wave-number vector $\boldsymbol{\kappa}^n$ [O6]. Hence, $\boldsymbol{\sigma}^n$ can be written as

$$\sigma_x^n = \cos(\phi^n) \cos(\theta^n) \cos(\alpha^n) - \sin(\phi^n) \sin(\alpha^n), \tag{G.6}$$
$$\sigma_y^n = \sin(\phi^n) \cos(\theta^n) \cos(\alpha^n) + \cos(\phi^n) \sin(\alpha^n), \tag{G.7}$$
$$\sigma_z^n = -\sin(\theta^n) \cos(\alpha^n), \tag{G.8}$$

where α^n is a randomly selected angle that gives the direction of $\boldsymbol{\sigma}^n$ in the plane orthogonal to $\boldsymbol{\kappa}^n$. The phase ψ^n in Equation (G.2) is selected randomly as well.

The magnitudes ρ^n of the discrete wave numbers in Equation (G.3) are selected by sampling an interval $[\kappa_{\text{min}}, \kappa_{\text{max}}]$ with N_{modes} points, i.e.

$$\rho^n = \kappa_{\text{min}} + \left(n - \frac{1}{2}\right)\Delta\kappa \tag{G.9}$$

for $n \in \{1, \ldots, N_{\text{modes}}\}$, where $\Delta\kappa = (\kappa_{\text{max}} - \kappa_{\text{min}})/N_{\text{modes}}$. The highest wave number $\kappa_{\text{max}} = 2\pi/\delta_x$ is set based on the grid spacing δ_x and the smallest wave number is set as $\kappa_{\text{min}} = \kappa_e/p$, where the factor $p = 5$ is chosen as suggested in [O6] and κ_e is specified below in Equation (G.10d) based on the turbulence model.

The modified von Kármán spectrum [O6, A54] is used to model the turbulent energy spectrum. The amplitude $\hat{v}^n$ of the n-th mode in Equation (G.2) is obtained from

$$\hat{v}^n = v_{\text{RMS}}\sqrt{E(\rho^n)\Delta k}, \tag{G.10a}$$

$$E(\kappa) = \frac{c_E}{\kappa_e}\frac{\left(\frac{\kappa}{\kappa_e}\right)^4}{\left[1 + \left(\frac{\kappa}{\kappa_e}\right)^2\right]^{17/6}}\exp\left(-2\left(\frac{\kappa}{\kappa_\eta}\right)^2\right), \tag{G.10b}$$

$$c_E = \frac{4}{\sqrt{\pi}}\frac{\Gamma\left(\frac{17}{6}\right)}{\Gamma\left(\frac{1}{3}\right)} \approx 1.452762113, \tag{G.10c}$$

$$\kappa_e = \frac{9\pi c_E}{55\mathcal{L}_{\text{int}}}, \qquad \kappa_\eta = \left(\frac{\epsilon}{\nu^3}\right)^{1/4}, \tag{G.10d}$$

where $v_{\text{RMS}} = \sqrt{\frac{2}{3}k}$ [m s^{-1}] is the turbulent velocity scale (root mean square), k [m^2 s^{-2}] is the turbulent kinetic energy, $\mathcal{L}_{\text{int}}$ [m] is the turbulent integral length scale, $\epsilon = \frac{k^{3/2}}{\mathcal{L}_{\text{int}}}$ [m^2 s^{-3}] is the dissipation rate of the turbulent kinetic energy, and ν [m^2 s^{-1}] is the kinematic viscosity of the fluid.

The input parameters of the turbulence generator are the quantities N_{modes}, k, and $\mathcal{L}_{\text{int}}$. For channel flow applications, $\mathcal{L}_{\text{int}}$ may be estimated as a fraction (e.g., 1/10 to 1/2) of the expected boundary layer height [O6].

G.2 Introducing time correlation

To generate velocity fluctuations at multiple discrete time levels $t_n = n\delta_t$, where n is an integer denoting the time level and δ_t is the time step used in a numerical simulation, additional computations are needed. Firstly, independent realizations of random fluctuations $\hat{\boldsymbol{v}}'$ are generated for each time level using the procedure described above. Then, time correlation between the realizations is introduced using an asymmetric time filter

$$(\boldsymbol{v}')^n = a(\boldsymbol{v}')^{n-1} + b(\hat{\boldsymbol{v}}')^n, \tag{G.11}$$

where $\boldsymbol{v}'$ denotes the time-correlated field, $\hat{\boldsymbol{v}}'$ denotes the time-independent field, superscripts denote the time levels and the coefficients are chosen as $a = \exp(-\Delta t/\mathcal{T}_{\text{int}})$ and $b = \sqrt{1 - a^2}$, where $\mathcal{T}_{\text{int}} = \mathcal{L}_{\text{int}}/V_b$ is set using the Taylor's hypothesis based on the desired bulk velocity $V_b \approx |\overline{\boldsymbol{v}}|$. The time filter ensures that $\mathcal{T}_{\text{int}}$ corresponds to the turbulent integral time scale and that the variance of the generated fluctuations is preserved [O6].

G.3 Example

Figure G1 shows an example of the synthetic fluctuating velocity field v'_x [$\mathrm{m\,s^{-1}}$] computed on a unit square discretized by 500×500 points. The parameters used in the generator are: grid spacing $\delta_x = 1/499\,\mathrm{m}$, integral length scale $\mathcal{L}_{\mathrm{int}} = 0.01\,\mathrm{m}$, number of discrete modes $N_{\mathrm{modes}} = 3000$, turbulent kinetic energy $k = 10^{-2}\,\mathrm{m^2\,s^{-2}}$, and kinematic viscosity $\nu = 1.5 \times 10^{-5}\,\mathrm{m^2\,s^{-1}}$.

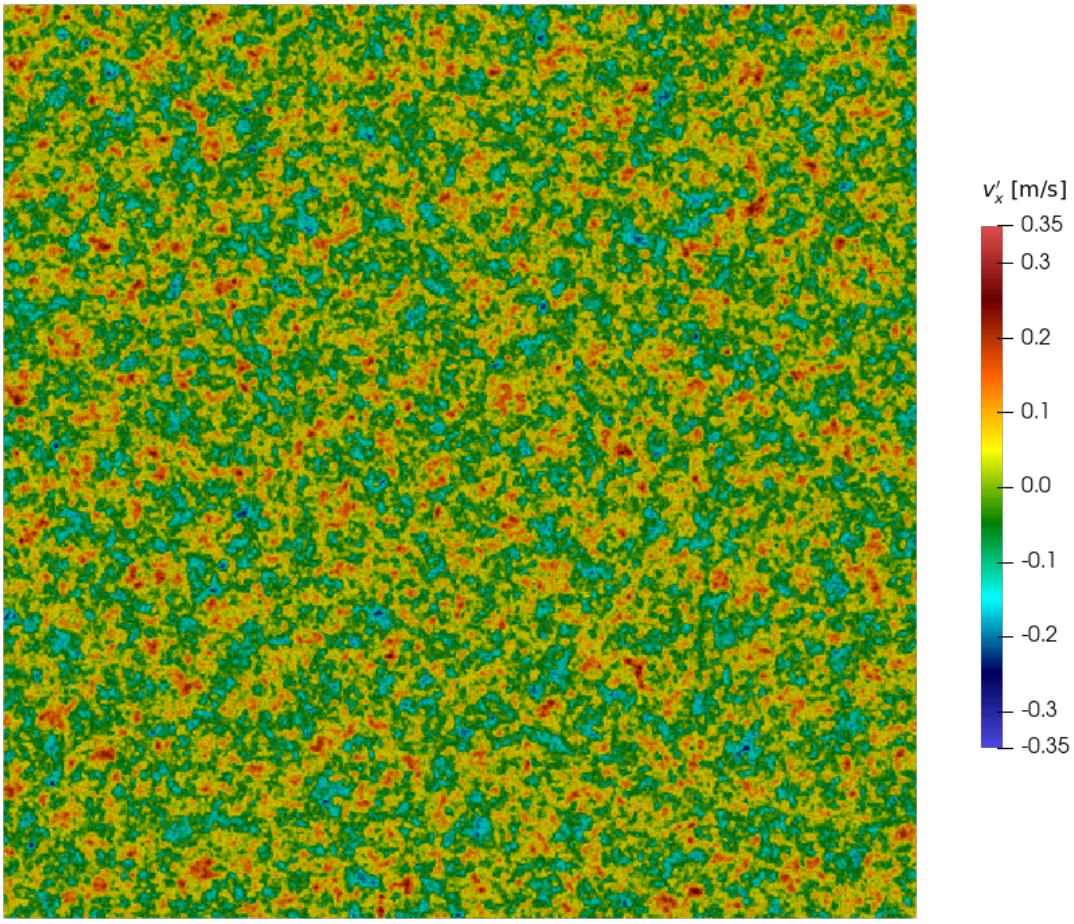


Figure G1: First component of synthetic fluctuating velocity field (v'_x [$\mathrm{m\,s^{-1}}$]) generated on a unit square discretized by 500×500 points.

Bibliography

Articles

Articles in scientific journals

[A1] R. Agelek, M. Anderson, W. Bangerth, and W. L. Barth, *On orienting edges of unstructured two-and three-dimensional meshes*, ACM Transactions on Mathematical Software 44.1 (2017), pages 1–22, DOI: 10.1145/3061708.

[A2] E. Agullo, L. Giraud, A. Guermouche, A. Haidar, and J. Roman, *Parallel algebraic domain decomposition solver for the solution of augmented systems*, Advances in Engineering Software 60-61 (2013), pages 23–30, ISSN: 0965-9978, DOI: 10.1016/j.advengsoft.2012.07.004.

[A3] A. Amritkar, E. de Sturler, K. Świrydowicz, D. Tafti, and K. Ahuja, *Recycling Krylov subspaces for CFD applications and a new hybrid recycling solver*, Journal of Computational Physics 303 (2015), pages 222–237, DOI: 10.1016/j.jcp.2015.09.040.

[A4] H. Anzt, E. Chow, and J. Dongarra, *ParILUT—a new parallel threshold ILU factorization*, SIAM Journal on Scientific Computing 40.4 (2018), pages C503–C519, DOI: 10.1137/16M1079506.

[A5] H. Anzt, E. Chow, J. Saak, and J. Dongarra, *Updating incomplete factorization preconditioners for model order reduction*, Numerical Algorithms 73.3 (2016), pages 611–630, ISSN: 1572-9265, DOI: 10.1007/s11075-016-0110-2.

[A6] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y. M. Tsai, and E. S. Quintana-Ortí, *Ginkgo: A modern linear operator algebra framework for high performance computing*, ACM Transactions on Mathematical Software 48.1 (Feb. 2022), pages 1–33, ISSN: 0098-3500, DOI: 10.1145/3480935.

[A7] H. Anzt, T. K. Huckle, J. Bräckle, and J. Dongarra, *Incomplete sparse approximate inverses for parallel preconditioning*, Parallel Computing 71 (2018), pages 1–22, ISSN: 0167-8191, DOI: 10.1016/j.parco.2017.10.003.

[A8] D. N. Arnold, F. Brezzi, B. Cockburn, and L. D. Marini, *Unified analysis of discontinuous Galerkin methods for elliptic problems*, SIAM Journal on Numerical Analysis 39.5 (2002), pages 1749–1779, DOI: 10.1137/S0036142901384162.

[A9] S. F. Ashby, T. A. Manteuffel, and J. S. Otto, *A comparison of adaptive Chebyshev and least squares polynomial preconditioning for Hermitian positive definite linear systems*, SIAM Journal on Scientific and Statistical Computing 13.1 (1992), pages 1–29, DOI: 10.1137/0913001.

[A10] A. H. Askar, T. H. Illangasekare, A. C. Trautz, J. Solovský, Y. Zhang, and R. Fučík, *Exploring the impacts of source condition uncertainties on far-field brine leakage plume predictions in geologic storage of CO2: integrating intermediate-scale laboratory testing with numerical modeling*, Water Resources Research 57.9 (2021), e2021WR029679, DOI: 10.1029/2021WR029679.

[A11] O. Axelsson and P. S. Vassilevski, *Algebraic multilevel preconditioning methods, I*, Numerische Mathematik 56.2 (1989), pages 157–177, DOI: 10.1007/BF01409783.

[A12] O. Axelsson and P. S. Vassilevski, *Algebraic multilevel preconditioning methods, II*, SIAM Journal on Numerical Analysis 27.6 (1990), pages 1569–1590, DOI: 10.1137/0727092.

[A13] Z. Bai, D. Hu, and L. Reichel, *A Newton basis GMRES implementation*, IMA Journal of Numerical Analysis 14.4 (1994), pages 563–581, DOI: 10.1093/imanum/14.4.563.

[A14] A. H. Baker, R. D. Falgout, and U. M. Yang, *An assumed partition algorithm for determining processor inter-communication*, Parallel Computing 32.5 (2006), pages 394–414, ISSN: 0167-8191, DOI: 10.1016/j.parco.2006.06.009.

[A15] A. H. Baker, E. R. Jessup, and T. V. Kolev, *A simple strategy for varying the restart parameter in GMRES(m)*, Journal of computational and applied mathematics 230.2 (2009), pages 751–761, DOI: 10.1016/j.cam.2009.01.009.

[A16] A. H. Baker, E. R. Jessup, and T. Manteuffel, *A technique for accelerating the convergence of restarted GMRES*, SIAM Journal on Matrix Analysis and Applications 26.4 (2005), pages 962–984, DOI: 10.1137/S0895479803422014.

[A17] W. Bangerth, R. Hartmann, and G. Kanschat, *deal.II – a general purpose object oriented finite element library*, ACM Transactions on Mathematical Software 33.4 (2007), pages 24/1–24/27, DOI: 10.1145/1268776.1268779.

[A18] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander, *A generic grid interface for parallel and adaptive scientific computing. Part II: implementation and tests in DUNE*, Computing 82.2-3 (2008), pages 121–138, DOI: 10.1007/s00607-008-0004-9.

[A19] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander, *A generic grid interface for parallel and adaptive scientific computing. Part I: abstract framework*, Computing 82.2-3 (2008), pages 103–119, DOI: 10.1007/s00607-008-0003-x.

[A20] P. Bauer, V. Klement, T. Oberhuber, and V. Žabka, *Implementation of the Vanka-type multigrid solver for the finite element approximation of the Navier–Stokes equations on GPU*, Computer Physics Communications 200 (2016), pages 50–56, DOI: 10.1016/j.cpc.2015.10.021.

[A21] N. Bell, S. Dalton, and L. N. Olson, *Exposing fine-grained parallelism in algebraic multigrid methods*, SIAM Journal on Scientific Computing 34.4 (2012), pages C123–C152, DOI: 10.1137/110838844.

[A22] N. Bell, L. N. Olson, and J. Schroder, *PyAMG: Algebraic multigrid solvers in Python*, Journal of Open Source Software 7.72 (2022), pages 1–4, DOI: 10.21105/joss.04142.

[A23] M. Beneš, P. Eichler, R. Fučík, J. Hrdlička, J. Klinkovský, M. Kolář, T. Smejkal, P. Skopec, J. Solovský, P. Strachota, R. Straka, and A. Žák, *Experimental and numerical investigation of air flow through the distributor plate in a laboratory-scale model of a bubbling fluidized bed boiler*, Japan Journal of Industrial and Applied Mathematics 39.3 (2022), pages 943–958, ISSN: 1868-937X, DOI: 10.1007/s13160-022-00518-x.

[A24] M. Benzi, *Preconditioning techniques for large linear systems: a survey*, Journal of Computational Physics 182.2 (2002), pages 418–477, ISSN: 0021-9991, DOI: 10.1006/jcph.2002.7176.

[A25] M. Benzi, C. D. Meyer, and M. Tůma, *A sparse approximate inverse preconditioner for the conjugate gradient method*, SIAM Journal on Scientific Computing 17.5 (1996), pages 1135–1149, DOI: 10.1137/S1064827594271421.

[A26] M. Benzi and M. Tůma, *A sparse approximate inverse preconditioner for nonsymmetric linear systems*, SIAM Journal on Scientific Computing 19.3 (1998), pages 968–994, DOI: 10.1137/S1064827595294691.

[A27] M. Bernaschi, M. Carrozzo, A. Franceschini, and C. Janna, *A dynamic pattern factored sparse approximate inverse preconditioner on graphics processing units*, SIAM Journal on Scientific Computing 41.3 (2019), pages C139–C160, DOI: 10.1137/18M1197461.

[A28] D. Bertaccini and S. Filippone, *Sparse approximate inverse preconditioners on high performance GPU platforms*, Computers & Mathematics with Applications 71.3 (2016), pages 693–711, ISSN: 0898-1221, DOI: 10.1016/j.camwa.2015.12.008.

[A29] M. Bollhöfer, *A robust ILU with pivoting based on monitoring the growth of the inverse factors*, Linear Algebra and its Applications 338.1-3 (2001), pages 201–218, DOI: 10.1016/S0024-3795(01)00385-8.

[A30] M. Bollhöfer and Y. Saad, *A factored approximate inverse preconditioner with pivoting*, SIAM Journal on Matrix Analysis and Applications 23.3 (2002), pages 692–705, DOI: 10.1137/S0895479800372122.

[A31] M. Bollhöfer and Y. Saad, *On the relations between ILUs and factored approximate inverses*, SIAM Journal on Matrix Analysis and Applications 24.1 (2002), pages 219–237, DOI: 10.1137/S0895479800372110.

[A32] E. F. F. Botta and F. W. Wubs, *Matrix renumbering ILU: An effective algebraic multilevel ILU preconditioner for sparse matrices*, SIAM Journal on Matrix Analysis and Applications 20.4 (1999), pages 1007–1026, DOI: 10.1137/S0895479897319301.

[A33] B. Braden, *The surveyor's area formula*, The College Mathematics Journal 17.4 (Jan. 1986), pages 326–337, DOI: 10.1080/07468342.1986.11972974.

[A34] R. A. Bridges, N. Imam, and T. M. Mintz, *Understanding GPU power: A survey of profiling, modeling, and simulation methods*, ACM Computing Surveys 49.3 (Sept. 2016), ISSN: 0360-0300, DOI: 10.1145/2962131.

[A35] R. Bridson and C. Greif, *A multipreconditioned conjugate gradient algorithm*, SIAM Journal on Matrix Analysis and Applications 27.4 (2006), pages 1056–1068, DOI: 10.1137/040620047.

[A36] R. Bridson and W.-P. Tang, *Refining an approximate inverse*, Journal of Computational and Applied Mathematics 123.1 (2000), pages 293–306, ISSN: 0377-0427, DOI: 10.1016/S0377-0427(00)00399-X.

[A37] R. H. Brooks and A. T. Corey, *Hydraulic properties of porous media and their relation to drainage design*, Transactions of the ASAE 7.1 (1964), pages 26–28, DOI: 10.13031/2013.40684.

[A38] F. Brunner, F. A. Radu, and P. Knabner, *Analysis of an upwind-mixed hybrid finite element method for transport problems*, SIAM Journal on Numerical Analysis 52.1 (Jan. 2014), pages 83–102, DOI: 10.1137/130908191.

[A39] N. T. Burdine, *Relative permeability calculations from pore size distribution data*, Journal of Petroleum Technology 5.03 (1953), pages 71–78, DOI: 10.2118/225-G.

[A40] N. T. Burdine, L. S. Gournay, and P. P. Reichertz, *Pore size distribution of petroleum reservoir rocks*, Journal of Petroleum Technology 2.07 (1950), pages 195–204, DOI: 10.2118/950195-G.

[A41] C. Calgaro, J.-P. Chehab, and Y. Saad, *Incremental incomplete LU factorizations with applications*, Numerical Linear Algebra with Applications 17.5 (2010), pages 811–837, DOI: 10.1002/nla.756.

[A42] M. J. Castro, S. Ortega, M. de la Asunción, and J. M. Mantas, *On the benefits of using GPUs to simulate shallow flows with finite volume schemes*, SeMA Journal 50.1 (2010), pages 27–44, ISSN: 2254-3902, DOI: 10.1007/BF03322540.

[A43] M. J. Castro, S. Ortega, M. de la Asunción, J. M. Mantas, and J. M. Gallardo, *GPU computing for shallow water flow simulation based on finite volume schemes*, Comptes Rendus Mécanique 339.2 (2011), pages 165–184, ISSN: 1631-0721, DOI: 10.1016/j.crme.2010.12.004.

[A44] C. Cecka, A. J. Lew, and E. Darve, *Assembly of finite element methods on graphics processors*, International Journal for Numerical Methods in Engineering 85.5 (Feb. 2011), pages 640–669, ISSN: 0029-5981, DOI: 10.1002/nme.2989.

[A45] T. F. Chan, G. H. Golub, and R. J. LeVeque, *Algorithms for computing the sample variance: Analysis and recommendations*, The American Statistician 37.3 (1983), pages 242–247, DOI: 10.1080/00031305.1983.10483115.

[A46] A. Chapman and Y. Saad, *Deflated and augmented Krylov subspace techniques*, Numerical linear algebra with applications 4.1 (1997), pages 43–66, DOI: 10.1002/(SICI)1099-1506(199701/02)4:1<43::AID-NLA99>3.0.CO;2-Z.

[A47] E. Chow, *A priori sparsity patterns for parallel sparse approximate inverse preconditioners*, SIAM Journal on Scientific Computing 21.5 (2000), pages 1804–1822, DOI: 10.1137/S106482759833913X.

[A48] E. Chow and A. Patel, *Fine-grained parallel incomplete LU factorization*, SIAM Journal on Scientific Computing 37.2 (2015), pages C169–C193, DOI: 10.1137/140968896.

[A49] E. Coleman and M. Sosonkina, *Self-stabilizing fine-grained parallel incomplete LU factorization*, Sustainable Computing: Informatics and Systems 19 (2018), pages 291–304, DOI: 10.1016/j.suscom.2018.01.003.

[A50] S. Cools and W. Vanroose, *The communication-hiding pipelined BiCGStab method for the parallel solution of large unsymmetric linear systems*, Parallel Computing 65 (2017), pages 1–20, DOI: 10.1016/j.parco.2017.04.005.

[A51] L. Dagum and R. Menon, *OpenMP: an industry standard API for shared-memory programming*, IEEE Computational Science and Engineering 5.1 (1998), pages 46–55, DOI: 10.1109/99.660313.

[A52] D. Dapelo, S. Simonis, M. J. Krause, and J. Bridgeman, *Lattice-Boltzmann coupled models for advection-diffusion flow on a wide range of Péclet numbers*, Journal of Computational Science 51 (2021), page 101363, ISSN: 1877-7503, DOI: 10.1016/j.jocs.2021.101363.

[A53] C. Dapogny, C. Dobrzynski, and P. Frey, *Three-dimensional adaptive domain remeshing, implicit domain meshing, and applications to free and moving boundary problems*, Journal of computational physics 262 (2014), pages 358–378, DOI: 10.1016/j.jcp.2014.01.005.

[A54] L. Davidson, *Using isotropic synthetic fluctuations as inlet boundary conditions for unsteady simulations*, Advances and Applications in Fluid Mechanics 1 (Jan. 2007), pages 1–35.

[A55] L. Davidson and M. Billson, *Hybrid LES-RANS using synthesized turbulent fluctuations for forcing in the interface region*, International journal of heat and fluid flow 27.6 (2006), pages 1028–1042, DOI: 10.1016/j.ijheatfluidflow.2006.02.025.

[A56] T. A. Davis, *Algorithm 832: UMFPACK v4.3—an unsymmetric-pattern multifrontal method*, ACM Transactions on Mathematical Softwtware 30.2 (June 2004), pages 196–199, ISSN: 0098-3500, DOI: 10.1145/992200.992206.

[A57] M. M. Dehnavi, D. M. Fernández, J.-L. Gaudiot, and D. D. Giannacopoulos, *Parallel sparse approximate inverse preconditioning on graphic processing units*, IEEE Transactions on Parallel and Distributed Systems 24.9 (2013), pages 1852–1862, DOI: 10.1109/TPDS.2012.286.

[A58] D. Demidov, *AMGCL: An efficient, flexible, and extensible algebraic multigrid implementation*, Lobachevskii Journal of Mathematics 40.5 (2019), pages 535–546, ISSN: 1818-9962, DOI: 10.1134/S1995080219050056.

[A59] D. Demidov, *AMGCL: A C++ library for efficient solution of large sparse linear systems*, Software Impacts 6.100037 (Nov. 2020), ISSN: 2665-9638, DOI: 10.1016/j.simpa.2020.100037.

[A60] B. Desjardins and E. Grenier, *Low Mach number limit of viscous compressible flows in the whole space*, Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences 455.1986 (1999), pages 2271–2279, DOI: 10.1098/rspa.1999.0403.

[A61] J. C. I. Dooge, *Looking for hydrologic laws*, Water Resources Research 22 (9S 1986), 46S–58S, DOI: 10.1029/WR022i09Sp0046S.

[A62] M. Dryja, M. V. Sarkis, and O. B. Widlund, *Multilevel Schwarz methods for elliptic problems with discontinuous coefficients in three dimensions*, Numerische Mathematik 72.3 (1996), pages 313–348, DOI: 10.1007/s002110050172.

[A63] P. Eichler, R. Fučík, and R. Straka, *Computational study of immersed boundary-lattice Boltzmann method for fluid-structure interaction*, Discrete & Continuous Dynamical Systems-S 14.3 (2021), page 819, DOI: 10.3934/dcdss.2020349.

[A64] P. Eichler, V. Fuka, and R. Fučík, *Cumulant lattice Boltzmann simulations of turbulent flow above rough surfaces*, Computers & Mathematics with Applications 92 (2021), pages 37–47, DOI: 10.1016/j.camwa.2021.03.016.

[A65] J. Erhel, K. Burrage, and B. Pohl, *Restarted GMRES preconditioned by deflation*, Journal of Computational and Applied Mathematics 69.2 (1996), pages 303–318, DOI: 10.1016/0377-0427(95)00047-X.

[A66] A. Franceschini, V. A. Paludetto Magri, M. Ferronato, and C. Janna, *A robust multilevel approximate inverse preconditioner for symmetric positive definite matrices*, SIAM Journal on Matrix Analysis and Applications 39.1 (2018), pages 123–147, DOI: 10.1137/16M1109503.

[A67] Z. Fu, T. James Lewis, R. M. Kirby, and R. T. Whitaker, *Architecting the finite element method pipeline for the GPU*, Journal of Computational and Applied Mathematics 257 (2014), pages 195–211, ISSN: 0377-0427, DOI: 10.1016/j.cam.2013.09.001.

[A68] R. Fučík, P. Eichler, J. Klinkovský, R. Straka, and T. Oberhuber, *Lattice Boltzmann method analysis tool (LBMAT)*, Numerical Algorithms (2022), ISSN: 1572-9265, DOI: 10.1007/s11075-022-01476-8.

[A69] R. Fučík, P. Eichler, R. Straka, P. Pauš, J. Klinkovský, and T. Oberhuber, *On optimal node spacing for immersed boundary–lattice Boltzmann method in 2D and 3D*, Computers & Mathematics with Applications 77.4 (2019), pages 1144–1162, DOI: 10.1016/j.camwa.2018.10.045.

[A70] R. Fučík, R. Galabov, P. Pauš, P. Eichler, J. Klinkovský, R. Straka, J. Tintěra, and R. Chabiniok, *Investigation of phase-contrast magnetic resonance imaging underestimation of turbulent flow through the aortic valve phantom: experimental and computational study using lattice Boltzmann method*, Magnetic Resonance Materials in Physics, Biology and Medicine 33.5 (2020), pages 649–662, ISSN: 1352-8661, DOI: 10.1007/s10334-020-00837-5.

[A71] R. Fučík, T. H. Illangasekare, and M. Beneš, *Multidimensional self-similar analytical solutions of two-phase flow in porous media*, Advances in Water Resources 90 (2016), pages 51–56, DOI: 10.1016/j.advwatres.2016.02.007.

[A72] R. Fučík, J. Klinkovský, J. Solovský, T. Oberhuber, and J. Mikyška, *Multidimensional mixed–hybrid finite element method for compositional two-phase flow in heterogeneous porous media and its parallel implementation on GPU*, Computer Physics Communications 238 (2019), pages 165–180, DOI: 10.1016/j.cpc.2018.12.004.

[A73] R. Fučík and J. Mikyška, *Discontinous Galerkin and mixed-hybrid finite element approach to two-phase flow in heterogeneous porous media with different capillary pressures*, Procedia Computer Science 4 (2011), pages 908–917, ISSN: 1877-0509, DOI: 10.1016/j.procs.2011.04.096.

[A74] R. Fučík and J. Mikyška, *Mixed-hybrid finite element method for modelling two-phase flow in porous media*, Journal of Math-for-Industry 3.2 (2011), pages 9–19, URL: http://hdl.handle.net/2324/20607.

[A75] R. Fučík and R. Straka, *Equivalent finite difference and partial differential equations for the lattice Boltzmann method*, Computers & Mathematics with Applications 90 (May 2021), pages 96–103, ISSN: 0898-1221, DOI: 10.1016/j.camwa.2021.03.014.

[A76] M. Gaedtke, S. Wachter, M. Rädle, H. Nirschl, and M. J. Krause, *Application of a lattice Boltzmann method combined with a Smagorinsky turbulence model to spatially resolved heat flux inside a refrigerated vehicle*, Computers & Mathematics with Applications 76.10 (2018), pages 2315–2329, ISSN: 0898-1221, DOI: 10.1016/j.camwa.2018.08.018.

[A77] G. Gambolati, M. Ferronato, and C. Janna, *Preconditioners in computational geomechanics: a survey*, International Journal for Numerical and Analytical Methods in Geomechanics 35.9 (2011), pages 980–996, DOI: 10.1002/nag.937.

[A78] R. Gandham, K. Esler, and Y. Zhang, *A GPU accelerated aggregation algebraic multigrid method*, Computers & Mathematics with Applications 68.10 (2014), pages 1151–1160, ISSN: 0898-1221, DOI: 10.1016/j.camwa.2014.08.022.

[A79] J. Gao, Y. Zhou, G. He, and Y. Xia, *A multi-GPU parallel optimization model for the preconditioned conjugate gradient algorithm*, Parallel Computing 63 (2017), pages 1–16, DOI: 10.1016/j.parco.2017.04.003.

[A80] A. Gaul, M. H. Gutknecht, J. Liesen, and R. Nabben, *A framework for deflated and augmented Krylov subspace methods*, SIAM Journal on Matrix Analysis and Applications 34.2 (2013), pages 495–518, DOI: 10.1137/110820713.

[A81] M. Geier, A. Pasquali, and M. Schönherr, *Parametrization of the cumulant lattice Boltzmann method for fourth order accurate diffusion part I: Derivation and validation*, Journal of Computational Physics 348 (2017), pages 862–888, DOI: 10.1016/j.jcp.2017.05.040.

[A82] M. Geier, A. Pasquali, and M. Schönherr, *Parametrization of the cumulant lattice Boltzmann method for fourth order accurate diffusion part II: Application to flow around a sphere at drag crisis*, Journal of Computational Physics 348 (2017), pages 889–898, DOI: 10.1016/j.jcp.2017.07.004.

[A83] M. Geier and M. Schönherr, *Esoteric twist: an efficient in-place streaming algorithmus for the lattice Boltzmann method on massively parallel hardware*, Computation 5.2 (Mar. 2017), page 19, DOI: 10.3390/computation5020019.

[A84] M. Geier, M. Schönherr, A. Pasquali, and M. Krafczyk, *The cumulant lattice Boltzmann equation in three dimensions: Theory and validation*, Computers & Mathematics with Applications 70.4 (2015), pages 507–547, DOI: 10.1016/j.camwa.2015.05.001.

[A85] C. Geuzaine and J.-F. Remacle, *Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities*, International Journal for Numerical Methods in Engineering 79.11 (2009), pages 1309–1331, DOI: 10.1002/nme.2579.

[A86] P. Ghysels, T. J. Ashby, K. Meerbergen, and W. Vanroose, *Hiding global communication latency in the GMRES algorithm on massively parallel machines*, SIAM Journal on Scientific Computing 35.1 (2013), pages C48–C71, DOI: 10.1137/12086563X.

[A87] M. B. van Gijzen, *A polynomial preconditioner for the GMRES algorithm*, Journal of Computational and Applied Mathematics 59.1 (1995), pages 91–107, ISSN: 0377-0427, DOI: 10.1016/0377-0427(94)00015-S.

[A88] M. B. van Gijzen and P. Sonneveld, *Algorithm 913: An elegant IDR(s) variant that efficiently exploits biorthogonality properties*, ACM Transactions on Mathematical Software 38.1 (Dec. 2011), ISSN: 0098-3500, DOI: 10.1145/2049662.2049667.

[A89] L. Giraud, S. Gratton, X. Pinel, and X. Vasseur, *Flexible GMRES with deflated restarting*, SIAM Journal on Scientific Computing 32.4 (2010), pages 1858–1878, DOI: 10.1137/080741847.

[A90] A. Giuliani and L. Krivodonova, *Face coloring in unstructured CFD codes*, Parallel Computing 63 (2017), pages 17–37, DOI: 10.1016/j.parco.2017.04.001.

[A91] C. Greif, T. Rees, and D. B. Szyld, *GMRES with multiple preconditioners*, SeMA Journal 74.2 (2017), pages 213–231, DOI: 10.1007/s40324-016-0088-7.

[A92] M. J. Grote and T. Huckle, *Parallel preconditioning with sparse approximate inverses*, SIAM Journal on Scientific Computing 18.3 (May 1997), pages 838–853, ISSN: 1064-8275, DOI: 10.1137/S1064827594276552.

[A93] M. Haussmann, A. C. Barreto, G. L. Kouyi, N. Rivière, H. Nirschl, and M. J. Krause, *Large-eddy simulation coupled with wall models for turbulent channel flows at high Reynolds numbers with a lattice Boltzmann method—Application to Coriolis mass flowmeter*, Computers & Mathematics with Applications 78.10 (2019), pages 3285–3302, DOI: 10.1016/j.camwa.2019.04.033.

[A94] M. Haussmann, P. Reinshaus, S. Simonis, H. Nirschl, and M. J. Krause, *Fluid-structure interaction simulation of a Coriolis mass flowmeter using a lattice Boltzmann method*, Fluids 6.4 (2021), ISSN: 2311-5521, DOI: 10.3390/fluids6040167.

[A95] P. Hénon and Y. Saad, *A parallel multistage ILU factorization based on a hierarchical graph decomposition*, SIAM Journal on Scientific Computing 28.6 (2006), pages 2266–2293, DOI: 10.1137/040608258.

[A96] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, *An overview of the Trilinos project*, ACM Transactions on Mathematical Software 31.3 (Sept. 2005), pages 397–423, ISSN: 0098-3500, DOI: 10.1145/1089014.1089021.

[A97] M. A. Heroux and J. M. Willenbring, *A new overview of the Trilinos project*, Scientific Programming 20.2 (Apr. 2012), pages 83–88, DOI: 10.1155/2012/408130.

[A98] D. Hilbert, *Über die stetige Abbildung einer Linie auf ein Flächenstück*, Mathematische Annalen 38 (1891), pages 459–460, DOI: 10.1007/BF01199431.

[A99] H. Hoteit and A. Firoozabadi, *Numerical modeling of two-phase flow in heterogeneous permeable media with different capillarity pressures*, Advances in Water Resources 31.1 (2008), pages 56–73, DOI: 10.1016/j.advwatres.2007.06.006.

[A100] D. Hysom and A. Pothen, *A scalable parallel algorithm for incomplete factor preconditioning*, SIAM Journal on Scientific Computing 22.6 (2001), pages 2194–2215, DOI: 10.1137/S1064827500376193.

[A101] D. A. Ibanez, E. S. Seol, C. W. Smith, and M. S. Shephard, *PUMI: Parallel unstructured mesh infrastructure*, ACM Transactions on Mathematical Software 42.3 (2016), pages 1–28, DOI: 10.1145/2814935.

[A102] Y. Ishihara, E. Shimojuma, and H. Harada, *Water vapor transfer beneath bare soil where evaporation is influenced by a turbulent surface wind*, Journal of Hydrology 131 (1992), pages 63–104, DOI: 10.1016/0022-1694(92)90213-F.

[A103] C. Janna and M. Ferronato, *Adaptive pattern research for block FSAI preconditioning*, SIAM Journal on Scientific Computing 33.6 (2011), pages 3357–3380, DOI: 10.1137/100810368.

[A104] C. Janna, M. Ferronato, F. Sartoretto, and G. Gambolati, *FSAIPACK: A software package for high-performance factored sparse approximate inverse preconditioning*, ACM Transactions on Mathematical Software 41.2 (2015), pages 1–26, ISSN: 0098-3500, DOI: 10.1145/2629475.

[A105] H. Kaiser, P. Diehl, A. S. Lemoine, B. A. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S. R. Brandt, N. Gupta, T. Heller, K. Huck, Z. Khatami, A. Kheirkhahan, A. Reverdell, S. Shirzad, M. Simberg, B. Wagle, W. Wei, and T. Zhang, *HPX: The C++ standard library for parallelism and concurrency*, Journal of Open Source Software 5.53 (2020), page 2352, DOI: 10.21105/joss.02352.

[A106] S. K. Kang and Y. A. Hassan, *The effect of lattice models within the lattice Boltzmann method in the simulation of wall-bounded turbulent flows*, Journal of Computational Physics 232.1 (2013), pages 100–117, DOI: 10.1016/j.jcp.2012.07.023.

[A107] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM Journal on Scientific Computing 20.1 (1998), pages 359–392, DOI: 10.1137/s1064827595287997.

[A108] C. E. Kees, M. W. Farthing, and C. N. Dawson, *Locally conservative, stabilized finite element methods for variably saturated flow*, Computer Methods in Applied Mechanics and Engineering 197.51-52 (2008), pages 4610–4625, DOI: 10.1016/j.cma.2008.06.005.

[A109] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey, `libMesh`*: a C++ library for parallel adaptive mesh refinement/coarsening simulations*, Engineering with Computers 22.3–4 (2006), pages 237–254, DOI: 10.1007/s00366-006-0049-3.

[A110] J. Klinkovský, T. Oberhuber, R. Fučík, and V. Žabka, *Configurable open-source data structure for distributed conforming unstructured homogeneous meshes with GPU support*, ACM Transactions on Mathematical Software 48.3 (Sept. 2022), pages 1–30, ISSN: 0098-3500, DOI: 10.1145/3536164.

[A111] J. Klinkovský, A. C. Trautz, R. Fučík, and T. H. Illangasekare, *Lattice Boltzmann method–based efficient GPU simulator for vapor transport in the boundary layer over a moist soil: development and experimental validation*, Computers & Mathematics with Applications 138 (May 2023), pages 65–87, DOI: 10.1016/j.camwa.2023.02.021.

[A112] L. Y. Kolotilina, A. A. Nikishin, and A. Y. Yeremin, *Factorized sparse approximate inverse preconditionings IV: Simple approaches to rising efficiency*, Numerical Linear Algebra with Applications 6.7 (1999), pages 515–531, DOI: 10.1002/(SICI)1099-1506(199910/11)6:7<515::AID-NLA176>3.0.CO;2-0.

[A113] L. Y. Kolotilina and A. Y. Yeremin, *Factorized sparse approximate inverse preconditioning II: Solution of 3D FE systems on massively parallel computers*, International Journal of High Speed Computing 7.02 (1995), pages 191–215, DOI: 10.1142/S0129053395000117.

[A114] L. Y. Kolotilina and A. Y. Yeremin, *Factorized sparse approximate inverse preconditionings I: Theory*, SIAM Journal on Matrix Analysis and Applications 14.1 (1993), pages 45–58, DOI: 10.1137/0614004.

[A115] J. Kopal, M. Rozložník, and M. Tůma, *An adaptive multilevel factorized sparse approximate inverse preconditioning*, Advances in Engineering Software 113 (2017), pages 19–24, ISSN: 0965-9978, DOI: 10.1016/j.advengsoft.2016.10.005.

[A116] T. Kozubek, V. Vondrák, M. Menšík, D. Horák, Z. Dostál, V. Hapla, P. Kabelíková, and M. Čermák, *Total FETI domain decomposition method and its massively parallel implementation*, Advances in Engineering Software 60-61 (2013), pages 14–22, ISSN: 0965-9978, DOI: 10.1016/j.advengsoft.2013.04.001.

[A117] B. Krasnopolsky, *The reordered BiCGStab method for distributed memory computer systems*, Procedia Computer Science 1.1 (2010), pages 213–218, DOI: 10.1016/j.procs.2010.04.024.

[A118] P. Kumar, K. Kutscher, M. Mößner, R. Radespiel, M. Krafczyk, and M. Geier, *Validation of a VRANS-model for turbulent flow over a porous flat plate by cumulant lattice Boltzmann DNS/LES and experiments*, Journal of Porous Media 21.5 (2018), DOI: 10.1615/JPorMedia.v21.i5.60.

[A119] D. Langr and P. Tvrdik, *Evaluation criteria for sparse matrix storage formats*, IEEE Transactions on Parallel and Distributed Systems 27.2 (2016), pages 428–440, DOI: 10.1109/tpds.2015.2401575.

[A120] M. G. Larson and A. J. Niklasson, *A conservative flux for the continuous Galerkin method based on discontinuous enrichment*, Calcolo 41.2 (2004), pages 65–76, DOI: 10.1007/BF02637255.

[A121] J. Latt, B. Chopard, O. Malaspinas, M. Deville, and A. Michler, *Straight velocity boundaries in the lattice Boltzmann method*, Physical Review E 77 (5 May 2008), page 056703, DOI: 10.1103/PhysRevE.77.056703.

[A122] M. Lehmann, *Esoteric pull and esoteric push: two simple in-place streaming schemes for the lattice Boltzmann method on GPUs*, Computation 10.6 (June 2022), ISSN: 2079-3197, DOI: 10.3390/computation10060092.

[A123] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, *Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect*, IEEE Transactions on Parallel and Distributed Systems 31.1 (2020), pages 94–110, ISSN: 1558-2183, DOI: 10.1109/TPDS.2019.2928289.

[A124] N. Li, Y. Saad, and E. Chow, *Crout versions of ILU for general sparse matrices*, SIAM Journal on Scientific Computing 25.2 (2003), pages 716–728, DOI: 10.1137/S1064827502405094.

[A125] R. Li and Y. Saad, *GPU-accelerated preconditioned iterative linear solvers*, The Journal of Supercomputing 63.2 (2013), pages 443–466, DOI: 10.1007/s11227-012-0825-3.

[A126] R. F. Ling, *Comparison of several algorithms for computing sample means and variances*, Journal of the American Statistical Association 69.348 (1974), pages 859–866, DOI: 10.1080/01621459.1974.10480219.

[A127] P.-L. Lions and N. Masmoudi, *Incompressible limit for a viscous compressible fluid*, Journal de mathématiques pures et appliquées 77.6 (1998), pages 585–627, DOI: 10.1016/S0021-7824(98)80139-6.

[A128] H. Liu, B. Yang, and Z. Chen, *Accelerating algebraic multigrid solvers on NVIDIA GPUs*, Computers & Mathematics with Applications 70.5 (2015), pages 1162–1181, ISSN: 0898-1221, DOI: 10.1016/j.camwa.2015.07.005.

[A129] Q. Liu, R. B. Morgan, and W. Wilcox, *Polynomial preconditioned GMRES and GMRES-DR*, SIAM Journal on Scientific Computing 37.5 (2015), S407–S428, DOI: 10.1137/140968276.

[A130] W. Ma, Y. Hu, W. Yuan, and X. Liu, *Developing a multi-GPU-enabled preconditioned GMRES with inexact triangular solves for block sparse matrices*, Mathematical Problems in Engineering 2021 (2021), DOI: 10.1155/2021/6804723.

[A131] M.-L. Maier, T. Henn, G. Thaeter, H. Nirschl, and M. J. Krause, *Multiscale simulation with a two-way coupled lattice Boltzmann method and discrete element method*, Chemical Engineering Technology 40.9 (Sept. 2017), pages 1591–1598, ISSN: 0930-7516, DOI: 10.1002/ceat.201600547.

[A132] W. J. Massman, *A review of the molecular diffusivities of H2O, CO2, CH4, CO, O3, SO2, NH3, N2O, NO, and NO2 in air, O2 and N2 near STP*, Atmospheric environment 32.6 (1998), pages 1111–1127, DOI: 10.1016/S1352-2310(97)00391-9.

[A133] M. Matinfar, H. Zareamoghaddam, M. Eslami, and M. Saeidy, *GMRES implementations and residual smoothing techniques for solving ill-posed linear systems*, Computers & Mathematics with Applications 63.1 (2012), pages 1–13, DOI: 10.1016/j.camwa.2011.09.022.

[A134] D. B. McWhorter and D. K. Sunada, *Exact integral solutions for two-phase flow*, Water Resources Research 26.3 (1990), pages 399–413, DOI: 10.1029/WR026i003p00399.

[A135] A. Mink, C. McHardy, L. Bressel, C. Rauh, and M. J. Krause, *Radiative transfer lattice Boltzmann methods: 3D models and their performance in different regimes of radiative transfer*, Journal of Quantitative Spectroscopy and Radiative Transfer 243 (2020), page 106810, ISSN: 0022-4073, DOI: 10.1016/j.jqsrt.2019.106810.

[A136] A. Mink, K. Schediwy, C. Posten, H. Nirschl, S. Simonis, and M. J. Krause, *Comprehensive computational model for coupled fluid flow, mass transfer, and light supply in tubular photobioreactors equipped with glass sponges*, Energies 15.20 (2022), ISSN: 1996-1073, DOI: 10.3390/en15207671.

[A137] R. C. Mittal and A. H. Al-Kurdi, *An efficient method for constructing an ILU preconditioner for solving large sparse nonsymmetric linear systems by the GMRES method*, Computers & Mathematics with Applications 45.10-11 (2003), pages 1757–1772, DOI: 10.1016/S0898-1221(03)00154-8.

[A138] S. Mittal and J. S. Vetter, *A survey of methods for analyzing and improving GPU energy efficiency*, ACM Computing Surveys 47.2 (Aug. 2014), ISSN: 0360-0300, DOI: 10.1145/2636342.

[A139] A. A. Mohamad and S. Succi, *A note on equilibrium boundary conditions in lattice Boltzmann fluid dynamic simulations*, The European Physical Journal Special Topics 171.1 (2009), pages 213–221, DOI: 10.1140/epjst/e2009-01031-9.

[A140] A. Muñoz, *Higher order ray marching*, Computer Graphics Forum 33.8 (2014), pages 167–176, ISSN: 1467-8659, DOI: 10.1111/cgf.12424.

[A141] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka, *AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods*, SIAM Journal on Scientific Computing 37.5 (2015), S602–S626, DOI: 10.1137/140980260.

[A142] T. Oberhuber, J. Klinkovský, and R. Fučík, *TNL: Numerical library for modern parallel architectures*, Acta Polytechnica 61.SI (2021), pages 122–134, DOI: 10.14311/AP.2021.61.0122.

[A143] T. Oberhuber, J. Vacata, and A. Suzuki, *New row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA*, Acta Technica CSAV 56.4 (2011), pages 447–466, DOI: 10.48550/arXiv.1012.2270.

[A144] G. Oyarzun, R. Borrell, A. Gorobets, and A. Oliva, *MPI-CUDA sparse matrix–vector multiplication for the conjugate gradient method with an approximate inverse preconditioner*, Computers & Fluids 92 (2014), pages 244–252, ISSN: 0045-7930, DOI: 10.1016/j.compfluid.2013.10.035.

[A145] C. C. Paige and M. A. Saunders, *Solution of sparse indefinite systems of linear equations*, SIAM Journal on Numerical Analysis 12.4 (1975), pages 617–629, DOI: 10.1137/0712047.

[A146] J. W. Pearson and J. Pestana, *Preconditioners for Krylov subspace methods: An overview*, GAMM-Mitteilungen 43.4 (2020), e202000015, DOI: 10.1002/gamm.202000015.

[A147] C. Peng, N. Geneva, Z. Guo, and L.-P. Wang, *Direct numerical simulation of turbulent pipe flow using the lattice Boltzmann method*, Journal of Computational Physics 357 (2018), pages 16–42, DOI: 10.1016/j.jcp.2017.11.040.

[A148] X. Qin, R. J. LeVeque, and M. R. Motley, *Accelerating an adaptive mesh refinement code for depth-averaged flows using GPUs*, Journal of Advances in Modeling Earth Systems 11.8 (2019), pages 2606–2628, DOI: 10.1029/2019MS001635.

[A149] F. A. Radu, N. Suciu, J. Hoffmann, A. Vogel, O. Kolditz, C. H. Park, and S. Attinger, *Accuracy of numerical simulations of contaminant transport in heterogeneous aquifers: a comparative study*, Advances in water resources 34.1 (2011), pages 47–61, DOI: 10.1016/j.advwatres.2010.09.012.

[A150] O. Rendel, A. Rizvanolli, and J.-P. M. Zemke, *IDR: A new generation of Krylov subspace methods?* Linear Algebra and its Applications 439.4 (2013), pages 1040–1061, DOI: 10.1016/j.laa.2012.11.021.

[A151] Y. Saad, *A flexible inner-outer preconditioned GMRES algorithm*, SIAM Journal on Scientific Computing 14.2 (1993), pages 461–469, DOI: 10.1137/0914028.

[A152] Y. Saad, *ILUT: A dual threshold incomplete LU factorization*, Numerical linear algebra with applications 1.4 (1994), pages 387–402, DOI: 10.1002/nla.1680010405.

[A153] Y. Saad and M. H. Schultz, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing 7.3 (1986), pages 856–869, DOI: 10.1137/0907058.

[A154] Y. Saad and J. Zhang, *BILUTM: a domain-based multilevel block ILUT preconditioner for general sparse matrices*, SIAM Journal on Matrix Analysis and Applications 21.1 (1999), pages 279–299, DOI: 10.1137/S0895479898341268.

[A155] K. V. Sharma, R. Straka, and F. W. Tavares, *Current status of lattice Boltzmann methods applied to aerodynamic, aeroacoustic, and thermal flows*, Progress in Aerospace Sciences 115 (2020), page 37, ISSN: 0376-0421, DOI: 10.1016/j.paerosci.2020.100616.

[A156] A. Shioya and Y. Yamamoto, *Block red–black MILU(0) preconditioner with relaxation on GPU*, Parallel Computing 103 (2021), pages 1–13, ISSN: 0167-8191, DOI: 10.1016/j.parco.2021.102760.

[A157] V. Simoncini and D. B. Szyld, *Recent computational developments in Krylov subspace methods for linear systems*, Numerical Linear Algebra with Applications 14.1 (2007), pages 1–59, DOI: 10.1002/nla.499.

[A158] J. Solovský, R. Fučík, M. R. Plampin, T. H. Illangasekare, and J. Mikyška, *Dimensional effects of inter-phase mass transfer on attenuation of structurally trapped gaseous carbon dioxide in shallow aquifers*, Journal of Computational Physics 405 (2020), page 109178, DOI: 10.1016/j.jcp.2019.109178.

[A159] P. Sonneveld and M. B. van Gijzen, *IDR(s): A family of simple and fast algorithms for solving large nonsymmetric systems of linear equations*, SIAM Journal on Scientific Computing 31.2 (2009), pages 1035–1062, DOI: 10.1137/070685804.

[A160] B. Sousedík, J. Šístek, and J. Mandel, *Adaptive-multilevel BDDC and its parallel implementation*, Computing 95.12 (2013), pages 1087–1119, DOI: 10.1007/s00607-013-0293-5.

[A161] N. Spillane, *An adaptive multipreconditioned conjugate gradient algorithm*, SIAM Journal on Scientific Computing 38.3 (2016), A1896–A1918, DOI: 10.1137/15M1028534.

[A162] K. Stüben, *A review of algebraic multigrid*, Journal of Computational and Applied Mathematics 128.1 (2001), pages 281–309, ISSN: 0377-0427, DOI: https://doi.org/10.1016/S0377-0427(00)00516-1.

[A163] A. A. Sulyok, G. D. Balogh, I. Z. Reguly, and G. R. Mudalige, *Locality optimized unstructured mesh algorithms on GPUs*, Journal of Parallel and Distributed Computing 134 (2019), pages 50–64, DOI: 10.1016/j.jpdc.2019.07.011.

[A164] S. Sun and M. F. Wheeler, *Projections of velocity data for the compatibility with transport*, Computer Methods in Applied Mechanics and Engineering 195.7-8 (2006), pages 653–673, DOI: 10.1016/j.cma.2005.02.011.

[A165] W.-P. Tang, *Toward an effective sparse approximate inverse preconditioner*, SIAM Journal on Matrix Analysis and Applications 20.4 (1999), pages 970–986, DOI: 10.1137/S0895479897320071.

[A166] A. C. Trautz, T. H. Illangasekare, and S. Howington, *Experimental testing scale considerations for the investigation of bare-soil evaporation dynamics in the presence of sustained above-ground airflow*, Water Resources Research 54.11 (2018), pages 8963–8982, DOI: 10.1029/2018WR023102.

[A167] A. C. Trautz, T. H. Illangasekare, S. Howington, and A. Cihan, *Sensitivity of a continuum-scale porous media heat and mass transfer model to the spatial-discretization length-scale of applied atmospheric forcing data*, Water Resources Research 55.4 (2019), pages 3520–3540, DOI: 10.1029/2018WR023923.

[A168] A. C. Trautz, T. H. Illangasekare, and I. Rodriguez-Iturbe, *Role of co-occurring competition and facilitation in plant spacing hydrodynamics in water-limited environments*, Proceedings of the National Academy of Sciences 114.35 (2017), pages 9379–9384, DOI: 10.1073/pnas.1706046114.

[A169] A. C. Trautz, T. H. Illangasekare, I. Rodriguez-Iturbe, K. Heck, and R. Helmig, *Development of an experimental approach to study coupled soil-plant-atmosphere processes using plant analogs*, Water Resources Research 53.4 (2017), pages 3319–3340, DOI: 10.1002/2016WR019884.

[A170] C. Trott, L. Berger-Vergiat, D. Poliakoff, S. Rajamanickam, D. Lebrun-Grandie, J. Madsen, N. Al Awar, M. Gligoric, G. Shipman, and G. Womeldorff, *The Kokkos ecosystem: comprehensive performance portability for high performance computing*, Computing in Science Engineering 23.5 (2021), pages 10–18, DOI: 10.1109/MCSE.2021.3098509.

[A171] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, *Kokkos 3: programming model extensions for the exascale era*, IEEE Transactions on Parallel and Distributed Systems 33.4 (2022), pages 805–817, DOI: 10.1109/TPDS.2021.3097283.

[A172] N. A. Tselepidis, C. K. Filelis-Papadopoulos, and G. A. Gravvanis, *Distributed algebraic tearing and interconnecting techniques*, Numerical Algorithms 82.3 (2019), pages 809–842, DOI: 10.1007/s11075-018-0628-6.

[A173] H. A. Van der Vorst, *Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing 13.2 (1992), pages 631–644, DOI: 10.1137/0913035.

[A174] P. Vaněk, J. Mandel, and M. Brezina, *Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems*, Computing 56.3 (1996), pages 179–196, DOI: 10.1007/BF02238511.

[A175] T. Veldhuizen, *Expression templates*, C++ Report 7.5 (1995), pages 26–31.

[A176] H. F. Walker, *Implementation of the GMRES method using Householder transformations*, SIAM Journal on Scientific and Statistical Computing 9.1 (1988), pages 152–163, DOI: 10.1137/0909010.

[A177] K. Wang, S.-B. Kim, J. Zhang, K. Nakajima, and H. Okuda, *Global and localized parallel preconditioning techniques for large scale solid Earth simulations*, Future Generation Computer Systems 19.4 (2003), pages 443–456, DOI: 10.1016/S0167-739X(03)00030-X.

[A178] W. Wang, Y. Cao, and T. Okaze, *Comparison of hexahedral, tetrahedral and polyhedral cells for reproducing the wind field around an isolated building by LES*, Building and Environment 195 (2021), page 107717, ISSN: 0360-1323, DOI: 10.1016/j.buildenv.2021.107717.

[A179] B. P. Welford, *Note on a method for calculating corrected sums of squares and products*, Technometrics 4.3 (1962), pages 419–420, DOI: 10.1080/00401706.1962.10490022.

[A180] M. Wittmann, T. Zeiser, G. Hager, and G. Wellein, *Comparison of different propagation steps for lattice Boltzmann methods*, Computers & Mathematics with Applications 65.6 (Mar. 2013), pages 924–935, DOI: 10.1016/j.camwa.2012.05.002.

[A181] M. Wittmann, T. Zeiser, G. Hager, and G. Wellein, *Domain decomposition and locality optimization for large-scale lattice Boltzmann simulations*, Computers & Fluids 80 (2013), pages 283–289, DOI: 10.1016/j.compfluid.2012.02.007.

[A182] S. A. Wolfe and W. G. Nickling, *The protective role of sparse vegetation in wind erosion*, Progress in physical geography 17.1 (1993), pages 50–68, DOI: 10.1177/030913339301700104.

[A183] J. Xu and L. Zikatanov, *Algebraic multigrid methods*, Acta Numerica 26 (2017), pages 591–721, DOI: 10.1017/S0962492917000083.

[A184] U. M. Yang and V. E. Henson, *BoomerAMG: A parallel algebraic multigrid solver and preconditioner*, Applied Numerical Mathematics 41.1 (Apr. 2002), pages 155–177, DOI: 10.1016/S0168-9274(01)00115-5.

[A185] A. Y. Yeremin, L. Y. Kolotilina, and A. A. Nikishin, *Factorized sparse approximate inverse preconditionings III: Iterative construction of preconditioners*, Journal of Mathematical Sciences 101.4 (2000), pages 3237–3254, DOI: 10.1007/BF02672769.

[A186] A. Younes, P. Ackerer, and F. Lehmann, *A new mass lumping scheme for the mixed hybrid finite element method*, International Journal for Numerical Methods in Engineering 67.1 (2006), pages 89–107, DOI: 10.1002/nme.1628.

[A187] C. Yu, K. Malakpoor, and J. M. Huyghe, *Comparing mixed hybrid finite element method with standard FEM in swelling simulations involving extremely large deformations*, Computational Mechanics 66.2 (2020), pages 287–309, DOI: 10.1007/s00466-020-01851-z.

[A188] A. Zakirov, A. Perepelkina, V. Levchenko, and S. Khilkov, *Streaming techniques: revealing the natural concurrency of the lattice Boltzmann method*, The Journal of Supercomputing 77.10 (2021), pages 11911–11929, DOI: 10.1007/s11227-021-03762-z.

[A189] X. Zhang, X. Guo, Y. Weng, X. Zhang, Y. Lu, and Z. Zhao, *Hybrid MPI and CUDA paralleled finite volume unstructured CFD simulations on a multi-GPU system*, Future Generation Computer Systems 139 (2023), pages 1–16, ISSN: 0167-739X, DOI: 10.1016/j.future.2022.09.005.

[A190] L. Zheng, H. Zhang, T. Gerya, M. Knepley, D. A. Yuen, and Y. Shi, *Implementation of a multigrid solver on a GPU for Stokes equations with strongly variable viscosity based on Matlab and CUDA*, The International journal of high performance computing applications 28.1 (2014), pages 50–60, DOI: 10.1177/1094342013478640.

[A191] S.-X. Zhu, T.-X. Gu, and X.-P. Liu, *Minimizing synchronizations in sparse iterative solvers for distributed supercomputers*, Computers & Mathematics with Applications 67.1 (2014), pages 199–209, ISSN: 0898-1221, DOI: 10.1016/j.camwa.2013.11.008.

Books and theses

Books, book chapters, and theses

[B1] T. M. Aamodt, W. W. L. Fung, and T. G. Rogers, *General-purpose graphics processor architectures*, edited by M. Martonosi, Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2018, pages 1–140, ISBN: 9781627056182, DOI: 10.2200/S00848ED1V01Y201804CAC044.

[B2] J. Ahrens, B. Geveci, and C. Law, "ParaView: An end-user tool for large-data visualization," *Visualization Handbook*, edited by C. D. Hansen and C. R. Johnson, Burlington: Butterworth-Heinemann, 2005, chapter 36, pages 717–731, ISBN: 978-0-12-387582-2, DOI: 10.1016/b978-012387582-2/50038-1.

[B3] P. Bastian, *Numerical computation of multiphase flow in porous media*, Habilitation thesis, Univeristät Kiel, 1999, URL: https://conan.iwr.uni-heidelberg.de/data/people/peter/pdf/Bastian_habilitationthesis.pdf.

[B4] R. B. Bird, W. E. Stewart, and E. N. Lightfoot, *Transport phenomena*, Second, John Wiley & Sons, Inc., 2002, ISBN: 0471410772, DOI: 10.1115/1.1424298.

[B5] J. Bobot, *Implementation of data structure for polyhedral numerical meshes in TNL*, Master's thesis, Faculty of Information Technology Czech Technical University in Prague, Feb. 2022, URL: https://dspace.cvut.cz/handle/10467/99479.

[B6] F. Brezzi and M. Fortin, *Mixed and hybrid finite elements methods*, Springer series in computational mathematics, Springer–Verlag, 1991, ISBN: 978-1-4612-7824-5, DOI: 10.1007/978-1-4612-3172-1.

[B7] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A multigrid tutorial*, SIAM, 2000, ISBN: 978-0-89871-462-3, DOI: 10.1137/1.9780898719505.

[B8] W. Brutsaert, *Evaporation into the atmosphere: Theory, history, and applications*, Springer, 1982, ISBN: 9789027712479.

[B9] D. R. Butenhof, *Programming with POSIX threads*, Addison-Wesley Professional, May 1997, ISBN: 0201633922.

[B10] A. K. G. Carr, *Recycling techniques for sequences of linear systems and eigenproblems*, PhD thesis, Virginia Polytechnic Institute and State University, June 2021, URL: https://vtechworks.lib.vt.edu/handle/10919/104143.

[B11] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil, "VisIt: an end-user tool for visualizing and analyzing very large data," *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, Oct. 2012, pages 357–372.

[B12] J. O. Coplien, "Curiously recurring template patterns," *C++ gems*, 1996, pages 135–144.

[B13] R. Fučík, *Advanced numerical methods for modelling two-phase flow in heterogeneous porous media*, PhD thesis, Faculty of Nuclear Sciences and Physical Engineering Czech Technical University in Prague, Nov. 2010.

[B14] A. Gaul, *Recycling Krylov subspace methods for sequences of linear systems*, PhD thesis, Technische Universität Berlin, Aug. 2014, DOI: 10.14279/depositonce-4147.

[B15] D. Göddeke, *Fast and accurate finite-element multigrid solvers for PDE simulations on GPU clusters*, PhD thesis, Technische Universität Dortmund, Fakultät für Mathematik, May 2010, ISBN: 978-3-8325-2768-6, URL: https://www.logos-verlag.de/cgi-bin/buch?isbn=2768.

[B16] A. Greenbaum, *Iterative methods for solving linear systems*, SIAM, 1997, ISBN: 978-0-89871-396-1, DOI: 10.1137/1.9781611970937.

[B17] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *Using advanced MPI: Modern features of the message-passing interface*, edited by W. Gropp and E. Lusk, Scientific and Engineering Computation, The MIT Press, Nov. 2014, ISBN: 9780262527637, URL: http://wgropp.cs.illinois.edu/usingmpiweb/.

[B18] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable parallel programming with the message-passing interface*, edited by W. Gropp and E. Lusk, Third, Scientific and Engineering Computation, The MIT Press, Nov. 2014, ISBN: 9780262527392, URL: http://wgropp.cs.illinois.edu/usingmpiweb/.

[B19] M. F. Hoemmen, *Communication–avoiding Krylov subspace methods*, PhD thesis, EECS Department, University of California, Berkeley, Apr. 2010, URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-37.html.

[B20] J. Klinkovský, *Mathematical modelling of two-phase compositional flow in porous media*, Master's thesis, Faculty of Nuclear Sciences and Physical Engineering Czech Technical University in Prague, June 2017.

[B21] T. Krüger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, and E. M. Viggen, *The lattice Boltzmann method*, Springer, 2017, DOI: 10.1007/978-3-319-44649-3.

[B22] R. J. LeVeque, *Finite volume methods for hyperbolic problems*, volume 31, Cambridge University Press, 2002, ISBN: 0-511-04219-1, DOI: 10.1017/CBO9780511791253.

[B23] M. Livesu, "Cinolib: a generic programming header only C++ library for processing polygonal and polyhedral meshes," *Transactions on Computational Science XXXIV*, edited by M. L. Gavrilova and C. J. K. Tan, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2019, pages 64–76, DOI: 10.1007/978-3-662-59958-7_4.

[B24] J. Lottes, *Towards robust algebraic multigrid methods for nonsymmetric problems*, PhD thesis, University of Oxford, UK, 2017, ISBN: 978-3-319-56306-0, DOI: 10.1007/978-3-319-56306-0.

[B25] J. W. Ruge and K. Stüben, "Algebraic multigrid," *Multigrid methods*, Frontiers in Applied Mathematics, SIAM, 1987, chapter 4, pages 73–130, DOI: 10.1137/1.9781611971057.ch4.

[B26] Y. Saad, *Iterative methods for sparse linear systems*, SIAM, 2003, ISBN: 0-89871-534-2, DOI: 10.1137/1.9780898718003.

[B27] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit: An object-oriented approach to 3D graphics*, Kitware, 2006, ISBN: 9781930934191.

[B28] B. F. Smith, P. E. Bjørstad, and W. D. Gropp, *Domain decomposition: Parallel multilevel methods for elliptic partial differential equations*, Cambridge University Press, 1996, ISBN: 0-521-49589-X.

[B29] D. Sunday, *Practical geometry algorithms with C++ code*, Independently published, May 2021, ISBN: 979-8749449730.

[B30] A. Toselli and O. Widlund, *Domain decomposition methods: Algorithms and theory*, Springer, Berlin, Heidelberg, 2004, ISBN: 3-540-20696-5.

[B31] U. Trottenberg, C. W. Oosterlee, and A. Schuller, *Multigrid*, Academic Press, 2001, ISBN: 0-12-701070-X.

[B32] M. Voss, R. Asenjo, and J. Reinders, *Pro TBB: C++ parallel programming with threading building blocks*, Springer, July 2019, ISBN: 978-1-4842-4398-5, DOI: 10.1007/978-1-4842-4398-5.

[B33] P. Wesseling, *An introduction to multigrid methods*, John Wiley & Sons, 1991, ISBN: 0-471-93083-0.

[B34] F. M. White, *Viscous fluid flow*, McGraw-Hill, New York, 2005, ISBN: 978-0072402315.

Conference papers

Conference papers and proceedings

[C1] H. Anzt, E. Chow, and J. Dongarra, *Iterative sparse triangular solves for preconditioning*, European Conference on Parallel Processing, edited by J. L. Träff, S. Hunold, and F. Versaci, Lecture Notes in Computer Science, Springer, 2015, pages 650–661, DOI: 10.1007/978-3-662-48096-0_50.

[C2] H. Anzt, E. Chow, D. B. Szyld, and J. Dongarra, *Domain overlap for iterative sparse triangular solves on GPUs*, Software for Exascale Computing-SPPEXA 2013-2015, edited by H.-J. Bungartz, P. Neumann, and W. E. Nagel, Lecture Notes in Computational Science and Engineering, Springer, 2016, pages 527–545, DOI: 10.1007/978-3-319-40528-5_24.

[C3] H. Anzt, W. Sawyer, S. Tomov, P. Luszczek, I. Yamazaki, and J. Dongarra, *Optimizing Krylov subspace solvers on graphics processing units*, 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, IEEE, 2014, pages 941–949, DOI: 10.1109/IPDPSW.2014.107.

[C4] H. Anzt, S. Tomov, and J. Dongarra, *Energy efficiency and performance frontiers for sparse computations on GPU supercomputers*, Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM '15, San Francisco, California: Association for Computing Machinery, 2015, pages 1–10, ISBN: 9781450334044, DOI: 10.1145/2712386.2712387.

[C5] A. Asgasri and J. E. Tate, *Implementing the Chebyshev polynomial preconditioner for the iterative solution of linear systems on massively parallel graphics processors*, CIGRE Canada Conference on Power Systems, 2009.

[C6] P. Bailey, J. Myre, S. D. C. Walsh, D. J. Lilja, and M. O. Saar, *Accelerating lattice Boltzmann fluid flow simulations using graphics processors*, 2009 International Conference on Parallel Processing, IEEE, 2009, pages 550–557, DOI: 10.1109/ICPP.2009.38.

[C7] A. H. Baker, T. Gamblin, M. Schulz, and U. M. Yang, *Challenges of scaling algebraic multigrid across modern multicore architectures*, 2011 IEEE International Parallel & Distributed Processing Symposium, IEEE, 2011, pages 275–286, DOI: 10.1109/IPDPS.2011.35.

[C8] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *Efficient management of parallelism in object oriented numerical software libraries*, Modern Software Tools in Scientific Computing, edited by E. Arge, A. M. Bruaset, and H. P. Langtangen, Birkhäuser Press, 1997, pages 163–202, DOI: 10.1007/978-1-4612-1986-6_8.

[C9] G. D. Balogh, I. Z. Reguly, and G. R. Mudalige, *Comparison of parallelisation approaches, languages, and compilers for unstructured mesh algorithms on GPUs*, International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, volume 10724, Lecture Notes in Computer Science, Springer International Publishing, Dec. 2017, pages 22–43, DOI: 10.1007/978-3-319-72971-8_2.

[C10] P. Bastian, M. Blatt, C. Engwer, A. Dedner, R. Klöfkorn, S. P. Kuttanikkad, M. Ohlberger, and O. Sander, *The distributed and unified numerics environment (DUNE)*, Proceedings of the 19th Symposium on Simulation Technique, 2006, URL: https://publications.imp.fu-berlin.de/1829/.

[C11] J. C. Beard, P. Li, and R. D. Chamberlain, *RaftLib: A C++ template library for high performance stream parallel processing*, Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM '15, San Francisco, California: Association for Computing Machinery, 2015, pages 96–105, ISBN: 9781450334044, DOI: 10.1145/2712386.2712400.

[C12] M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt, *OpenMesh – a generic and efficient polygon mesh data structure*, 1st OpenSG Symposium, 2002.

[C13] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, *Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks*, Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures, Association for Computing Machinery, 2009, pages 233–244, DOI: 10.1145/1583991.1584053.

[C14] J. M. Cebri'n, G. D. Guerrero, and J. M. García, *Energy efficiency analysis of GPUs*, 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IEEE, 2012, pages 1014–1022, DOI: 10.1109/IPDPSW.2012.124.

[C15] E. Chow, A. J. Cleary, and R. D. Falgout, *Design of the hypre preconditioner library*, SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, edited by M. Henderson, C. Anderson, and S. Lyons, Lawrence Livermore National Laboratory, CA (United States), Oct. 1998, pages 106–116, URL: https://digital.library.unt.edu/ark:/67531/metadc786848/.

[C16] E. Chow and Y. Saad, *Approximate inverse preconditioners for general sparse matrices*, Colorado conference on iterative methods, volume 2, Dec. 1994, URL: https://www.osti.gov/biblio/219599.

[C17] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, Proceedings of the 1969 24th national conference, Association for Computing Machinery, 1969, pages 157–172, DOI: 10.1145/800195.805928.

[C18] L. Davidson, *Hybrid LES-RANS: Inlet boundary conditions for flows with recirculation*, Notes on Numerical Fluid Mechanics and Multidisciplinary Design, edited by S.-H. Peng and W. Haase, volume 97, Springer, Berlin, Heidelberg, 2008, pages 55–66, DOI: https://doi.org/10.1007/978-3-540-77815-8_6.

[C19] R. D. Falgout, J. E. Jones, and U. M. Yang, *The design and implementation of hypre, a library of parallel high performance preconditioners*, Numerical solution of partial differential equations on parallel computers, edited by A. M. Bruaset and A. Tveito, volume 51, Lecture Notes in Computational Science and Engineering, Springer, Berlin, Heidelberg, 2006, pages 267–294, ISBN: 978-3-540-31619-0, DOI: 10.1007/3-540-31619-1_8.

[C20] R. D. Falgout and U. M. Yang, *hypre: A library of high performance preconditioners*, International Conference on computational science, edited by P. M. A. Sloot, A. G. Hoekstra, C. J. K. Tan, and J. J. Dongarra, volume 2331, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2002, pages 632–641, DOI: 10.1007/3-540-47789-6_66.

[C21] Z. Feng and P. Li, *Multigrid on GPU: Tackling power grid analysis on parallel SIMT platforms*, 2008 IEEE/ACM International Conference on Computer-Aided Design, IEEE, 2008, pages 647–654, DOI: 10.1109/ICCAD.2008.4681645.

[C22] R. L. Graham, S. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer, *ConnectX-2 InfiniBand management queues: first investigation of the new support for network offloaded collective operations*, 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, IEEE, 2010, pages 53–62, DOI: 10.1109/CCGRID.2010.9.

[C23] M. A. Heroux and M. Sala, *The design of Trilinos*, Applied Parallel Computing. State of the Art in Scientific Computing, edited by J. Dongarra, K. Madsen, and J. Waśniewski, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2006, pages 620–628, ISBN: 978-3-540-33498-9, DOI: 10.1007/11558958_74.

[C24] Q. Huang, Z. Huang, P. Werstein, and M. Purvis, *GPU as a general purpose computing resource*, Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies, IEEE, 2008, pages 151–158, DOI: 10.1109/PDCAT.2008.38.

[C25] D. Hysom and A. Pothen, *Efficient parallel computation of ILU(k) preconditioners*, Proceedings of the 1999 ACM/IEEE Conference on Supercomputing, SC '99, Association for Computing Machinery, 1999, 29–es, ISBN: 1581130910, DOI: 10.1145/331532.331561.

[C26] H. Jasak, A. Jemcov, and Z. Tukovic, *OpenFOAM: A C++ library for complex physics simulations*, International workshop on coupled methods in numerical dynamics, volume 1000, IUC Dubrovnik, Croatia, 2007, pages 1–20.

[C27] G. Karypis and V. Kumar, *Parallel threshold-based ILU factorization*, Proceedings of the 1997 ACM/IEEE Conference on Supercomputing, SC '97, Association for Computing Machinery, 1997, pages 1–24, ISBN: 0897919858, DOI: 10.1145/509593.509621.

[C28] D. Langr, P. Tvrdík, T. Dytrych, and J. Draayer, *Fake run-time selection of template arguments in C++*, International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, edited by C. A. Furia and S. Nanz, volume 7304, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2012, pages 140–154, DOI: 10.1007/978-3-642-30561-0_11.

[C29] M. Martineau, S. Posey, and F. Spiga, *AmgX GPU solver developments for OpenFOAM*, 8th ESI OpenFOAM Conference, 2020, URL: https://www.esi-group.com/sites/default/files/resource/other/1672/8th_OpenFOAM_Conference_NVIDIA_Posey.pdf.

[C30] K. Matsuzaki and K. Emoto, *Implementing fusion-equipped parallel skeletons by expression templates*, Implementation and Application of Functional Languages, edited by M. T. Morazán and S.-B. Scholz, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pages 72–89, ISBN: 978-3-642-16478-1, DOI: 10.1007/978-3-642-16478-1_5.

[C31] A. Monakov, A. Lokhmotov, and A. Avetisyan, *Automatically tuning sparse matrix-vector multiplication for GPU architectures*, High Performance Embedded Architectures and Compilers, volume 5952, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2010, pages 111–125, DOI: 10.1007/978-3-642-11515-8_10.

[C32] T. Oberhuber and M. Heller, *Improved row-grouped CSR format for storing of sparse matrices on GPU*, Proceedings of Algoritmy, 2012, pages 282–290, ISBN: 978-80-227-3742-5.

[C33] S. Posey and F. Pariente, *Opportunities for GPU acceleration of OpenFOAM*, 7th ESI OpenFOAM Conference, 2019, URL: https://www.esi-group.com/sites/default/files/resource/other/1702/posey_abstract_openfoam_2019.pdf.

[C34] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, *Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs*, 2013 42nd International Conference on Parallel Processing, IEEE, 2013, pages 80–89, DOI: 10.1109/ICPP.2013.17.

[C35] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, *Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels*, 2019 IEEE International Conference on Embedded Software and Systems (ICESS), IEEE, 2019, pages 1–8, DOI: 10.1109/ICESS.2019.8782524.

[C36] T. Schneider, T. Hoefler, R. E. Grant, B. W. Barrett, and R. Brightwell, *Protocols for fully offloaded collective operations on accelerated network adapters*, 2013 42nd International Conference on Parallel Processing, IEEE, 2013, pages 593–602, DOI: 10.1109/ICPP.2013.73.

[C37] J. Solovský and R. Fučík, *Mass lumping for MHFEM in two phase flow problems in porous media*, Numerical Mathematics and Advanced Applications ENUMATH 2017, edited by F. A. Radu, K. Kumar, I. Berre, J. M. Nordbotten, and I. S. Pop, Springer International Publishing, 2019, pages 635–643, DOI: 10.1007/978-3-319-96415-7_58.

[C38] Q. Wu, Y. Ha, A. Kumar, S. Luo, A. Li, and S. Mohamed, *A heterogeneous platform with GPU and FPGA for power efficient high performance computing*, 2014 International Symposium on Integrated Circuits (ISIC), IEEE, 2014, pages 220–223, DOI: 10.1109/ISICIR.2014.7029447.

[C39] I. Yamazaki, S. Tomov, T. Dong, and J. Dongarra, *Mixed-precision orthogonalization scheme and adaptive step size for improving the stability and performance of CA-GMRES on GPUs*, International Conference on High Performance Computing for Computational Science, volume 8969, Lecture Notes in Computer Science, Springer, 2015, pages 17–30, DOI: 10.1007/978-3-319-17353-5_2.

[C40] I. Yamazaki, S. Tomov, and J. Dongarra, *Deflation strategies to improve the convergence of communication-avoiding GMRES*, 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, IEEE, 2014, pages 39–46, DOI: 10.1109/ScalA.2014.6.

[C41] L. T. Yang and R. P. Brent, *The improved BiCGStab method for large and sparse unsymmetric linear systems on parallel distributed memory architectures*, Fifth International Conference on Algorithms and Architectures for Parallel Processing, IEEE, 2002, pages 324–328, DOI: 10.1109/ICAPP.2002.1173595.

[C42] R. Zayer, M. Steinberger, and H.-P. Seidel, *A GPU-adapted structure for unstructured grids*, Computer Graphics Forum, volume 36, 2, Wiley Online Library, 2017, pages 495–507, DOI: 10.1111/cgf.13144.

Manuals and technical reports

Software manuals, user guides, and technical reports (not peer-reviewed)

[M1] AMD, *HIP programming guide, version 5.2*, AMD, 2022, URL: https://docs.amd.com/bundle/HIP-Programming-Guide-v5.2/page/Introduction.html.

[M2] AMD, *ROCm documentation, version 5.2*, AMD, 2022, URL: https://docs.amd.com/category/ROCm%E2%84%A2%20v5.x.

[M3] ANSYS Inc., *ANSYS Fluent 12.0 documentation*, ANSYS Inc., 2009, URL: https://www.afs.enea.it/project/neptunius/docs/fluent/.

[M4] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, and J. Zhang, *PETSc/TAO users manual*, technical report ANL-21/39 – Revision 3.17, Argonne National Laboratory, 2022, URL: https://petsc.org/release/docs/manual/manual.pdf.

[M5] T. D. Blacker, W. J. Bohnhoff, and T. L. Edwards, *CUBIT mesh generation environment. Volume 1: Users manual*, technical report, Sandia National Labs., Albuquerque, NM (United States), 1994, DOI: 10.2172/10176386.

[M6] S. Bnà, I. Spisso, M. Olesen, and G. Rossi, *PETSc4FOAM: A library to plug-in PETSc into the OpenFOAM framework*, technical report, PRACE, 2020, DOI: 10.5281/zenodo.3923780.

[M7] D. Bonachea and G. Funck, *UPC language and library specifications, version 1.3*, technical report, UPC Consortium, Nov. 2013, DOI: 10.2172/1134233.

[M8] COMSOL Multiphysics, *Introduction to COMSOL multiphysics®*, COMSOL Multiphysics, Burlington, MA, 1998, URL: https://www.comsol.com/.

[M9] C. J. Greenshields, *OpenFOAM user guide version 8*, technical report, The OpenFOAM Foundation, 2020.

[M10] Khronos, *OpenCL API specification, version 3.0*, Khronos Group, May 2022, URL: https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html.

[M11] Khronos, *SYCL 2020 specification (revision 5)*, Khronos Group, May 2022, URL: https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html.

[M12] D. R. Kincaid, J. R. Respess, and D. M. Young, *ITPACK 2.0: User's Guide*, technical report, Center for Numerical Analysis, University of Texas, 1979, URL: http://www.ma.utexas.edu/CNA/ITPACK/.

[M13] X. Li, *Direct solvers for sparse matrices*, Apr. 2022, URL: https://portal.nersc.gov/project/sparse/superlu/SparseDirectSurvey.pdf.

[M14] D. Lukarski and N. Trost, *PARALUTION user manual version 1.1.0*, Jan. 2016, URL: http://www.paralution.com/downloads/paralution-um.pdf.

[M15] Message Passing Interface Forum, *MPI: A Message-Passing Interface standard version 3.1*, June 2015, URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[M16] M. Naumov, *Incomplete-LU and Cholesky preconditioned iterative methods using CUSPARSE and CUBLAS*, technical report, NVIDIA, June 2011, URL: https://developer.download.nvidia.com/assets/cuda/files/psts_white_paper_final.pdf.

[M17] M. Naumov, P. Castonguay, and J. Cohen, *Parallel graph coloring with applications to the incomplete-LU factorization on the GPU*, technical report, NVIDIA, May 2015, URL: https://research.nvidia.com/sites/default/files/publications/nvr-2015-001.pdf.

[M18] NVIDIA, *CUDA toolkit documentation, version 1.0*, 2007, URL: https://developer.nvidia.com/content/cuda-10.

[M19] NVIDIA, *CUDA toolkit documentation, version 11.8*, 2022, URL: https://docs.nvidia.com/cuda/archive/11.8.0/.

[M20] NVIDIA, *Developing a Linux kernel module using GPUDirect RDMA*, 2022, URL: https://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[M21] NVIDIA, *NVIDIA Tesla A100 Tensor Core GPU architecture*, technical report, NVIDIA, 2020, URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[M22] NVIDIA, *NVIDIA Tesla V100 GPU architecture*, technical report, NVIDIA, 2017, URL: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[M23] NVIDIA, *Thrust quick start guide*, 2022, URL: https://docs.nvidia.com/cuda/archive/11.8.0/pdf/Thrust_Quick_Start_Guide.pdf.

[M24] K. Pruess, C. M. Oldenburg, and G. Moridis, *TOUGH2 user's guide version 2*, technical report, Lawrence Berkeley National Laboratory, Berkeley, CA (United States), 1999, DOI: 10.2172/751729.

[M25] F. Rudolf, K. Rupp, and J. Weinbub, *ViennaGrid 2.1.0 – user manual*, technical report, Vienna University of Technology, 2014, URL: http://viennagrid.sourceforge.net/viennagrid-manual-current.pdf.

[M26] *Standard for Information Technology–Portable Operating System Interface (POSIX) – System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*, technical report, IEEE, 1996, URL: https://standards.ieee.org/ieee/1003.1c/1393/.

[M27] M. Syamlal, W. Rogers, and T. J. O'Brien, *MFIX documentation theory guide*, technical report, USDOE Morgantown Energy Technology Center, WV (United States), 1993, DOI: 10.2172/10145548.

[M28] T. J. Tautges, C. Ernst, C. Stimpson, R. J. Meyers, and K. Merkley, *MOAB: a mesh-oriented database*, technical report, Sandia National Laboratories, 2004, DOI: 10.2172/970174.

[M29] *The OpenACC application programming interface*, Version 2.5, Oct. 2015, URL: https://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5_0.pdf.

Online resources

Online resources (not peer-reviewed)

[O1] AMD, *rocThrust – port of the Thrust parallel algorithms library to the ROCm/HIP platform*, 2022, URL: https://github.com/ROCmSoftwarePlatform/rocThrust.

[O2] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. M. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, and J. Zhang, *PETSc web page*, 2022, URL: https://petsc.org/.

[O3] A. Bienz and L. N. Olson, *RAPtor: Parallel algebraic multigrid version 0.4*, 2019, URL: https://github.com/raptor-library/raptor.

[O4] C++ reference contributors, *Iterator library — cppreference.com*, [Accessed January 2023], Dec. 2022, URL: https://en.cppreference.com/mwiki/index.php?title=cpp/iterator%5C&oldid=145889.

[O5] P. Cignoni and F. Ganovelli, *Visualization and Computer Graphics Library (VCGlib)*, [Accessed November 2017], 2017, URL: http://vcg.isti.cnr.it/vcglib/.

[O6] L. Davidson, *Fluid mechanics, turbulent flow and turbulence modeling*, Lecture notes in MSc courses, Feb. 2022, URL: http://www.tfd.chalmers.se/~lada/postscript_files/solids-and-fluids_turbulent-flow_turbulence-modelling.pdf.

[O7] Engineering ToolBox, *Air – dynamic and kinematic viscosity*, [Accessed March 27 2022], 2003, URL: https://www.engineeringtoolbox.com/air-absolute-kinematic-viscosity-d_601.html.

[O8] Epetra project team, *Trilinos Epetra package*, URL: https://trilinos.github.io/epetra.html.

[O9] G. Guennebaud, B. Jacob, et al., *Eigen v3*, URL: https://eigen.tuxfamily.org.

[O10] Hypre project developers, *hypre documentation*, Lawrence Livermore National Laboratory, CA (United States), 2022, URL: https://hypre.readthedocs.io/_/downloads/en/latest/pdf/.

[O11] H. Iliev, *Answer to Can I safely use OpenMP with C++11? on StackOverflow*, 2012, URL: https://stackoverflow.com/a/13839719.

[O12] Intel, *oneAPI DPC++ compiler documentation*, [Accessed January 2023], URL: https://intel.github.io/llvm-docs/.

[O13] Intel, *oneAPI Threading Building Blocks (oneTBB)*, [Accessed January 2023], 2023, URL: https://github.com/oneapi-src/oneTBB.

[O14] Intel, *Xeon Gold 6146 processor specifications*, WikiChip Semiconductor & Computer Engineering, [Accessed February 2021], 2017, URL: https://en.wikichip.org/w/index.php?title=intel/xeon_gold/6146&oldid=95146.

[O15] IT4Innovations, *IT4Innovations documentation – Karolina cluster*, [Accessed December 2022], 2022, URL: https://docs.it4i.cz/karolina/introduction/.

[O16] H. Kaiser, M. Simberg, B. Adelstein Lelbach, T. Heller, A. Berge, J. Biddiscombe, A. Reverdell, A. Bikineev, G. Mercer, A. Schaefer, K. Huck, A. Lemoine, T. Kwon, J. Habraken, M. Anderson, S. Brandt, M. Copik, S. Yadav, M. Stumpf, D. Bourgeois, A. Nair, D. Blank, G. Gonidelis, R. Stobaugh, N. Gupta, S. Jakobovits, V. Amatya, L. Viklund, P. Diehl, and Z. Khatami, *STEllAR-GROUP/hpx: HPX v1.8.1: The C++ standards library for parallelism and concurrency*, version 1.8.1, July 2022, DOI: 10.5281/zenodo.6969649.

[O17] Kokkos developers, *Kokkos Core documentation*, [Accessed January 2023], Sandia National Laboratories, URL: https://kokkos.github.io/kokkos-core-wiki/.

[O18] J. Kraus, *An introduction to CUDA-aware MPI*, NVIDIA Developer Blog, [Accessed February 2021], 2013, URL: https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/.

[O19] D. Lukarski and N. Trost, *PARALUTION*, URL: http://www.paralution.com/.

[O20] L. Maréchal, *The GMlib library v3.30*, [Accessed February 2021], 2021, URL: https://github.com/LoicMarechal/GMlib.

[O21] Mathematical Modelling Group, *Latticee Boltzmann Method project repository*, Czech Technical University, Faculty of Nuclear Sciences and Physical Engineering, 2022, URL: https://mmg-gitlab.fjfi.cvut.cz/gitlab/lbm/lbm.

[O22] N. Morita, *Monolis: Monolithc linear equation solver based on non-overlapping and overlapping domain decomposition*, 2022, URL: https://github.com/nqomorita/monolis.

[O23] A. Muñoz, *qdt: A C++11 header-based library for quadrature integrators*, 2015, URL: https://github.com/adolfomunoz/qdt.

[O24] F. Nidito, *Replacing virtual methods with templates*, [Accessed November 2017], 2014, URL: http://www.di.unipi.it/~nids/docs/templates_vs_inheritance.html.

[O25] NVIDIA, *Algebraic multigrid solver (AmgX) library*, URL: https://github.com/NVIDIA/AMGX.

[O26] NVIDIA, *Thrust: The C++ parallel algorithms library*, 2022, URL: https://nvidia.github.io/thrust/.

[O27] ROCm developers, *rocALUTION*, AMD, URL: https://github.com/ROCmSoftwarePlatform/rocALUTION.

[O28] N. Schlömer, meshio*: input/output for many mesh formats*, [Accessed February 2021], 2021, URL: https://github.com/nschloe/meshio.

[O29] J. Šístek, *Multilevel BDDC solver library*, 2022, URL: https://github.com/sistek/bddcml.

[O30] TNL project developers, *PyTNL: Python bindings for the Template Numerical Library*, Czech Technical University, Faculty of Nuclear Sciences and Physical Engineering, 2022, URL: https://gitlab.com/tnl-project/pytnl.

[O31] TNL project developers, *Template Numerical Library documentation*, Czech Technical University, Faculty of Nuclear Sciences and Physical Engineering, 2022, URL: https://tnl-project.org/documentation/.

[O32] TOP500, *Frontier*, [Accessed January 2023], 2022, URL: https://www.top500.org/system/180047/.

[O33] TOP500, *June 2022*, [Accessed January 2023], 2022, URL: https://www.top500.org/lists/top500/2022/06/.

[O34] TOP500, *Karolina*, [Accessed January 2023], 2022, URL: https://www.top500.org/system/180026/.

[O35] TOP500, *November 2022*, [Accessed January 2023], 2022, URL: https://www.top500.org/lists/top500/2022/11/.

[O36] A. C. Trautz and T. H. Illangasekare, *Evapotranspiration from two limestone blocks (plant analog) experiments*, Mendeley Data, 2022, DOI: 10.17632/6fryw4xzgh.1.

[O37] Trilinos project team, *The Trilinos Project Website*, URL: https://trilinos.github.io.

[O38] UCX project developers, *Unified Communication X – frequently asked questions*, Version 1.12.1, [Accessed December 2022], 2021, URL: https://github.com/openucx/ucx/blob/v1.12.1/docs/source/faq.md.

[O39] J. Walter, M. Koch, et al., *uBLAS*, URL: https://github.com/boostorg/ublas.

[O40] Wikipedia contributors, *Frontier (supercomputer) — Wikipedia, The Free Encyclopedia*, [Accessed January 2023], 2023, URL: https://en.wikipedia.org/w/index.php?title=Frontier_(supercomputer)%5C&oldid=1133682103.

[O41] Wikipedia contributors, *InfiniBand — Wikipedia, The Free Encyclopedia*, [Accessed January 2023], 2023, URL: https://en.wikipedia.org/w/index.php?title=InfiniBand%5C&oldid=1131703843.

[O42] A. Williams and V. J. B. Escriba, *Boost.Thread: Portable C++ multi-threading library*, Version 1.81.0, Dec. 2022, URL: https://www.boost.org/doc/libs/1_81_0/doc/html/thread.html.