Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Ole Ludvig Lingjærde Hozman

Performance analysis of domain geometry optimizations in an LBM proxy application

Master's thesis in Computer Science
Supervisor: Jan Christian Meyer
June 2023

NTNU
Norwegian University of
Science and Technology

Ole Ludvig Lingjærde Hozman

Performance analysis of domain geometry optimizations in an LBM proxy application

Master's thesis in Computer Science
Supervisor: Jan Christian Meyer
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Problem Statement

This study aims to explore optimizations of Lattice-Boltzmann method proxy applications by exploiting domain specific geometry such as macroporous materials and 3-dimensional tubular systems, and quantify the resulting performance.

Abstract

The Lattice-Boltzmann Method (LBM) is a method for simulating fluid flow well suited for simulating flow in complex geometries. Modern High-Performance Computing platforms allow large-scale and accurate simulations using LBM models, such as the 3-dimensional $D3Q27$ model, attracting interest in areas like blood-flow simulations. This thesis analyses the impact of geometry-specific optimizations targeted towards sparse geometries like coronary arteries and explores the same optimizations for porous rocks using an array of emulated geometries.

Using distributed and shared memory parallelism with MPI and OpenMP, we have created a proxy application for the LBM using the $D3Q27$ model. We have implemented optimizations targeted towards these geometries, using indirect addressing and loop structure optimizations.

Through experiments on the Fram and Betzy supercomputers, we show that these optimizations together can achieve up to 48x speedup over the geometry-unaware baseline for sparse geometries like coronary arteries. We show that the optimized versions scale close to linearly with decreasing porosity, and that we see up to 5x speedup for porosities around 10%, commonly found in porous rocks like sandstone.

Sammendrag

Lattice-Boltzmann Metoden (LBM) er en metode for å simulere væskestrøm spesielt egnet for å simulere flyt i komplekse geometrier. Moderne Høy-Ytelses Databehandlingsplattformer legger til rette for store og nøyaktige simuleringer ved bruk av LBM modeller, som f.eks. $D3Q27$ modellen, noe som vekker interesse innen områder som simulering av blodstrøm. Denne avhandlingen analyserer effekten av geometrispesifikke optimaliseringer rettet mot spredte geometrier som koronararterier, og utforsker de samme optimaliseringene for porøse steiner ved bruk av en rekke emulerte geometrier.

Ved hjelp av distribuert og delt minne-parallelitet med MPI og OpenMP, har vi laget en proxy-applikasjon for LBM ved bruk av $D3Q27$-modellen. Vi har implementert optimaliseringer rettet mot disse geometriene, ved bruk av indirekte adressering og optimalisering av løkkestrukturer.

Gjennom eksperimenter på superdatamaskiner som Fram og Betzy viser vi at disse optimaliseringene sammen kan oppnå opptil 48x raskere ytelse enn den geometri-uvitende basislinjen for spredte geometrier som koronararterier. Vi viser at de optimaliserte versjonene skalerer nær lineært med synkende porøsitet, og at vi oppnår opptil 5x raskere ytelse for porøsiteter rundt 10%, sammenlignbart med porøse steiner som sandstein.

Acknowledgements

Thanks to my supervisor Jan Christian Meyer, for his continued support, introducing me to and providing valuable insights into the world of High-Performance Computing and Computational Fluid Dynamics. And a large thank you for always providing support when morals are both low and high.

The computations were performed on resources provided by Sigma2 — the National Infrastructure for High-Performance Computing and Data Storage in Norway and the Idun[30] cluster at NTNU.

Table of Contents

List of Tables

List of Figures

Code Listings

Abbreviations

LBM	=	Lattice Boltzmann Metod
RCA	=	Right Coronary Artery
MPI	=	Message Passing Interface
HPC	=	High Performance Computing
GPU	=	Graphics Processing Unit
CPU	=	Central Processing Unit
SPMD	=	Single Program Multiple Data
NRIS	=	Norwegian Research Infrastructure Services
IA	=	Indirect Addressing
IALO	=	Indirect Addressing and Loop Optimizations

Chapter 1

Introduction

1.1 Motivation

1.1.1 Computational Fluid Dynamics

Computational Fluid Dynamics (CFD) is an increasingly important tool in the modern world, with applications in the energy sector, medical sciences and environmental studies. The Lattice-Boltzmann Method is a popular method used in simulating fluid flow that has attracted a lot of attention in the past two decades, much due to its numerical nature and inherently parallel algorithm[5]. Coupled with its ability to accurately simulate macroscopic behavior in Newtonian fluids, this puts the LBM in a position of high interest both for simulating fluid in many scenarios and a valuable asset in exploring the increasing availability of parallel machines in High-Performance Computing (HPC).

The LBM can be discretized in both two and three dimensions with a fitting lattice configuration, providing various degrees of accuracy depending on the needs of the simulation. With the availability of increasingly powerful machines, higher dimensions and larger lattice configurations of the LBM are becoming more viable targets for simulations, requiring more research into optimizing the LBM in these configurations.

Porous media have long been a popular target for the LBM, with applications like oil and gas, erosion, and pollution propagation[9, 20], much due to the LBMs no-slip bounceback boundary conditions that make it an efficient scheme for complicated geometries. Image-based modeling of blood flow using CFDs has broad and impactful implications in medical science and the planning of patient-specific interventions. However, reviews have called for tools and demonstrations of the practical feasibility of this application[34]. With a significant pain point of the LBM being its large requirements of computing resources, exploring optimization methods targeted specifically towards blood-flow simulations can help achieve this goal.

1.1.2 Proxy applications

Proxy applications can make identifying and analyzing specific performance issues in large-scale applications easier by creating a minimal application focusing on one or a few performance issues, referred to as *proxy applications*. These applications allow programmers to more accurately profile performance issues in the application and explore methods to improve them without creating a full-scale application. Creating a proxy application for the LBM, targeted towards geometry-specific optimizations, provides a platform for rapid implementation and analysis of optimizations in LBM applications.

1.2 Scope

In this thesis, we create a proxy application for the LBM to analyze optimizations that exploit the geometries of simulations in porous rocks and blood-flow simulations. As we are studying the LBM applied to blood-flow applications and porous rocks, we opt to use the $D3Q27$ discretization of the LBM, due to it's high accuracy[18] and large resource requirements. This proxy application is based on a hybrid-memory parallel $D2Q9$ proxy application we have created as part of this thesis. We then adapt the $D3Q27$ proxy application to be useful in the context of both porous rocks and blood-flow simulation, which involves creating and initializing the simulation environment for these simulation targets. Lastly, we implement some optimizations, namely indirect addressing and resulting optimizations to loop structures, and analyze their impact on the proxy application in the context of the aforementioned geometries. Our contributions are:

- We have created an LBM $D3Q27$ proxy application targeted towards HPC environments using a hybrid memory programming model with MPI and OpenMP.
- We have implemented methods to evaluate the proxy application in the context of simulating flow through emulated porous rocks and blood-flow simulations of patient-specific arteries.
- We have experimented with and analyzed optimizations targeted toward sparse geometries like blood-flow simulations, and evaluated these optimizations on emulated porous rock geometries. We show the optimizations effectiveness for each domain geometry and analyze their scalability with different configurations of domain size and computing environment. We then provide insights into the relation of the optimized LBM application and domain geometry traits like sparsity and porosity.

1.3 Structure

- Chapter 2 covers related work on indirect addressing and the LBM, and presents related work on the LBM applied to blood flow computations and porous rocks.
- Chapter 3 presents the theoretical background used in the implementation and analysis of the proxy application.
- Chapter 4 provides details on the implementation of the proxy application and the optimizations.
- Chapter 5 provides details on the experiments including the simulation domains, test environment, and benchmarks.
- Chapter 6 analyzes the results of the experiments.
- Chapter 7 concludes the results, outlines our findings, and presents ideas for future work.

Chapter 2

Related work

This chapter briefly covers related work on optimizing the LBM using indirect addressing and GPU programming. Porous rock simulations using the LBM is an explored topic, while blood-flow simulations have seen less attention. We cover both these topics in short.

2.1 Indirect addressing

Zeiser *et al.* [36] argue that simplified LBM kernels describing large applications often fail to describe the pattern of realistic simulation scenarios. They simulate an example geometry where only 44% of the geometry's cells are fluid nodes, mirroring a domain used in chemical processes. With large simulation domains, the memory requirements grow unnecessarily large, considering most of the memory covers solid nodes that don't require any computation. They implement an LBM solver where only fluid nodes are allocated, substantially reducing the memory requirement and load using indirect addressing.

Adapting the LBM to sparse geometries has previously introduced different indirect addressing schemes, such as *connectivity matrix*[29] and *fluid index array*[24, 35]. The connectivity matrix aims to reduce indirect accesses at the cost of a slight increase in memory requirements compared to a simpler access scheme such as the fluid index array, shown to impact the performance greatly. For sparse geometries, even the fluid index array scheme is shown to provide a massive benefit to the LBM, with GPU-enabled applications using fluid index arrays on sparse geometries with 1/5 or less of nodes being fluid nodes having a speedup of up to 4 times.

2.2 GPU accelerated LBM applications

GPUs provide high floating-point performance compared to CPUs and are designed to process large computations in parallel, making them a popular target for HPC applications. Several studies have already shown that the parallel nature of the GPU and its high memory bandwidth is an effective platform for the LBMs memory bound nature[18, 19]. Several

implementations also use modern GPU clusters to simulate large D3Q19 and D3Q27 LBM models. They show that combining GPU-accelerated kernels with message-passing using MPI is an effective way to utilize GPU clusters for large-scale LBM simulations. Nita *et al.* [24] show a speedup of 19.42x of a single-GPU LBM application compared to a CPU-parallel version using OpenMP. This shows the effectiveness, especially for memory-bound applications like the LBM. However, we see an issue in this comparison as a single GPU is not directly comparable to a single CPU. Comparing the GPU version to a multi-CPU cluster might be more indicative of real-world scenarios, especially in HPC environments.

2.3 Blood flow simulations in the LBM

Simulating blood flow has been shown to be accurate on a macroscopic level using the LBM. Using imaging techniques makes it possible to accurately simulate patient-specific blood flow using the LBM[34, 22]. Using both GPU accelerated clusters and indirect addressing, optimizing the LBM for these types of simulations has been explored and shown to be an exceptionally effective target geometry for these optimizations[24].

2.4 Porous rock simulations

Mattila *et al.* [20] use indirect addressing in a GPU-accelerated LBM application to simulate flow through porous rocks. They also use a *recursive bisection* domain decomposition method to achieve an almost perfect workload balance between subdomains. They show that it is viable to simulate sample sizes up to $16,384^3$ lattice nodes.

Ramstad *et al.* [27] use a $D3Q19$ lattice model to show that the LBM accurately simulates capillary pressure and wetting effects in capillary tubes. Using models of sandstone obtained from X-ray microtomography, they conclude that the simulated data obtained from LBM simulations correspond well to experimental data. Although preliminary results show the usefulness of the LBM in this simulation, they conclude that a lot of work is required to investigate the LBM's viability for full-scale simulations.

Chapter 3

Background

This chapter briefly covers the theory of the Lattice-Boltzmann method, including the chosen parameters used in our implementation. We then cover the motivation and ideas behind proxy applications and performance modeling in the HPC space. Further, we describe the programming models used, particularly shared-memory and distributed programming models using MPI and OpenMP. Lastly, we also cover the basics of CFDs and applications of the LBM for porous rocks and blood flow simulations.

3.1 Lattice-Boltzmann Method

The Lattice-Boltzmann method assumes that simple kinetic models can accurately model the macroscopic behavior of fluids when averaged over a large area[5]. The LBM models fluids as a lattice of discrete particle densities. While traditional kinetic theory uses an entirely functional space for the particles' velocity directions, the LBM uses only a few velocity directions and movement directions. Details of the LBM are already provided by Chen *et al.*[5] and several implementations of the method[19, 32, 12]. This section will give an overview of the most essential parts of the LBM.

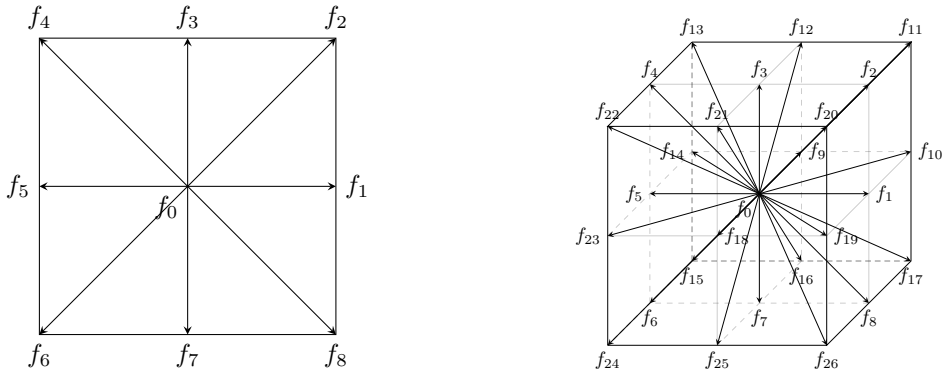
The LBM is described by the equation:

$$f_i(x + c_i dt, t + dt) = f_i(x, t) + \Delta_i(f - f^{eq}) \tag{3.1}$$

where the left side describes the *streaming* phase and the right side the *collision* phase. f_i is the density distribution at lattice point i. The density equilibrium distribution function f^{eq} is given by:

$$f^{eq}(\rho, u) = w_i \rho \left(1 + \frac{c_i \cdot u}{c_s^2} + \frac{(c_i \cdot u)^2}{2c_s^4} - \frac{u^2}{2c_s^2}\right) \tag{3.2}$$

$$\rho = \sum_i f_i \tag{3.3}$$



(a) $D2Q9$ **(b)** $D3Q27$

Figure 3.1: Lattice models for 2- and 3-dimensional lattices.

$$\rho u = \sum_i c e_i f_i \tag{3.4}$$

where c_i is the direction vector for the lattice point i and c_s determines the velocity of the fluid, chosen to be the speed of sound $c_s = 1/\sqrt{3}$. The direction vectors for $D2Q9$ and $D3Q27$ are illustrated in Fig. 3.1. w_i is the lattice weight coefficient dependent on the lattice model. ρ is the fluid density and u is the fluid velocity, given by Equations (3.3) and (3.4) respectively.
The last term in (3.1) is given as a function of the time relaxation parameter τ and the difference between the current density distribution and the equilibrium distribution after collision:

$$\Delta_i(f - f^{eq}) = -\frac{1}{\tau}(f_i - f_i^{eq}) \tag{3.5}$$

The weight coefficients for $D2Q9$ are chosen as:

$$w_i = \begin{cases} \frac{4}{9} & i = 0 \\ \frac{1}{9} & i = 1, 3, 5, 7 \\ \frac{1}{36} & i = 2, 4, 6, 8 \end{cases}$$

and for $D3Q27$:

$$w_i = \begin{cases} \frac{8}{27} & i = 0 \\ \frac{2}{27} & i = 1, 3, 5, 7, 9, 18 \\ \frac{1}{54} & i = 2, 4, 6, 8, 10, 12, 14, 16, 19, 21, 23, 25 \\ \frac{1}{216} & i = 11, 13, 15, 17, 20, 22, 24, 26 \end{cases}$$

To compute the density (and, by effect, the macroscopic velocity), we compute the local equilibrium distribution for all fluid directions before propagating the new distribution to the neighboring points. One method to do this is to store the *next* version of the density distribution $f_i + \Delta_i(f - f^{eq})$, which is computed in the collision step in a variable. The *current* version of the density distribution f_i is stored in another variable, which is updated by propagating the *next* density distribution to the *current* in the streaming step. A simulation using the LBM will then consist of several iterations of the *collide* and *stream* steps recomputing the density at each lattice point for every time step.

3.2 Proxy applications

Identifying performance issues in HPC applications is an increasingly important and challenging task, due to increasing complexity of both hardware and software. Modern applications require developers to be increasingly aware of hardware-software co-design, which further increases the complexity of porting and optimizing applications for exascale systems. To better the process of predicting the performance of such applications, the idea of *miniapplications*[7] allows developers to extract the most important factors of a full-scale application and condense it into a small program more suitable for performance evaluation and porting to several different systems and architectures. Mini-applications have also been commonly regarded as *Proxy applications*, which is the term we use in this thesis.

Modern supercomputers are highly complex and increasingly specialized systems. Systems can vary implementation details such as the underlying computing architecture, network topology, memory configuration, and core count. To develop effective programs that can take advantage of the systems they run on, programmers need to take into account a plethora of hardware-specific quirks that can affect the effectiveness and the speed of the program. To help developers and HPC system builders better understand the synergy of hardware and software, Barker *et al.* [2] provides a method to accurately model and predict the performance of applications on different systems. The method used in this thesis, inspired by Barker *et al.* [2], is visualized in Fig. 3.2.

Measuring the performance of our proxy application with different parameters, *e.g.*:

- Domain size
- Domain-specific parameters (geometry, balance)
- Process topology
- Process and thread count

we can analyze the performance impact of changes to one of the parameters. This analysis can provide the basis for an understanding of how the application will perform in similar scenarios.

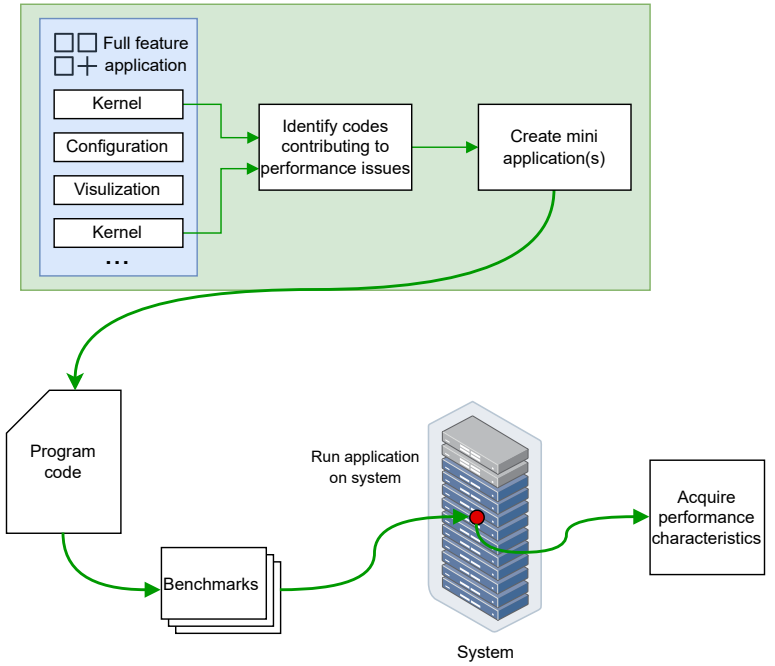


Figure 3.2: Development and analysis of a proxy application. Adapted from the graphic in [2].

3.3 Programming models

Creating programs for parallel computing systems requires that the programmer is aware of the architecture of the hardware they are targeting. In this context, a parallel programming model classifies how we construct programs to be parallel, specifically, how the parts of the program running in parallel communicate and synchronize. There exist several well-known parallel programming models[15]. However, we only cover shared-memory programming and distributed memory programming/message-passing.

3.3.1 Distributed memory programming

In a distributed memory programming model, also known as *message-passing*, multiple processes (tasks) exchange data by sending and receiving messages on a network. These tasks only require network access between each other and do not depend on the underlying architecture of the system. Tasks can reside on different machines on a network or on the same machine.

MPI

Distributed memory programming applications require software support to synchronize and share messages between processes. *Message Passing Interface* (MPI) is a *message-passing library interface specification*[10] designed to do precisely this. The MPI standard

defines a large set of procedures and datatypes that allow programmers to send messages between processes connected over a network. It has several implementations, most notably in the C and Fortran programming languages. The most well-known implementations for C compilers include *OpenMPI* and *Intel MPI*. As MPI defines a strict interface, switching out the implementation that is used is as simple as changing the compiler with no change to the underlying program.

Using MPI for distributed-memory programming can be done with the help of the *Single Program, Multiple Data* (SPMD) programming paradigm. In SPMD, the programmer creates a single application that is run on all processes, where MPI function calls are identical on all processes. The MPI library then handles the synchronization and message passing between processes upon the invocation of MPI procedures. Since all programs have a *rank* (or process ID), the programmer may also dynamically execute code on specific processes.

Working with complicated data structures in distributed-memory applications is a non-trivial matter. To help with this issue, MPI provides a robust and flexible type system that can be used directly in MPI communication procedures. The programmer can define MPI_Types that describe the memory layout of a specific object and how objects are spread out in memory. Some notable MPI types used in this thesis include:

- **Contiguous types:** A contiguous, fixed-size block of data.
- **Strided vectors:** Allows defining a vector with evenly spaced blocks of data in memory.
- **Indexed blocks:** Allows defining data structure that is non-uniformly spread across a memory region by defining an index array containing the memory offsets to each object in the memory region.

MPI also allows the combination and redefinition of MPI types based on other MPI types to create complex types that build on more fundamental types. *E.g.* one can define a 1D vector corresponding to a row of data in a 2D matrix. Expanding this to 3 dimensions, one can use the 1D vector type to define a 2D vector corresponding to a plane in a 3D matrix space. With these types, programmers can send messages containing a 2D plane normal to arbitrary x, y, or z coordinates in MPI message-passing procedures by passing the type and a memory offset as parameters.

A feature not related to the message-passing that MPI implements is *timers*. MPI defines the MPI_Wtime() function that accurately returns the elapsed time of some time in the past, for the process it is called in.

3.3.2 Shared-memory programming

A common way of implementing parallel programming is using *shared-memory programming*. In this programming model, we utilize the operating systems' light-weight threads that share the same memory space, operating on the same values in memory. By using the same memory space, the threads can communicate by accessing and storing data in memory as usual, without the need for message-passing procedures, such as in MPI. Since these threads use the same memory space, it also allows us to quickly create and destroy new threads during the lifetime of a program. The drawback of this method is that it is

effectively limited by the hardware the program is running on, in that we can only run as many parallel threads as the CPU supports. To overcome this limitation and combine the strengths of shared and distributed-memory processing, programmers can use a *hybrid programming model*.

OpenMP

OpenMP[26] is a specification for compiler-supported shared-memory programming. It is implemented in many standard C compilers, such as the GNU compiler and the Intel compilers used in this thesis. OpenMP provides the programmer with a set of *directives* and several library functions that can be used to create threads. An OpenMP compiler directive consists of several combined constructs to effectively inform OpenMP of how to parallelize a structured block in a program. One such directive is `parallel for`. The example C code in Listing 3.1 shows how this compiler directive is used in a program to parallelize a for loop. In this example, OpenMP will create `OMP_NUM_THREADS` threads, each working on a subset of the loop. `OMP_NUM_THREADS` is either decided by OpenMP automatically by inferring the number of available CPU cores, defined using environment variables, or set by the program using a library function.

The `parallel for` directive has several options for scheduling the loop iterations among threads, where the default scheduling is left up to the compiler to decide.

```
1 #pragma omp parallel for
2 for (int i=0; i < length; i++) {
3    compute_value(i);
4 }
```

Listing 3.1: Shared-memory parallelization of a for loop using OpenMP directives

3.3.3 Hybrid-memory programming

To best utilize the strengths of each programming model, we can combine the two models to create a hybrid-memory parallel program. This model uses a distributed-memory programming interface to create separate processes assigned to several compute nodes. Each process can then use shared-memory programming to maximally utilize the parallel capability of the CPU it is running on without the overhead of inter-process message passing.

3.4 Amdahls law

Amdahl's law[1] describes the theoretical speedup of an application at a fixed workload. It describes how the performance improvement of an application is limited by the fraction of time spent by the part that is susceptible to improvement. In the context of parallel programs, Amdahl's law can be formulated by:

$$Speedup = \frac{1}{(1-P) + \frac{P}{N}} \tag{3.6}$$

where N is the core count, provided the parallelizable part has a speedup equal to the number of cores. P is the proportion of time occupied by the parallelizable part.

The law can also provide an upper bound for the theoretical speedup of an application, when $N \to \infty$:

$$Speedup = \frac{1}{1-P} \tag{3.7}$$

3.5 Strong and weak scaling

Strong scaling

The type of scaling that is described in Amdahl's law, where the problem size stays constant while the core count increases, we refer to as **strong scaling**.

Weak scaling

Gustafson's law[13] states that the parallel part of a program scales linearly with the number of cores, while the serial part stays constant. This scaling, where the problem size is increased with an increased core count, is referred to as **weak scaling**.

3.6 LBM simulation domains

3.6.1 Porous rocks

Porosity is defined as the fraction of the bulk volume of the porous sample that is occupied by pore or void space[8]. In fluid simulations, the porosity is interpreted as the percentage of volume that can be occupied by a fluid. Another important aspect of porous media relevant in simulating fluid flows is the *permeability*. The permeability is a measure of the conductivity of Newtonian fluids in a medium[8], essentially describing how well a fluid can flow through the medium. Cancelliere *et al.* [4] have experimented with randomly generated porous media and their permeability using the LBM. The medium is created using a collection of spheres with a radius R, with the sphere center placed at random points.

When simulating fluid flow through a medium obtained by scanning porous rocks, we know that typical porosity levels of porous rocks can range from around 10–25% for sandstone[9, 3], down to 8–20% when simulating gas flow through gas rocks[14]. This

provides a basis for the range of porosities that are interesting to consider when creating proxy domains for porous rock simulations using the LBM.

3.6.2 Blood flow simulations

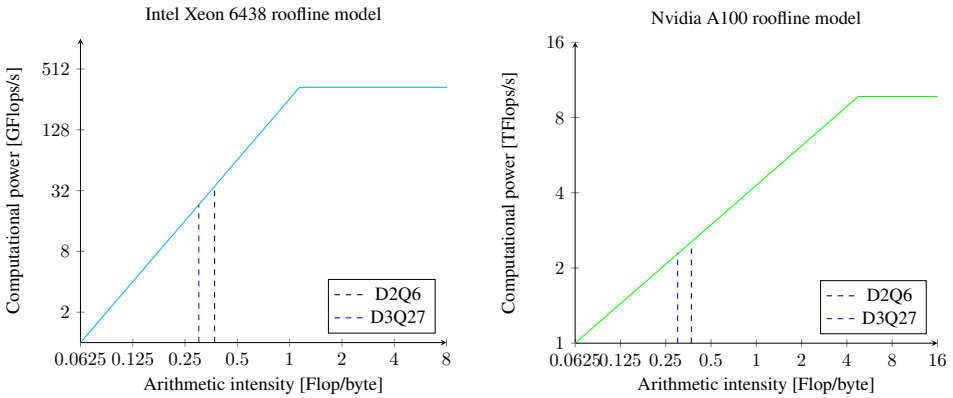
Even though blood is considered a non-newtonian fluid on the microscopic level CITE, it is useful to simulate blood flow using the LBM on both microscopic and macroscopic levels. Sun and Munn [33] apply the LBM by simulating the interactions of red and white blood cells moving through a Newtonian fluid. For large arteries and coronary systems, however, the condition of Newtonian fluid is met, which makes the LBM a contender for accurately simulating blood flow[22].

Boundary conditions in blood flow simulations are slightly more complicated than porous rocks, as it's not as easy to define a periodic flow direction to a potentially more complicated structure. One possible boundary condition involves expressing, at the inlet, u_x and f_i ($i = 1, 2, 8, 10, 11, 17, 19, 20, 26$ for $D3Q27$) based on the known f_i and a set ρ_{in} after streaming. Using the expression for the velocity in the x-direction and a bounce-back rule for the unknown f_i, we can satisfy the momentum and define all f_i[37]. This approach is used for the blood flow simulations performed by Nita *et al.* [24]

3.7 LBM applications

It is well known that the LBM is highly memory bound, much due to the low operational intensity and non-local memory access pattern of the stream kernel. We have created roofline models showing the memory-bound nature of the LBM using an estimated arithmetic intensity for the collide and stream kernels of a $D2Q9$ and $D3Q27$ proxy application (Fig. 3.3). The roofline models also show that large models, such as the $D3Q27$ models, further decrease the arithmetic intensity and increase the memory load of the application.

This shows that optimizations of the LBM should primarily focus on improving the memory performance of the application. These improvements should be directed towards the streaming step, as it has very low arithmetic intensity and usually exhibits a non-ideal memory access pattern for an array-of-structures memory layout. This is due to the low spacial locality resulting from the memory layout placing adjacent fluid nodes far apart in memory.



(a) Roofline model for Intel Xeon gold 6438

(b) Roofline model for Nvidia A100

Figure 3.3: Roofline models for the LBM on a modern CPU and GPU. The dotted line shows the operational intensity of the LBM and what performance can be expected. The LBM kernel has low arithmetic intensity and is, as such very memory bound on most architectures

Chapter 4

Implementation

In this chapter, we provide details on the implementation of the D3Q27 proxy application developed for this thesis. We begin by describing the general flow of the program before going into more detail on how we have applied the theory from Section 3.1 regarding the LBM into the implementation of the streaming and collision kernels. We also detail the approach used to parallelize the program using both OpenMP and MPI. Next, we cover the modifications made during the implementation of the indirect addressing scheme for the optimized proxy application. We also present the memory layout both with and without indirect addressing.

4.1 Baseline implementation

The base implementation of the proxy application can be split up into three parts:

- Initialization
- Main loop
- Finalize

The program's execution model is visualized with the flow diagram in Fig. 4.1.

Node types

The application differentiates between three different types of nodes: **solid**, **fluid** and **wall**. The wall node type is a special node to simplify the logic of fluid-solid boundary conditions explained in Section 4.1.7.

4.1.1 Initialization

In the initialization step, we perform basic setup of MPI, as well as domain setup to either load a pre-generated domain definition or generate a domain from user parameters:

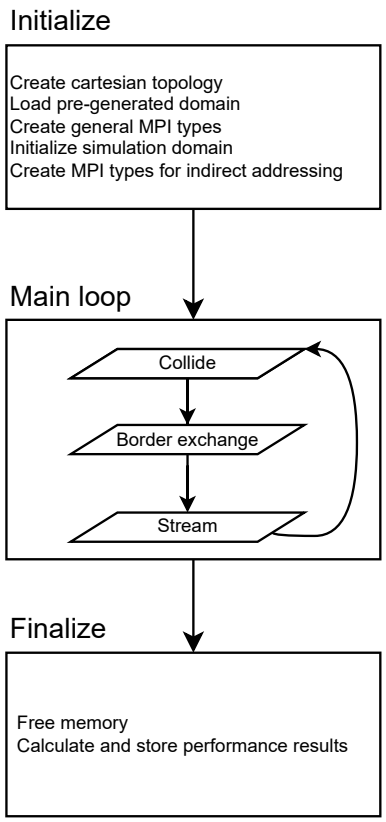


Figure 4.1: Flow diagram of the proxy application

1. **Create cartesian topology**: The first step is to initialize MPI. Any dynamic parameters, such as iteration count, domain specifications, and topology configurations, are broadcast to all processes. We then create a new communicator and initialize the MPI cartesian topology. The method is loosely based on the methodology outlined in [6].

2. **Load pre-generated domain**: Read a domain definition file to load an existing domain in voxel format.

3. **Create general MPI types**: Based on the size of the domain, MPI vector types are created for rows and planes in all directions, in addition to basic types for a fluid node.

4. **Initialize simulation domain**: Create the simulation domain described by the input file or user parameters. This configures the memory for all fluid/solid nodes and classifies any wall nodes belonging to the specific rank.

5. (**Create MPI types for indirect addressing**: For the indirect addressing only, we

create new types for all borders based on the domain definition from step 4 to be used in place of the general MPI types in border exchange.)

4.1.2 Main loop

The main loop performs, in order, the *collision*, *border exchange*, and *streaming* steps. We also take a snapshot of the current state of the simulation at configurable intervals.

4.1.3 Finalize

The finalization step calculates and writes performance data collected during the simulation, cleans up memory and finalizes MPI before exiting.

4.1.4 Memory layout

For the baseline configuration, we store two copies for the density distribution, the *current* and *next* distribution. In the collision kernel, we compute the density distribution at each lattice point from the *current* distribution and store the new value in the *next* array. During the streaming phase, the *next* density distributions are copied to the neighboring lattice points in the *current* array. The current and next density distribution arrays are stored next to each other in memory.

The memory layout of the density distribution arrays is chosen to be the array-of-structures type of C memory layout, where we store the lattice points of a single fluid node consecutively in memory in order $f_0 \dots f_{26}$ as defined in Fig. 3.1a. The fluid points are then stored consecutively again. The order of the fluid nodes is such that the coordinates of the fluid nodes run from fastest to slowest in the x-direction, y-direction, then z-direction. The blue, red, and green arrays and corresponding blocks in the cube in Fig. 4.2 show how this memory layout relates to the physical simulation environment.

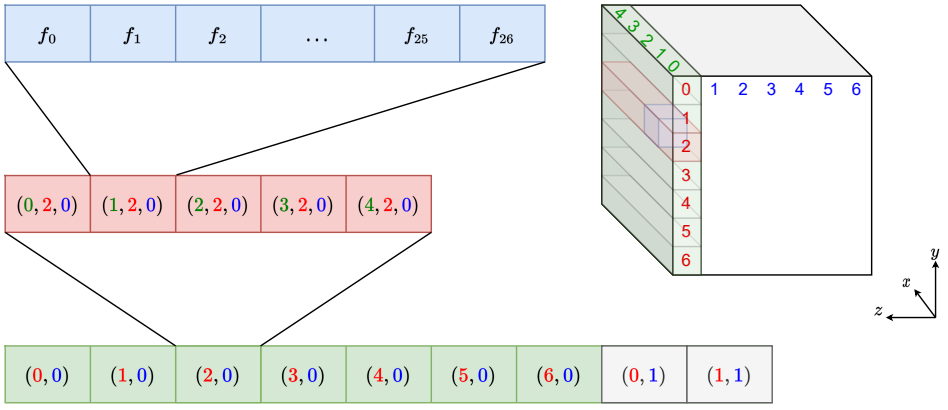


Figure 4.2: Memory layout of the density arrays. The array in blue corresponds to a single fluid node, the array in red a vector in the x-direction, and the array in green a plane in the x-y direction.

With this memory layout, we can define the macro in Listing 4.1 to access a single lattice point in the density array using its coordinates and lattice point index. We add two to the height and one to the depth and width to account for ghost cells. This means that D_now(0, 0, 0) is the first internal node of the ranks domain, and D_now(0, −1, −1) is the upper left ghost cell in the same rank. We don't want to store ghost cells separately, as keeping the ghost cells close to the other cells could be beneficial for the memory performance of the stream kernel.

```
#define D_now(i, j, k, d) \
  density[0][D_COUNT * ( \
      (((width) * (height + 2)) * (k + 1)) + (width) * (i + 1) + (j) \
    ) + (d)]
```

Listing 4.1: The macro used to calculate the position in memory of the lattice point d in the fluid node at $(x, y, z) = (j, i, k)$

4.1.5 Streaming

As the name implies, the streaming step operates similarly to the STREAM program[21]. The entirety of the stream kernel is included in Listing 4.2, where the streaming operation is highlighted. As we need to account for ghost cells, we also add a set of boundary checks to ensure that we only propagate the density to an internal fluid node and don't perform illegal memory accesses.

The kernel makes sure to loop over the fluid nodes and lattice points in the same order they appear in memory, taking advantage of data locality and maintaining cache performance. The outer loop is parallelized using the OpenMP parallel **for** directive. We assume that the compiler will find an optimal scheduling for the threads with this setup.

```
1  #pragma omp parallel for
2  for (int_t k = -1; k < depth + 1; k++)
3    for (int_t i = -1; i < height + 1; i++)
4      for (int_t j = 0; j < width; j++) {
5        for (int_t d = 0; d < D_COUNT; d++) {
6
7          int_t ni = (i + OFFSET(d, 0));
8          int_t nj = (j + OFFSET(d, 1) + width) % width; // wrap around in x
9          int_t nk = (k + OFFSET(d, 2));
10
11         if (ni < 0 || nk < 0 || ni > height - 1 || nk > depth - 1)
12           continue;
13
14         /* Propagate present fluid density
15          * to neighbors
16          */
17         D_now(ni, nj, nk, d) = D_nxt(i, j, k, d);
18     }
19 }
```

Listing 4.2: The stream kernel as programmed in the D3Q27 proxy application. The streaming operation is highlighted on line 17.

4.1.6 Collision

For the collision step, we use the same loop structure as the streaming kernel, with the difference that we don't iterate over the ghost cells. As we need the current velocity for the collision operation in Eq. (3.2), we calculate the current velocity of the fluid node in all directions before iterating over the lattice points. The fluid velocity is calculated from the *current* density distribution using Eq. (3.4). The relevant code performing this calculation is included in Listing 4.3, where the value for ρ is computed simultaneously.

```
for (int_t d = 0; d < D_COUNT; d++) {
  rho += D_now(i, j, k, d);
  V(i, j, k, 0) += c[d][0] * D_now(i, j, k, d);
  V(i, j, k, 1) += c[d][1] * D_now(i, j, k, d);
  V(i, j, k, 2) += c[d][2] * D_now(i, j, k, d);
}
V(i, j, k, 0) /= rho;
V(i, j, k, 1) /= rho;
V(i, j, k, 2) /= rho;
```

Listing 4.3: Calculation of the fluid velocity for each fluid node in the collision kernel.

After completing the calculation of the equilibrium function at each point, we add the result of the collision operator to the *next* density distribution. In Listing 4.4, the new distribution is added to the corresponding lattice point based on the type of the node:

- **Fluid**: The redistributed density is copied to the same lattice point in the *next* array.
- **Wall**: If the lattice point is part of a wall node, the density is copied to the opposing lattice point in the *next* density array. This corresponds to the bounce-back boundary condition.
- **Solid**: Ignore

```
switch (MAP(i, j, k)) {
 /* Redistribute fluid according to density/velocity */
 case FLUID:
   D_nxt(i, j, k, d) = D_now(i, j, k, d) + delta_N;
   break;
 /* Walls reflect incoming mass in opposite direction */
 case WALL:
   if (d != 0)
     D_nxt(i, j, k, bounce(d)) = D_now(i, j, k, d);
   break;
 /* No work to do on solid points */
 case SOLID:
   break;
}
```

Listing 4.4: Storing the result of the collision operation to the new density distribution based on the node type.

The complete collision kernel is included in Appendix A.1.

4.1.7 Boundary conditions

The boundary conditions we have used in this proxy application use common bounceback conditions[37] and periodic flow at the inlet/outlet[32]. The bounceback condition is applied to wall nodes in the collision kernel. For the borders of the simulation domain, we use a periodic flow in a chosen flow direction, creating an inlet and an outlet. For this application, we have opted not to make the flow periodic in the other directions, and instead make the assumption that there are always walls at the other edges of the domain. The flow direction is chosen to be the x direction, such that the inlet is at $x = 0$ and the outlet at $x = width$ as shown in Fig. 6.1c.

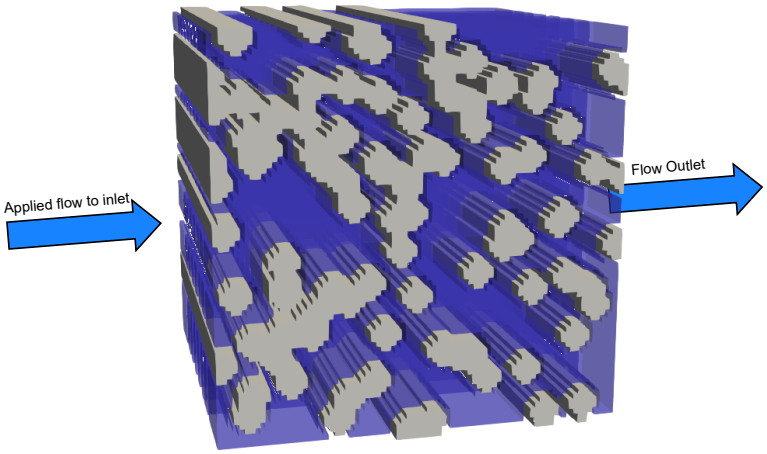


Figure 4.3: The boundary conditions shown on an 80% generated porous domain. Solid nodes are colored gray while the fluid nodes are shown in blue. The flow is applied at the left edge and flows through the domain to the outlet at the right edge. The periodic flow means that the fluid flow at the outlet is wrapped around to the inlet again.

In Listing 4.5, the inlet is programmed by adding a configurable force after calculating Δ_i in the collision kernel.

```
1    The collision kernel applies An artificial force, j, k) == FLUID)
2  delta_N += w[d] * (c[d][0] * force[0] + c[d][1] * force[1] +
3                     c[d][2] * force[2]);
```

Listing 4.5: Boundary condition at the inlet. An artificial force is applied to the fluid nodes at $x = 1$.

At the outlet boundary, we program the flow to be periodic by streaming the density of a lattice point with a direction vector in the positive x direction to the inlet. In Listing 4.6, the index of the *current* fluid node is calculated this way.

```
1 int_t ni = (i + OFFSET(d, 0));
2 int_t nj = (j + OFFSET(d, 1) + width) % width; // wrap around in x
3 int_t nk = (k + OFFSET(d, 2));
```

Listing 4.6: Outlet boundary condition. The density distribution is streamed by wrapping around in the x direction. OFFSET is a macro to the lattice points direction vector.

4.1.8 Border exchange

Traditional stencil applications involve computing the value at a single point as a function of neighboring points in a *pull*. In the LBM, a similar strategy is used, except in reverse. In a single node, we propagate all values in that node to all neighbors in a *push* fashion. This means that nodes at the edges of a process' local domain need to both: **propagate to all neighboring processes** and **push a neighbor's values to its local edge nodes**. For MPI and distributed computing applications, this can be done using *ghost cells*[17]. These ghost cells are added to the outside of a rank's local domain, acting as a proxy for the neighbor points. The values are added to the ghost cells by transferring the local edge of a process' local domain to its neighbor's corresponding ghost cells.

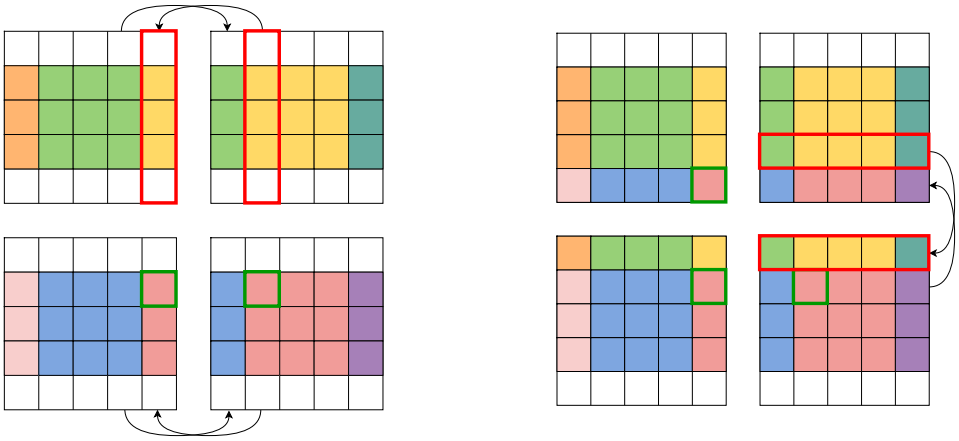


Figure 4.4: Border exchange of ghost cells in a cartesian topology. The red outline indicates a border, while the green block represents a corner node being transferred across two processes.Fig. 4.4a displays the first border exchange of columns, while Fig. 4.4b shows the row exchange. The green corner cell, transferred from the bottom-right to the top-left process, is part of both exchanges.

To perform the border exchange, we use the method detailed in [17]. The simplest method is to use the blocking MPI_Sendrecv method to both send and receive simultaneously. Using this method, we ensure that the borders are transferred in the correct order to avoid deadlocks. The main issue to overcome with a cartesian topology is correctly transferring corner cells. The neighboring process does not own these cells, so it can't be included directly in the border exchange. To overcome this issue, we exchange all cells in one direction first, including all corner cells (Fig. 4.4a). The corner cells will not be correct in this instance, but since the corner is part of the neighbor's ghost border, we can transfer the corner by including part of this border when transferring in the other direction, as seen in Fig. 4.4b. The complete code for the border exchange (with the optimizations) is included in Appendix A.3.

In order to exchange the border cells, we use MPI types to easily identify and transfer the relevant data to neighboring processes. In this proxy application, we use the same process topology as a 2-dimensional application with an added dimension. This means that instead

of 1-dimensional borders, the borders in the application are 2-dimensional planes. In order to create the required MPI types for these border planes, we can build on the 2D borders by creating a new vector that uses the 2D vectors as base units and adding $height$ vectors; we now have an MPI type representing surface planes of all sides of a process. The complete code for creating the MPI types are included in Appendix A.2.

4.1.9 Result storage

In order to evaluate the correctness of the application, we need to periodically save the state of the system so that we can see how the system evolves over time. It is also useful to visualize several aspects of the application and the domain it is simulating. For this proxy application, we have opted to save the raw data for post-processing at a later stage. Using NetCDF[28], we save the variables listed in Table 4.1 for each node in the simulated domain. We use NetCDF due to its robust C library with MPI parallel file access integration, and its widespread use in scientific computing.

Table 4.1: The NetCDF variables we store for each node in the simulation geometry.

map	Map type of the node (Solid, Fluid or Wall)
velocity	Absolute velocity of the node
v_z	Velocity in z-direction
v_y	Velocity in y direction
v_x	Velocity in x-direction
rank	The rank of the process that owns the node

For each snapshot of the system state, we create a new netcdf4 file and use NetCDF's MPI parallel access mode to write the value of all variables owned by the rank to the file. The velocity is calculated from v_x, v_y and v_z at each snapshot.

4.2 Experiment domain

4.2.1 Domain loading

As a large part of the motivation for this thesis is testing various domain geometries and types, we require a method to use already existing models in addition to generated domain geometries. Several of the models used in the experiment are 3D models of coronary arteries[31] that are accessible in the form of STL files. In order to use the models in a simulation environment, we need the dataset in a fixed-size voxel format. To convert the 3D mesh files to a voxel format, we use the *binvox* voxelizer program[23, 25] to convert the STL models to fixed sized binvox files.

With the domain geometries in a fixed-size voxel format, the application needs to parse the geometry file and set up the domain in accordance with the specification. To do this, we have created a small library containing functions for both writing and loading files in the binvox file format. These functions are used in the proxy application to load arbitrary

binvox files in the domain initialization step. Similarly, we use the function to write a domain geometry in memory as a binvox file. This last step is a requirement for being able to reuse a pseudo-random generated domain geometry in multiple experiments, as detailed in Section 5.1.

4.3 Indirect addressing

The indirect addressing scheme chosen for this proxy application is the *fluid node index* method. In this method, we use the same memory layout as detailed in Section 4.1.4, but only fluid nodes are allocated in the density arrays. This means that the nodes in the density array no longer correspond to the node's index in the physical domain. Instead, we create a new *index array*, shown in Fig. 4.5. The index array contains the data index of the node in the density arrays. Solid nodes have an index of -1, indicating that they have no corresponding data. The issue with this method is that we no longer have a reverse mapping from a node in the density array to its index in the physical domain. This is not an issue directly, but as we detail in later sections, iterating over the index array and finding its index is useful for further optimizations. This mapping is implemented by creating a corresponding array containing the x, y, and z coordinates of all fluid nodes in the density array. The two arrays have corresponding indices, providing the ability to map from one to the other.

In order to set up the indirect addressing, we calculate the number of fluid nodes after creating the environment map and only allocate memory for the fluid nodes.

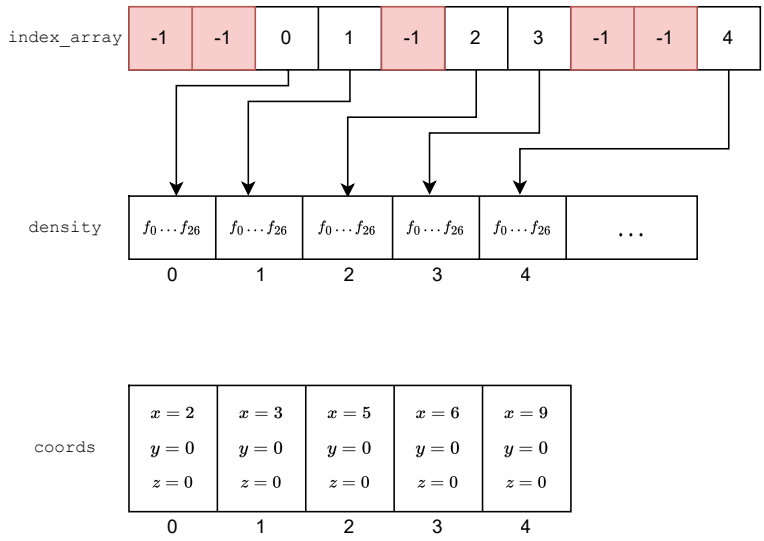


Figure 4.5: The indirect addressing implemented here. The index array points to the location of the data in the density address if it exists. Nodes without data are set to -1. An additional array is added, containing a mapping from indices in the density array to its coordinates.

As we already have a macro for accessing the density arrays, we can exchange the code in Listing 4.1 with the two macros in Listing 4.7 to use indirect addressing with minimal change to the source code.

```
// Get the index of the fluid node at position (i, j, k). Might be -1
#define P_index(i, j, k)                                                \
  index_arr[                                                            \
    (((width) * (height + 2)) * (k + 1)) + (width) * (i + 1) + (j)  \
  ]

// Current density deistribution at (i, j, k) of lattice point d
#define D_now(i, j, k, d) density[0][D_COUNT * P_index(i, j, k) + (d)]
```

Listing 4.7: The updated macros used to access the density array using the index from the index array.

4.3.1 Border exchange

The indirect address mapping poses a challenge, especially for distributed computing, when we need to distribute borders to neighboring processes. One method could be to store all border cells separately from internal cells and perform the transfers as before. However, one benefit we wish to explore is how minimizing data transfers will affect the performance of the application. This is especially interesting for sparse geometries, as we might see many border exchanges drastically reduce in data size. For some borders, we can also expect to see that the borders don't contain any fluid nodes, and we can omit the transfer altogether.

The downside of using an indirect addressing scheme is that it is no longer trivial to transfer a block, or evenly spaced blocks of data in memory as is possible with a filled data structure. As the fluid nodes we now want to transfer can be located anywhere within the memory region of the density arrays, we want to be able to transfer a specific subset of the nodes in the array using the same MPI functions as before.

To transfer borders in the same manner as before, we utilize the MPI_Indexed_block type. The process of creating the border MPI types needed for both ghost borders and internal borders is visualized in Fig. 4.6. This process is extracted to a function that can create the indirect addressed border types from the mapped border type, the contents of which are included in Listing 4.8

Using MPI_Gather with a normal border type, we can copy the index array of a border (1) such that we are left with only the indexes of the cells in question. This data also implicitly contains the mapping information of a single border. The blue cells in (2) indicate the fluid cells in a border. We then iterate over the fluid indices of the border, where the blue cells will contain the index to the node in the density array (3), and solid nodes contain -1. For every valid index, we add this to a new array containing just the indices of the border cells that exist in the density array (4). The resulting array can be used directly in MPI_Create_Indexed_block to create a new MPI type for all fluid nodes in the border. This process is repeated for all the borders in each process.

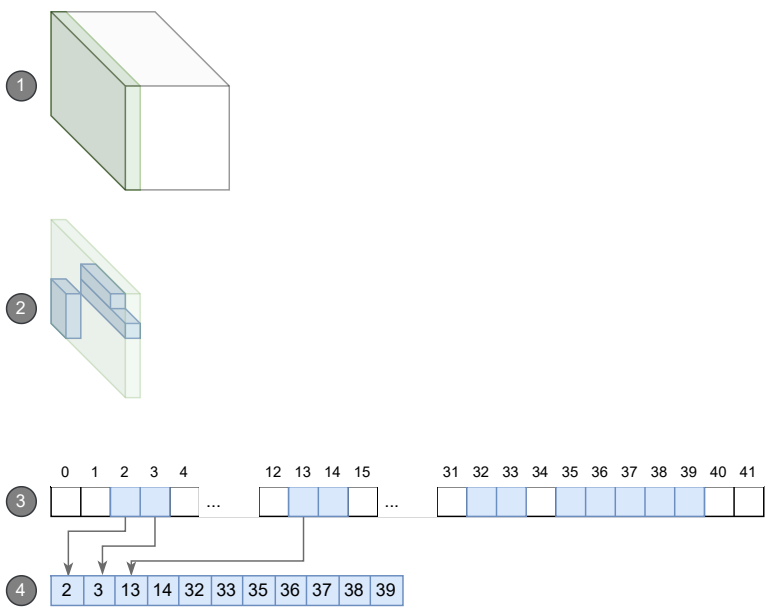


Figure 4.6: The process of finding fluid nodes in a process' border and creating an index array of the fluid nodes in the border to be used when creating MPI types.

```
// Initialize border index array
int_t node_count = 0;
for (int_t i = 0; i < l_tot_size; i++) {
 local_indexes[i] = -1;
}

// Copy border indices to local array
MPI_Gather(
 indices, 1, *old_type, local_indexes, count, MPI_INT64_T, 0,
    MPI_COMM_SELF);

// Add fluid node indices to displacement array
for (int_t i = 0; i < count; i++) {
 if (local_indexes[i] >= 0) {
   displacements[node_count++] = local_indexes[i];
 }
}

// Create indexed block for the border
MPI_Type_create_indexed_block(
 node_count, 1, displacements, lattice_pt, new_type);

MPI_Type_commit(new_type);
```

Listing 4.8: The code for creating a border type using indirect addressing based on the full-size border

Ghost cells

Creating the MPI types needed to send an internal border to a neighboring process requires minimal changes in the underlying data structures describing the domain. However, the receiving process also needs the necessary information to *receive* the border cells from a neighbor process. Since each process only contains information about its own part of the simulation domain, we cannot immediately use the same process as with the internal border nodes.

In order to create the necessary type information required to receive ghost cells, we also add ghost cells to the domain map, such that each process also has information about all neighboring process's borders. Similarly to the naive method, we exchange borders of the domain map during the domain setup phase so that we can utilize the same method as for internal nodes. The exact displacements of fluid nodes in each MPI_Type is irrelevant for successful synchronization, meaning we don't need to worry about the memory layout of the density array matching that of its neighbors and only need to make sure the number of ghost cells in the index mapping table matches that of its neighbors.

With separate MPI types for all borders, we can keep the same logic for the border exchange and just replace the MPI types used in the MPI_Sendrecv calls as indicated in Table 4.2.

Table 4.2: Changing the types used for border exchange to support indirect addressing

```
MPI_Sendrecv(density[1],
             1,
--           row_dep_plane,
             NB_N,
             0,
             density[1],
             1,
--           row_dep_plane,
             NB_S,
             0,
             comm_cart,
             MPI_STATUS_IGNORE);
```

```
MPI_Sendrecv(density[1],
             1,
++           mpi_t_plane_N,
             NB_N,
             0,
             density[1],
             1,
++           mpi_t_plane_S_ghost,
             NB_S,
             0,
             comm_cart,
             MPI_STATUS_IGNORE);
```

4.3.2 Loop optimizations

As a consequence of the indirect addressing, a further optimization we wish to explore is changing the outer loop structures in both the collision and streaming kernels and evaluate the effect on the performance.

With direct addressing, the outer loops of both kernels iterate over all points in the domain and perform a check on whether the point is a solid point, skipping over the point in this case (Listing 4.9, line 12). When we switch to indirect addressing, we already have a complete list of all fluid points in the domain, essentially pre-computing the existing conditional. By iterating over the fluid points instead of the index array, we can remove

the conditional and save a lot of unnecessary branching in the program's execution. An interesting effect that we might be able to observe is how this optimization affects the cache performance of the streaming kernel. With the indirect addressing, we expect the hardware to much more effectively cache the fluid nodes compared to the naive approach with solid nodes flooding the cache, especially for more porous simulation environments. However, with this loop optimization, we might be able to observe even better cache performance since we are iterating over the list directly.

A downside of this optimization is that we no longer have implicit information about the location of the fluid node through the three loop variables. This means that we need to store the indices of the nodes in a separate structure, p_coords in Listing 4.9. Although these values are relatively small compared to the lattice point structure ($3 \cdot 8bytes = 24bytes$ compared to $27 \cdot 8bytes = 216bytes$), the extra memory load could affect the program negatively.

As this is a simple change to the code, we wish to explore how this will affect the performance of the individual kernels and the application as a whole.

```
#ifdef OSTREAM_LOOP_INDIRECT
  for (int_t n = 0; n < fluid_node_count; n++) {
    int_t k = p_coords[n].z;
    int_t i = p_coords[n].y;
    int_t j = p_coords[n].x;
#else
  for (int_t k = -1; k < depth + 1; k++)
    for (int_t i = -1; i < height + 1; i++)
      for (int_t j = 0; j < width; j++) {
#endif
    for (int_t d = 0; d < D_COUNT; d++) {
      if (P_index(i, j, k) < 0)
        continue;
```

Listing 4.9: Loop optimizations in the stream kernel. The green highlight shows the direct looping. The original looping is highlighted in red.

Chapter 5

Experiment

In this chapter, we describe how we conduct the experiments using the proxy application. We describe and show the different simulation environments that are considered and used, how they relate to the real world, and how we create or obtain these environments. We then briefly describe the test environment, including the HPC platforms, libraries, and compiler versions and options. Lastly, we describe the variations of the proxy applications used in the experiments, their differences, and what we expect to see when applying them to various geometries and HPC environments.

5.1 Simulation domains

In this thesis, we have performed experiments on several existing models and simulated environments. The base experiment uses a common *Poisseuille flow* geometry, as shown in Fig. 6.1a. This model consists of a rectangle with walls on all sides in the xy plane. The model is open on both sides of the flow direction, such that fluid density can wrap around to simulate increasing flow. This type of environment has no solid points, and all ranks are fully saturated and balanced. An example simulation of this environment is included in Fig. 5.1.

In order to simulate varying degrees of porous domains in likeness to porous rocks in reality, we also create a rough model to approximate these environments. We generate the models by filling the domain with solid cylinders in the xy plane. These cylinders are added at pseudo-random positions until the suitable porosity is reached. For lower porosities, we also add xy planes of solid points along the z axis to keep a good balance of solid points in the domain. Fig. 5.2 include renders of the 10% and 80% porosity domains generated with this method.

All generated models are pre-generated and saved to a binvox file such that all experiments use the exact same geometry. We don't evaluate or consider the correctness of these models to real-world scenarios, as these models aim to assess the proxy applications on *good enough* balanced domains with varying degrees of porosity. Table 5.1 includes the total, rank-minimum, and rank-maximum porosity for all the generated domains used in the

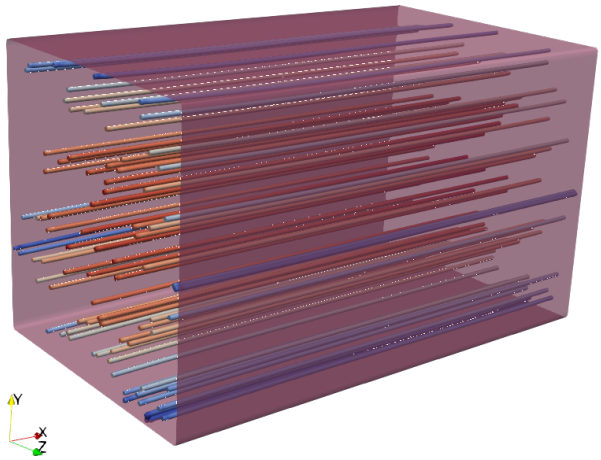


Figure 5.1: Example Poisseuille flow domain. The fluid flow is generated at the inlet (left) and flows in the x direction. The fluid wraps around the edge on the x-axis.

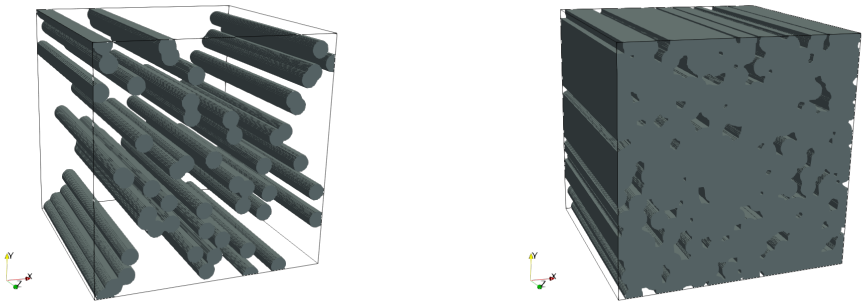
experiments. We evaluate the difference in porosities to be small enough for the purpose of these experiments.

Table 5.1: Porosity for the tested domains with 16 ranks. Min and max porosity represent the minimum and maximum porosity among all ranks for the respective domain.

Domain type	Size	Target	Porosity		
			Total	Min	Max
RCA	400		1.2%	0.0%	3.6%
	1024		1.0%	0.0%	3.0%
Generated	400	10%	10.2%	8.9%	11.3%
		20%	18.2%	9.7%	24.6%
		30%	29.8%	24.7%	34.5%
		40%	41.0%	35.7%	45.7%
		50%	47.6%	32.6%	58.6%
		60%	60.9%	52.6%	72.8%
		70%	69.2%	60.4%	77.0%
		80%	82.5%	76.3%	87.3%
		90%	91.9%	88.4%	96.2%
		100%	100%	100%	100%

In the extreme case, we wish to experiment on very sparse domains that contain a very small amount of fluid nodes compared to solid points, as well as being less balanced. Blood flow simulations are a great candidate for this type of geometry. We use 3D models of a patient-specific right coronary artery (RCA)[31]. This model is converted from vector images to cubic voxel data using the binvox program. We use two sizes of the RCA: 400x400x400 and 1024x1024x1024. An image of the RCA model is included in Fig. 5.3. These environments have very low porosities, with the RCA model at ca. 1% total porosity in addition to being unbalanced as a contrast to the generated domains. As the few fluid points in this model are highly clustered, we expect many of the processes to have no work as an effect. In these cases, it is useful to evaluate the rank balance and process topology in order to achieve better balance among processes.

Although the voxel files are cubic, we perform an operation step to find the bounding box of the fluid domain in the initialization step. This means that the real width of the simulated environment might be smaller in one direction. The reported porosities are obtained from this bounded domain.



(a) Generated domain with 80% porosity

(b) Generated domain with 10% porosity

Figure 5.2: Generated domains with different degrees of porosity. The images show the solid obstructions in the domains.

Figure 5.3: 3D model of the patient-specific right coronary artery used in the simulations.

5.2 Test environment

For the tests conducted as part of this thesis, we have used three different supercomputers all located in Norway: Idun, Betzy, and Fram.

The job scheduling on Idun gives shared access to compute nodes, meaning several users are allocated a fraction of resources on the node and can run processes simultaneously. Fram and Betzy provide regardless of the share of resources that is in use.

5.2.1 Idun

Idun is a GPGPU-enabled computing cluster operated by the high-performance computing group at NTNU. Idun consists of a mix of Intel Xeon and AMD Epyc 75F3 processors. The compute nodes are connected using an InfiniBand interconnect in a tree topology[30]. The Idun cluster has been used for the development and validation of the proxy applications.

5.2.2 Betzy

Betzy is the newest and largest supercomputer made available by the Norwegian Research Infrastructure Services (NRIS). Betzy is a BullSequana XH200 computer located at NTNU in Trondheim. The supercomputer has a peak floating point performance of 6.2 Petaflops provided by 1344 compute nodes connected in a dragonfly topology[16] using an InfiniBand HDR 100 interconnect. Details on the system's specifications are included in Table 5.2.

Table 5.2: Details of the Betzy supercomputer

Compute nodes	1344
CPU type	AMD® Epyc™ 7742 @ 2.25GHz
CPU cores per node	128
Memory per node	256 GiB
Interconnect	InfiniBand HDR 100 Dragonfly+ topology

5.2.3 Fram

Fram is another supercomputer provided by NRIS. It is a Lenovo NeXtScale nx360 with a peak floating point performance of 1.1 Petaflops. Fram is located at UiT, the Arctic University of Norway, and is configured with 1004 compute nodes connected in an island topology. The system configuration of Fram is included in Table 5.3.

5.2.4 Software

All benchmarks use the same software versions across all computing environments. The benchmarks were performed using the Intel compiler toolchain, version 2020b, and Intel MPI.

Table 5.3: Details of the Fram supercomputer

Compute nodes	1004
CPU type	Intel® Xeon® E5-2683v4 @ 2.1GHz
CPU cores per node	32
Memory per node	64 GiB
Interconnect	InfiniBand island topology

Table 5.4: Software versions and compiler settings

Compiler flags	-g -O3 -Wall -Wextra -fopenmp -lm -lnetcdf
Compiler version	Intel 2020b
MPI version	Intel MPI 2020b
NetCDF version	4.7.4 (compiled with mpiicc 2020b)

5.2.5 Measurements

To measure the performance of the applications, we use MPI's built-in MPI_Wtime method. We collect four measurements in the main loop of the application, before and after each of the main parts of the application (*collide*, *stream*, and *border exchange*). This allows us to calculate the time difference before and after each part and, thus, the duration. The relevant code is highlighted in Listing 5.1.

```
1  /* Time integration loop */
2  for (int_t iter = 0; iter < max_iter; iter++) {
3      start_time = MPI_Wtime();
4      collide();
5      collide_time = MPI_Wtime();
6      border_exchange();
7      exchange_time = MPI_Wtime();
8      stream();
9      stream_time = MPI_Wtime();
10     itertime = MPI_Wtime();
11
12     collide_total += collide_time - start_time;
13     exchange_total += exchange_time - collide_time;
14     stream_total += stream_time - exchange_time;
```

Listing 5.1: Time measurements in the main loop using MPI_Wtime

Each process maintains its own measurements for the entire duration of the application runtime. When the simulation is done, we use collective MPI file write operations to write each process' measurements and statistics to a single CSV file for further processing. In addition to timing data, we collect data on the individual process' porosity and emit some data on the size of the process' borders.

5.3 Benchmarks

In order to gather valuable insights into the performance and behavior of the variations of the proxy application, we perform a large set of benchmarks. Table 5.5 presents the four variations of the application we have used when conducting these tests. To evaluate the variations in reference to different geometries, we perform simulations of fluid flow on porous domains of different porosities as well as the RCA model. We run simulations on different sizes of the domains. In order to evaluate the scalability of the programs and the selected geometry, the benchmarks are also performed with varying process counts.

Table 5.5: Identifiers for the different variations of the D3Q27 proxy application used in the benchmarks and analysis

Base	Baseline implementation as described in Section 4.1
IA	Optimized with indirect addressing. Only data structures and the border exchange have been modified compared to the baseline.
IALO	Same as IA, with loop optimization applied to both the stream and collide kernels as described in Section 4.3.2.
IALO 1D	Same as IALO, with a different topology. This version partitions the process topology only in the y direction.

5.3.1 Optimizations

Indirect addressing

The effects of the optimizations are expected to have a considerable impact on the performance, especially for lower porosity models. This is due to the much lower memory requirements and increased data locality resulting from the removal of "empty data" from the density arrays, in the form of solid nodes. For a cubic simulation domain 400 nodes wide, the memory requirements of just the density arrays adds up to a very large amount:

$$400^3 \cdot 27 \cdot 8B \cdot 2 = 27.6GB$$

For even larger models, such as with 1024 node width, the number grows to $462GB$. Not only will this hurt memory performance for less porous environments, but it also limits the feasibility of simulating large environments and smaller environments in great detail. With the indirect addressing scheme, the memory requirements of the density arrays are directly proportionate to the porosity, meaning the same 400-wide environment only requires $276MB$ of memory for the density arrays. We expect this to have a considerable impact on cache performance as well as the ability to perform simulations at this size on available computing platforms.

Loop optimizations

As detailed in Section 4.3.2, the loop optimizations are enabled by changing the outer loop structure in the stream and collide kernels, iterating over the density array directly and

skipping one of the indirect accesses. In addition, this optimization will skip many empty loop iterations (iterations operating on solid nodes). This optimization is not expected to have a considerable impact on larger porosity domains ($> 60\%$), however this loop structure might allow the CPU to more effectively cache and predict the memory access pattern in addition to the fewer iterations making a larger difference for lower porosities and sparse geometries. We discuss the effect of the loop optimizations in Section 6.2.3.

5.3.2 Scalability

It is interesting to evaluate the scalability of the optimizations compared to the baseline program on the different geometries. The LBM applied to Poiseuille flow simulations, common in benchmarks, benefits from strong scaling. However, we expect resource utilization to worsen as we transition to less porous environments with fewer fluid nodes. In this experiment, we only evaluate the strong scaling capabilities of the programs, using the same simulation geometry and dimensions on all tests, and do not consider weak scaling.

We measure the scalability of all versions on the following geometries:

- RCA model at 400 node width
- RCA model at 1024 node width
- Generated domain with 50% porosity at 400 node width
- Generated domain with 10% porosity at 400 node width

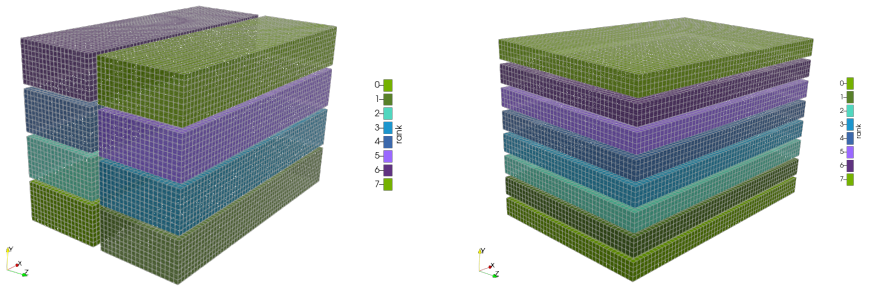
With these tests, we aim to see how the different optimizations applied to the above domain geometries affect the program's ability to scale to many processes. The benchmarks in this test use a constant 8 CPU cores per rank (OpenMP threads) and will allocate as many ranks per node as possible. For Betzy, this means a maximum of 16 ranks per node and a maximum of 4 ranks per node for Fram.

5.3.3 Porosity

The porosity experiments are conducted by running all variations of the proxy application on simulation domains of different degrees of porosity. We run the benchmark on all the generated models listen in Table 5.1. All tests are conducted on cubic 400 node-wide domains. For these tests, we have opted to use 16 ranks, as this is a middle-ground in terms of process count and will give all ranks a working size of 8 million nodes.

5.3.4 Topology

Unless specified, the benchmarks use a cartesian process topology with automatic dimensions. This means that it is left up to the MPI implementation to choose. In most implementations, and the version used here, MPI will attempt to balance the dimensions in both directions as shown in Fig. 5.4a. The IALO 1D program also uses MPI's cartesian topology but with a limit of 1 for the width of the topology in the z direction, as shown in Fig. 5.4b. In this configuration, the processes only have two borders (north and south) compared to the four in the normal configuration.



(a) Default cartesian topology

(b) 1D topology

Figure 5.4: A simulation domain split between 8 ranks with default cartesian topology in Fig. 5.4a and 1-dimensional topology in Fig. 5.4b.

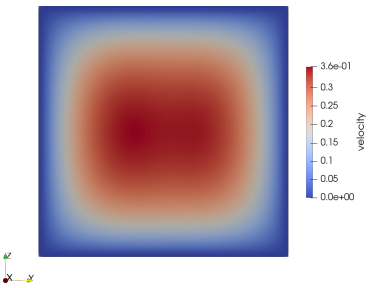
Chapter 6

Analysis

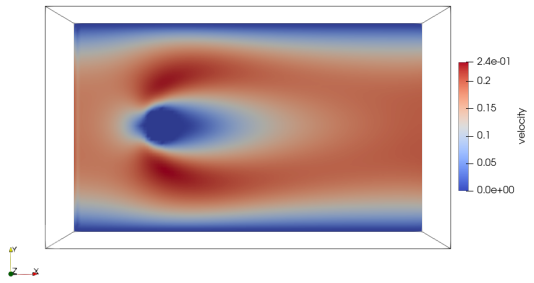
In this chapter, we analyze the results of the experiments outlined in Chapter 5. Firstly, we validate the correctness of the application in a Poisseuille flow simulation and a generated porous model. We then compare the overall performance of the indirect addressing, loop optimizations, and topology applied to the RCA model and select generated porous environments. Further, we explore the proxy applications' ability to scale to many processes and see how the optimizations and different geometries affect weak scaling. Lastly, we will discuss in more detail how the porosity of a simulated domain affects the program and the optimizations before we look at the differences in the default and 1-dimensional topology applied to the geometries.

6.1 Validation

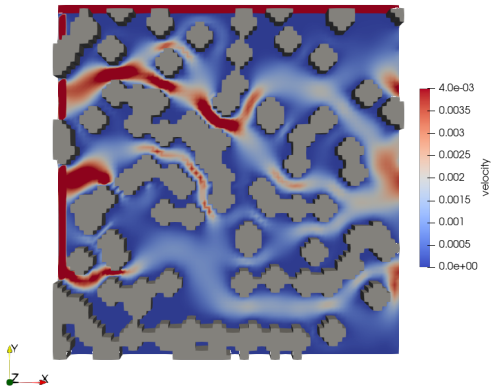
As the proxy applications used in the experiments only implement a simple border condition that wraps the density distribution around one axis, we validate the applications where this condition is most applicable. For the simplest case, we validate the application using a Poisseuille flow. Figs. 6.1a and 6.1b shows cross-sections of the xy and xz plane in a Poisseuille flow simulation with the inlet at the minimum of the x-axis. We observe that the velocity of the fluid is highest in the center in the flow direction, with the velocity gradually slowing towards the walls of the container. This is in line with what we should expect from a Poisseuille flow simulation. This same simulation is also pictured in three dimensions in Fig. 5.1, with a cylindrical obstruction the length of the z axis Here we see that the fluid is pushed above and below the cylinder, with an area of still particles behind the obstruction. We also validate the fluid flow in the generated porous domains, where Fig. 6.1c shows a cross-section of the fluid velocity on such a domain. This simulation also shows similar results to what we can see in previous work. The fluid has high velocity in the narrow channels near the inlet and low velocity near the outlet.



(a) Cross-section of Poisseuille flow simulation using the $D3Q27$ model



(b) Cross-section of the xy plane showing flow over a cylindrical obstruction in the z direction.



(c) Flow in an 80% porosity generated domain.

6.2 Performance optimizations

6.2.1 General performance analysis

In this section, we will look at the general performance differences of the various optimizations applied to the proxy application, and how these optimizations vary in different domain geometries.

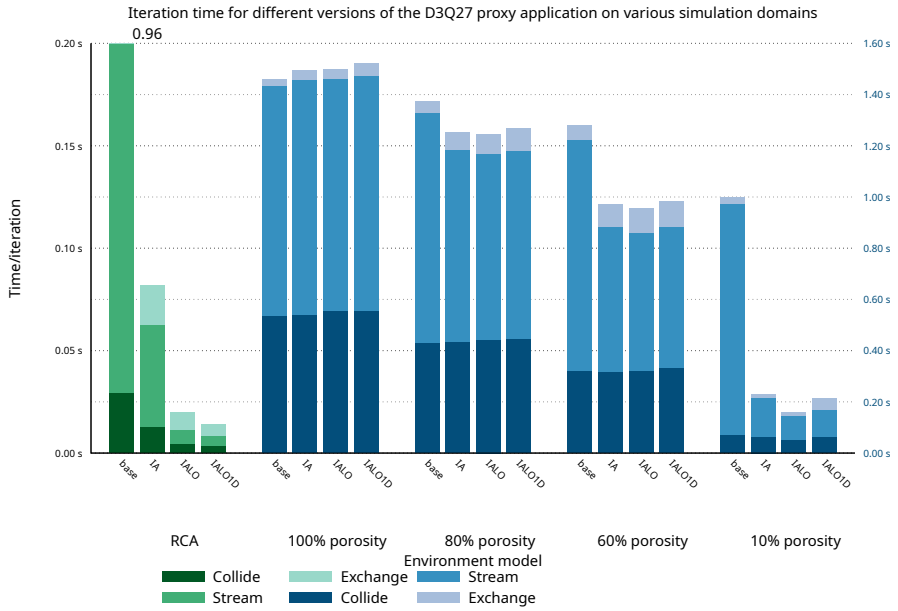


Figure 6.2: Overview of the performance of the various optimizations applied to a subset of the tested models (RCA and various generated porous domains). All models have a height, width, and depth of 400 nodes. The RCA measurements follow the left y-axis. The remaining measurements follow the right axis. The tests were performed with 4 ranks on the Fram supercomputer.

Evaluating the performance of all variations on different domain geometries of the same size, we can see that all optimizations have a noticeable effect on non-filled geometries. In Fig. 6.2, we have plotted the per-iteration runtime of all parts of the application (stream, collide, and exchange) for all variations on several geometries. The optimizations have no benefit for the performance on the 0% porosity geometry (Poisseuille flow), and in fact, contribute to a negative impact on the overall performance.

When we decrease the porosity of the geometry, we see that the optimizations start to have an effect. Already at 60% porosity, both the indirect addressing and loop optimizations decrease the runtime of the stream kernel especially, performing slightly worse with the less memory bound collide kernel. The speedup of the IA program is already at **1.3x** at this relatively high porosity.

Decreasing the porosity even more, we start to see a much more marked effect, with the 10% porosity geometry benefiting strongly from both indirect addressing and loop op-

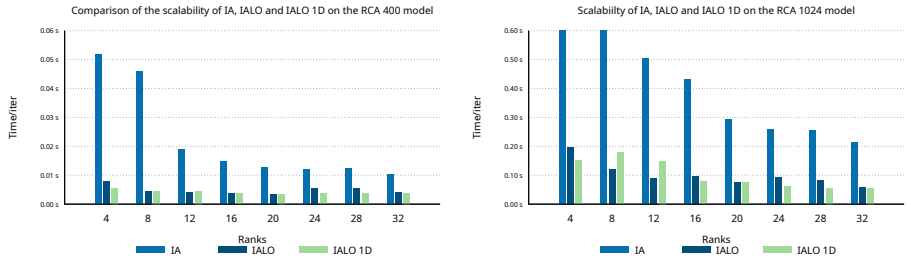
timizations in all parts of the program. At this stage, the IA program has a speedup of **4.8x** over the baseline.

For the very sparse and unbalanced RCA model, we start to see how these optimizations can greatly benefit the LBM. With 4 ranks, the speedup of IALO over the baseline is at **44x**. With the RCA model, the collide kernel has minimal impact on the performance of the base application, with the streaming kernel contributing to the bulk of the processing time. Although the baseline implementation is very naive in streaming all particles regardless of the type, this shows how a geometry-unaware LBM application can be ill-suited for blood-flow simulations and similar applications. With this geometry, we also see how a topology that is less balanced, matching that of the geometry, can be used instead of the more balanced 2D cartesian topology to further increase the performance.

6.2.2 Indirect addressing

We see that the sparse geometries of the blood flow simulations strongly benefit from an indirect addressing scheme. For the RCA model we have tested, only changing the memory layout and border exchange to utilize indirect addressing results in a speedup of **11.7x** over the baseline. Even though the baseline could be further optimized to consider a large solid/fluid node ratio other than with indirect addressing, this result shows that indirect addressing is a highly effective and simple way to optimize the LBM for sparse geometries.

In addition to sparse geometries, we have also observed that indirect addressing can be a very effective method to speed up the LBM for balanced porous domains such as porous rocks. At 10% porosity we see a speedup of over **4.4x** over the baseline. Considering that real porous rocks such as sandstone have been shown to have a porosity between 8–25%[3, 14], an indirect addressing optimized LBM can also be useful for porous rock simulations.



(a) RCA 400 **(b)** RCA 1204

Figure 6.3: Comparison of the effect of indirect addressing (IA) and indirect addressing with loop optimizations (IALO) on the scalability of the RCA model at 400 and 1024 node widths.

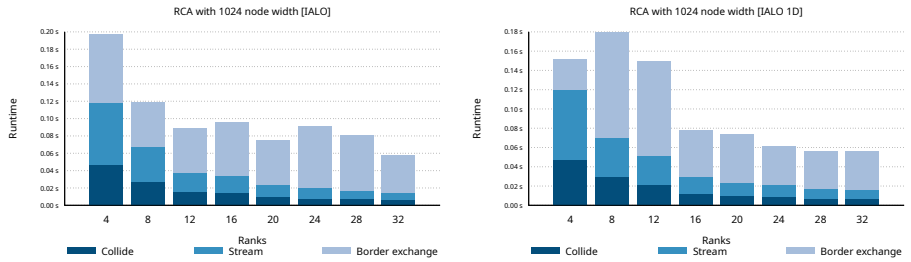
6.2.3 Loop optimizations

The loop optimizations (IALO) have relatively little impact for higher porosity domains, as we see in Fig. 6.2, with comparable performance for the 60% porosity model, and a speedup of **1.45x** for the 10% porosity model. However, for the RCA 400 model, IALO

achieves a speedup of **4x** over IA with 4 ranks. The results are even better for the larger RCA 1024 model, with a speedup of **6.6x** at 4 ranks. Figs. 6.3a and 6.3b shows the same trend for both the 400 and 1024 wide RCA models. Where the speedup is substantial for most ranks, although not as large for many ranks (4.6x for RCA 1024 at 16 ranks). These results show that the performance gains resulting from the optimized loop structure using indirect addressing have a large effect over a traditional loop structure, especially for sparse domains like the RCA model.

6.3 Scalability

6.3.1 Blood flow simulations



(a) Average runtime of optimized LBM on the RCA model with 1024 node width. Measured on Fram.

(b) Runtime of 1D topology LBM on the RCA model with 1024 node width. Measured on Fram.

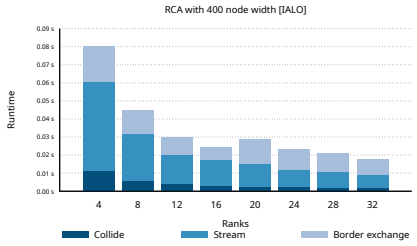
Figure 6.4: Per-iteration runtime of IALO and IALO 1D on the 1024-wide RCA model.

RCA 1024

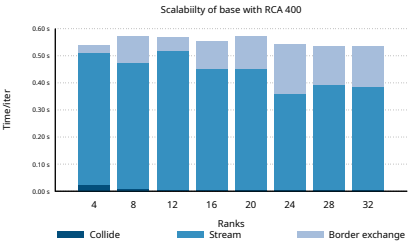
The IALO optimized version scales well with the RCA model on large geometries, with a **1.7x** speedup from 4 to 8 ranks using the RCA 1024 model. From 8 to 16 ranks, the speedup is reduced to **1.2x**, with even less speedup for increasing rank counts. Fig. 6.4a shows the iteration time for the 1024-wide RCA model. We observe that most of the gains are in the stream and collide kernels, with the time spent on border exchange increasing in many cases, despite smaller borders. At 32 ranks, the speedup of computation combined (stream & collide) over 4 ranks is ca. **8x**, while the collision kernel sees a **1.8x** speedup. As the communication takes up 2/5 of the time at 4 ranks, using a maximum speedup of 2x for the communication, a theoretical 4/5 of the application benefits from increased parallelism. Using Amdahl's law, this result limits the theoretical speedup of the application to roughly **5x** for the RCA model at this size. Although an approximation, this provides a basis for the scalability in this specific case.

Due to this limitation, at 20 ranks, the benefit of increasing the rank count over 16 is minuscule. We also see that the topology and domain distribution greatly affects the impact of the border exchange, with some rank counts and partitions being very unfavorable for the RCA model. With 12 ranks, the border exchange uses 0.05s per iteration on average,

while for 24 ranks it uses 0.07s. This makes the iteration time *slower* at 24 ranks compared to 12 ranks. As our measurements don't take into account the effective time used for border exchange, but also include synchronization wait time, we don't know how much of this time is spent waiting compared to communication.



(a) Average runtime of optimized LBM on the RCA model with 400 node width. Measured on Fram.



(b) Average runtime of baseline LBM on the RCA model with 400 node width. Measured on Betzy.

Figure 6.5: Per-iteration runtime of baseline and IALO on the 400-wide RCA model.

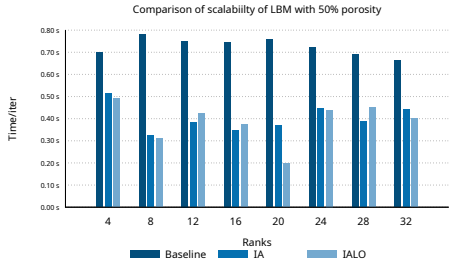
RCA 400

Looking at smaller models like the 400-wide RCA model in Fig. 6.5a, we see a similar result. The distribution of time among the stream, collide and border exchange is quite different for this problem size, occupying 58%, 13%, and 29% respectively at 16 ranks. Comparatively, the distribution is 21%, 15%, and 64% at 1024 nodes. The lower fraction of time spent on the less scalable border exchange for the smaller model translates to slightly better scalability. Where the speedup from 8 to 16 ranks was **1.2x** for RCA 1024, we see a speedup of **1.8x** for the RCA 400 model. Performing the same rough analysis using Amdahl's law, with 93/100 of the application benefiting from parallelism, this smaller model has a theoretical speedup limit of **14x**. This rough analysis shows that the reduced partition of time taken up by the less parallelizable border exchange provides the proxy application with more room to scale for small problem sizes, at least for the rank counts tested here.

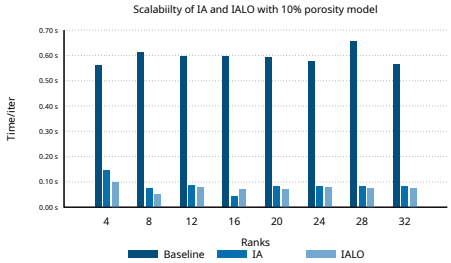
We will look closer at the partitioning and workload in Section 6.5, but since the rank partitioning and communication pattern has a large impact on the performance for sparse geometries, especially for larger problem sizes, it is clear that a more efficient process partitioning and communication model is required to take advantage of increased resource availability.

6.3.2 Balanced porous geometries

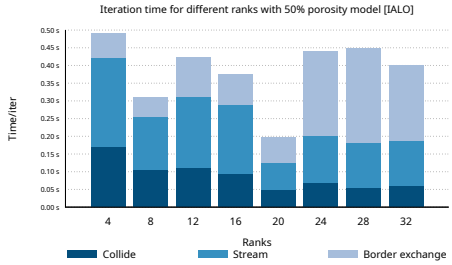
When we look at balanced, porous geometries, we observe that the scalability is hard to quantify based on rank count alone. In Fig. 6.7, we see that in a porous domain with half of the nodes being solid, the collide and stream kernel has a speedup of **2.2x** at 32 ranks compared to 4 ranks. This is not only over 3.5x worse than the RCA model at the same



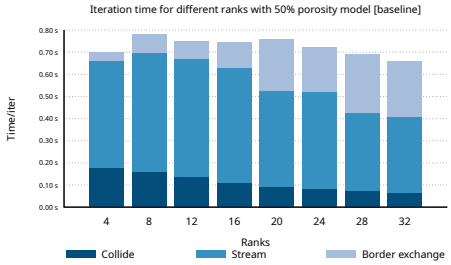
(a) Scalability of the versions with the 50% porosity model



(b) Scalability of the versions with the 10% porosity model



(a) Runtime of IALO



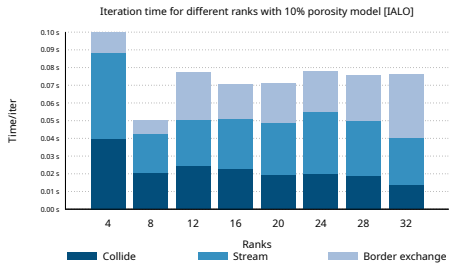
(b) Baseline runtime

Figure 6.7: Performance of 50% porous geometries with up to 32 ranks measured on Betzy.

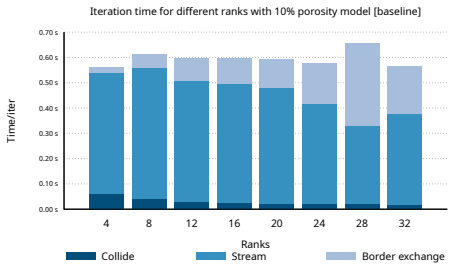
size, but we see more variability in the performance gains at different rank counts, pointing to the fact that the specific process topology and rank partitioning is an important factor for these geometries despite smaller borders. For 16 ranks, we measured the size of a single border to contain between 13000 and 23000 fluid nodes. With these large borders, a single message amounts to between $2808kB$ and $4968kB$, compared to $1260kB$ — $2457kB$ for 32 ranks. This results in overall smaller messages with 32 ranks, but the border exchange still sees a **0.4x** speedup (almost 2x slower) at 32 ranks over 16 ranks.

Compared to the RCA model, the stream kernel does not scale as well with this geometry. Where we see a **6.5x** speedup for the RCA model, the 50% geometry only sees a **2x** speedup from 4 to 32 ranks in the stream kernel. This worse scaling is almost identical in the collide kernel as well (6.8x vs 2.8x for RCA and 50% respectively).

We also see a large dip in the iteration time at 20 ranks for this geometry, where we saw an increase in time for the RCA model. This is expected for the RCA model, as its measurements on Fram causes the node count to jump from 4 to 5 nodes at 20 ranks, increasing the latency between nodes. This is not the case for the porous geometry tests, as all tests are run on 4 nodes.



(a) Runtime of a 10% porosity model on betzy



(b) Baseline runtime

Figure 6.8: Performance of 10% porous geometries with up to 32 ranks.

10% porosity

We have seen that the optimizations we have applied start to take considerable effect at around 10% porosity. Looking at the scalability at this stage, we observe that it is similar to the 50% geometry. Compared to the 50% porosity geometry however, the fraction of time spent on the border exchange is more consistent for the 10% geometry while also being generally smaller (2–26%). With 16 ranks, this domain reaches a maximum border size of ca. 4500 fluid nodes, with a size of $4500 \cdot 27 \cdot 8B = 972kB$, roughly 5 times smaller than than at 50% porosity. However, even though the border exchanges provide less of an impact, both the streaming and collision kernels still see minimal performance gains with increasing ranks also at this stage.

In general, we see that the porous domains, at all tested porosities, scale much poorer than the sparse RCA model at these rank sizes.

6.4 Porosity

In Fig. 6.9, we compare the per-iteration time for the different versions of the proxy application on porous geometries from 100 — 10%. Similarly to what we have already seen in Fig. 6.2, the optimized version scales much better with decreasing velocity than the baseline. For this experiment, we see that the baseline has a median speedup of **1.04x** for every 10% decrease in porosity with a total speedup of **1.5x** from 100% to 10% porosity. Comparatively, the IALO version has a median speedup of **1.13x** per 10% decrease and a **9x** total speedup from 100% to 10% porosity. For the optimized version, we see speedup increased with decreasing porosity, and this at a nearly linear rate, with the speedup increasing at the same rate as the decreasing porosity. This result shows that the indirect addressing scheme is highly effective for low porosity geometries and that for balanced workloads we can expect the performance of the application to be proportional to the number of fluid nodes.

Looking at how the time is distributed among streaming, collision, and border exchange in Fig. 6.10, we see that all parts of the program see a speedup for decreasing porosity. For each 10% decrease in porosity, the stream and collide kernels see a median speedup of 1.24x and 1.21x respectively, and the border exchange 1.17x. This shows that all parts of the

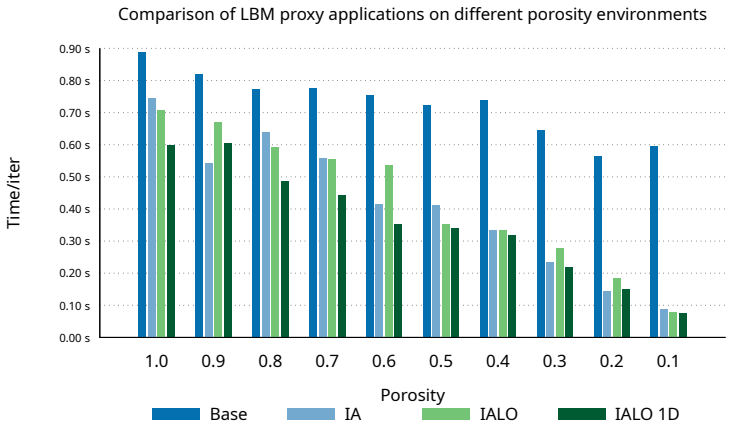


Figure 6.9: Comparison of the variations of the proxy application on varying porosities. Running with 16 ranks on a 400x400x400 balanced geometry.

application are susceptible to speedups when we apply the LBM to less porous geometries. Given an ideal programming model that is not dependent upon the specific geometry of the simulation domain, as is the case in this proxy application, we could expect this trend to apply to sparse domains as well.

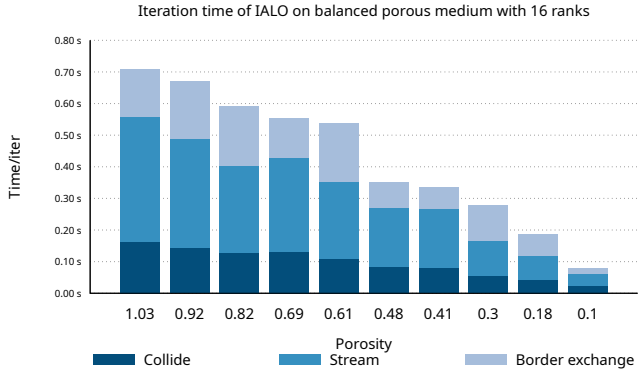


Figure 6.10: Per-iteration time for IALO with 16 ranks on 400 wide generated balanced geometries of varying porosity.

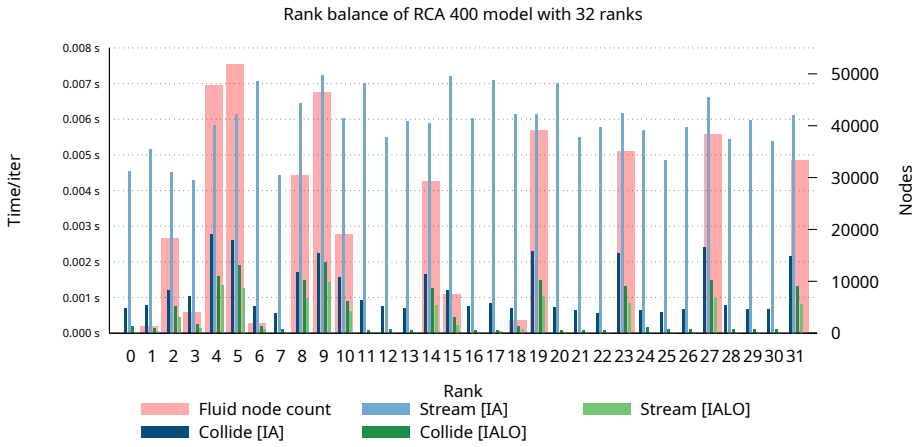


Figure 6.11: Rank balance for RCA-400 model

6.5 Topology and rank balance

6.5.1 RCA geometry

Looking more closely at how the work is balanced between nodes, we can reason more about how the application scales with different process topologies and domain configurations. Fig. 6.11 shows the average time per iteration for each rank when running the application with a normal cartesian process topology on the RCA model with a size of 400 nodes. The red bars show the number of fluid nodes for the rank, while the green and blue bars plot the time per iteration of the collide and stream kernels with and without loop optimizations respectively.

Firstly, we see how this particular geometry fails to take advantage of the large process count with a normal cartesian distribution. Since the fluid nodes are highly localized and non-uniformly spread in the geometry, 20 of the 32 ranks end up with negligible working sets, with the remaining 12 performing the bulk of the calculation. We have visualized rank 9's relation to the geometry in Fig. 6.12, which is among the ranks with the largest working set. As the rank's local domains are very small at this large rank count, we see how many of the ranks are allocated a partition of the domain that does not contain any fluid nodes. This shows that a uniform, cartesian process topology may not apply well to sparse geometries, especially at large rank counts.

Fig. 6.13 shows the same experiment, but with a one-dimensional process topology. Here we see that the one-dimensional topology can achieve much greater balance than that of a naive two-dimensional topology. As we have seen in Figs. 6.3a and 6.3b, for many ranks, the increased work balance achieved with this topology can result in a slight increase in performance, at the cost of increased synchronization for other topologies.

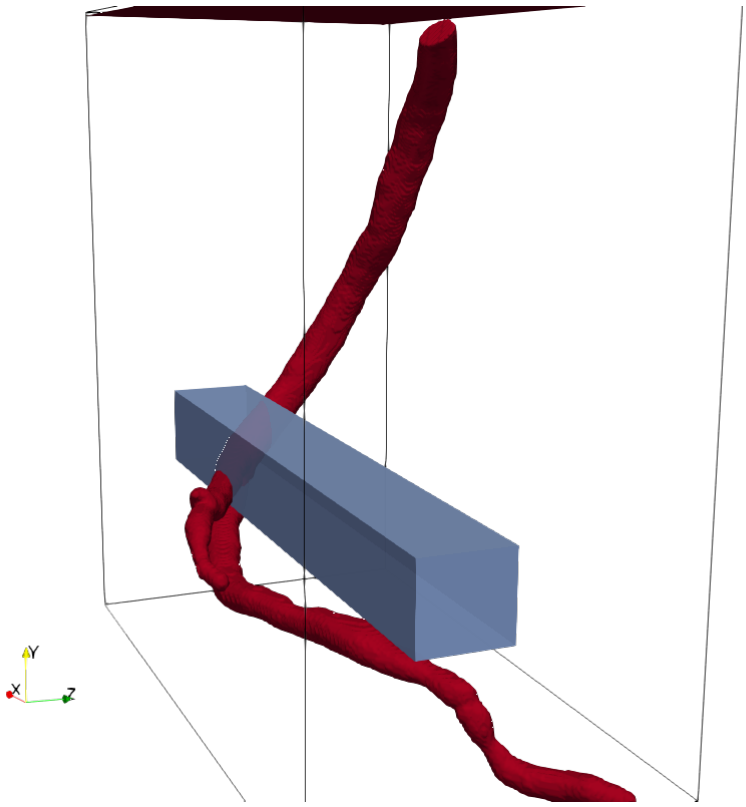


Figure 6.12: Outline of the working set of rank 9 on the RCA model

6.5.2 Balanced porous geometries

As we have seen in earlier sections, the one-dimensional topology consistently shows worse performance for the balanced porous geometries, with a 30% performance penalty for the 10% geometry at 4 ranks.

In Fig. 6.14, we plot the same figure for the 10% porosity model again with 32 ranks. With this geometry, the ranks are much more balanced, with a standard deviation of 39 000 fluid nodes and a median of 211 000. Compared to the RCA model's 18 000 standard deviation and 600 median, the partitioning of the fluid nodes among the ranks is much more balanced for the porous domains. As a result, the computation time for each rank is also much more similar. However, as we saw in the discussion of the scalability (Section 6.3.2) of this particular configuration, the increased balance does not mean that the scalability is better. The portion of time spent on border exchange and waiting for synchronization is also higher for the balanced porosities despite the better balance.

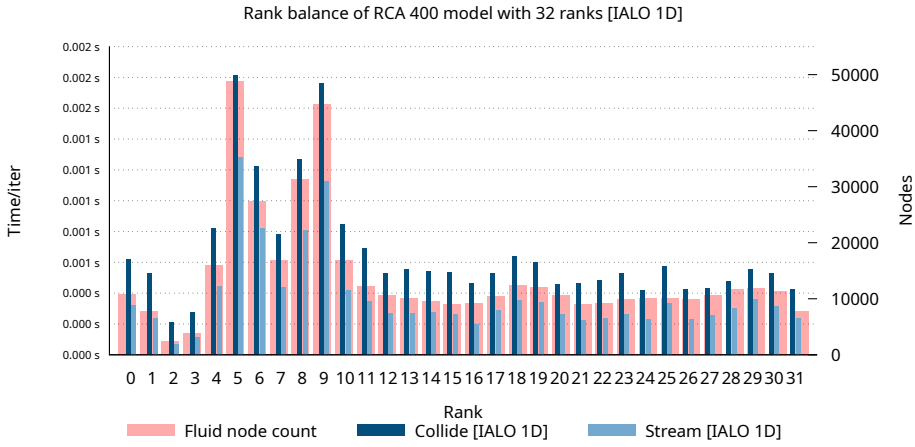


Figure 6.13: Rank balance for RCA-400 model with 1D topology

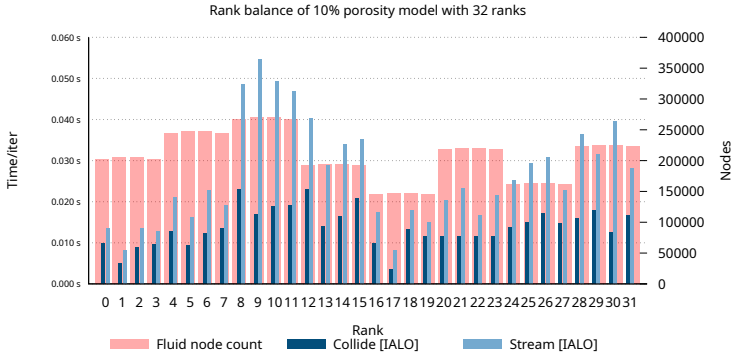


Figure 6.14: Rank balance for 10% porosity model with default topology.

Chapter 7

Conclusion

In this thesis, we have developed a proxy application for the $D3Q27$ LBM method and analyzed geometry-specific optimizations on blood-flow simulations and flow through porous rocks.

The proxy application uses MPI and OpenMP to utilize available CPU resources in HPC environments, and we have conducted tests on the Fram and Betzy supercomputers. We have implemented indirect addressing to reduce memory requirements and memory load throughout the application, in addition to creating an effective way to leverage the indirect addressing memory layout with ghost cell patterns and MPI. By only transferring fluid nodes, we show that the border exchange can achieve close to 10% speedup for respective 10% decreases in porosity. We also show that the stream and collide operations achieve up to 20% speedup with 10% decreases in porosity using indirect addressing and loop optimizations. Combined, the speedup of the optimized LBM application is measured to be proportional to decreasing porosity in the simulation domain.

Using 3D models of patient-specific coronary arteries, we have evaluated the effect of indirect addressing and subsequent optimizations to loop structures on sparse geometries, such as those found in blood-flow simulations. We have found that these optimizations are very well suited for these geometries, achieving over **12x** speedup compared to the baseline with indirect addressing and **48x** with the optimized loop structure.

We have also evaluated the differences in performance when using a default cartesian process topology and a topology that achieves better balancing of work among ranks. We show that a one-dimensional topology achieves similar performance for porous rocks and can achieve a speedup of up to **1.13x** for blood-flow geometries at specific rank sizes.

7.1 Future work

7.1.1 Communication model

In this proxy application, we have not implemented overlapping communication and computation. This means that several ranks will needlessly wait for border cells when they can compute cells that are independent of ghost cells. We have seen that the communication overhead can be quite high for the LBM simulating porous domains, especially for many ranks. Implementing overlapping communication and computation can considerably affect the application's performance and ability to scale to more processes.

Similarly, none of the partitioning schemes for the MPI ranks used in this thesis is particularly optimal for sparse geometries. Exploring other partitioning techniques to utilize the available resources better can achieve better scaling and significantly impact the performance at even a few ranks. Due to the sparse artery domain in blood-flow simulations, achieving good balance even for smaller rank counts can require non-default partitioning techniques.

We also don't take into account empty messages using indirect addressing. This means that, although the data is empty, we perform unnecessary synchronization and are affected by the added latency of the empty transfers. A simple target for further optimization of the communication using indirect addressing is to apply the knowledge of the geometry at the borders to omit border exchanges that don't contain any fluid nodes.

7.1.2 Further optimization

The indirect addressing scheme used in the proxy application developed as part of this thesis used the *fluid index array* method. This method results in a substantial amount of indirect memory accesses, hurting the performance of the streaming kernel somewhat. It would be valuable to see how a different method of indirect addressing, such as the *connectivity matrix*, compares to the one outlined in this thesis.

Intertwining streaming and collision in the LBM[11] is another method that can greatly affect the memory performance of the LBM as a whole. Such an optimization will enhance temporal locality, another useful exploration path for various geometries.

7.1.3 Blood flow simulations

We have only used a single patient-specific RCA model for blood-flow simulations, due to the availability of such models. Although this has proven a valuable real-world model to evaluate performance optimizations of the LBM in blood-flow simulations, a single example domain will not capture enough information about such simulations in general. Experimenting with geometry-specific optimizations on more examples of blood-flow simulations such as other cardiovascular aorta and more examples with various degrees of porosity and sparsity will validate the findings in this thesis and possibly explore further optimizations in blood-flow simulations using the LBM.

Bibliography

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference on - AFIPS '67 (Spring)*, page 483. ACM Press. doi: 10.1145/1465482.1465560. URL http://portal.acm.org/citation.cfm?doid=1465482.1465560.

[2] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Using Performance Modeling to Design Large-Scale Systems. 42(11):42–49. ISSN 0018-9162. doi: 10.1109/MC.2009.372. URL http://ieeexplore.ieee.org/document/5331904/.

[3] E. S. Boek and M. Venturoli. Lattice-Boltzmann studies of fluid flow in porous media with realistic rock geometries. 59(7):2305–2314. ISSN 08981221. doi: 10.1016/j.camwa.2009.08.063. URL https://linkinghub.elsevier.com/retrieve/pii/S0898122109006427.

[4] A. Cancelliere, C. Chang, E. Foti, D. H. Rothman, and S. Succi. The permeability of a random medium: Comparison of simulation with theory. 2(12):2085–2088. ISSN 0899-8213. doi: 10.1063/1.857793. URL https://pubs.aip.org/aip/pof/article/2/12/2085-2088/401818.

[5] S. Chen and G. D. Doolen. LATTICE BOLTZMANN METHOD FOR FLUID FLOWS. 30(1):329–364. ISSN 0066-4189, 1545-4479. doi: 10.1146/annurev.fluid.30.1.329. URL https://www.annualreviews.org/doi/10.1146/annurev.fluid.30.1.329.

[6] L. Dalcin, M. Mortensen, and D. E. Keyes. Fast parallel multidimensional FFT using advanced MPI. URL http://arxiv.org/abs/1804.09536.

[7] S. Dosanjh, R. Barrett, D. Doerfler, S. Hammond, K. Hemmert, M. Heroux, P. Lin, K. Pedretti, A. Rodrigues, T. Trucano, and J. Luitjens. Exascale design space exploration and co-design. 30:46–58. ISSN 0167739X. doi: 10.1016/j.future.2013.04.018. URL https://linkinghub.elsevier.com/retrieve/pii/S0167739X13000782.

[8] F. Dullien. Pore Structure. In *Porous Media*, pages 5–115. Elsevier. ISBN 978-0-12-223651-8. doi: 10.1016/B978-0-12-223651-8.50007-9. URL https://linkinghub.elsevier.com/retrieve/pii/B9780122236518500079.

[9] B. Ferréol and D. H. Rothman. Lattice-Boltzmann Simulations of Flow Through Fontainebleau Sandstone. In P. M. Adler, editor, *Multiphase Flow in Porous Media*, pages 3–20. Springer Netherlands. ISBN 978-90-481-4645-1 978-94-017-2372-5. doi: 10.1007/978-94-017-2372-5_1. URL http://link.springer.com/10.1007/978-94-017-2372-5_1.

[10] M. P. I. Forum. MPI: A Message-Passing Interface Standard Version 4.0, 2021-jun. URL https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[11] Y. Fu, F. Li, F. Song, and L. Zhu. Designing a Parallel Memory-Aware Lattice Boltzmann Algorithm on Manycore Systems. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 97–106. IEEE. ISBN 978-1-5386-7769-8. doi: 10.1109/CAHPC.2018.8645909. URL https://ieeexplore.ieee.org/document/8645909/.

[12] Z. Guo, B. Shi, and N. Wang. Lattice BGK Model for Incompressible Navier–Stokes Equation. 165(1):288–306. ISSN 00219991. doi: 10.1006/jcph.2000.6616. URL https://linkinghub.elsevier.com/retrieve/pii/S0021999100966166.

[13] J. L. Gustafson. Reevaluating Amdahl's law. 31(5):532–533. ISSN 0001-0782, 1557-7317. doi: 10.1145/42411.42415. URL https://dl.acm.org/doi/10.1145/42411.42415.

[14] P. Hou, X. Liang, F. Gao, J. Dong, J. He, and Y. Xue. Quantitative visualization and characteristics of gas flow in 3D pore-fracture system of tight rock based on Lattice Boltzmann simulation. 89:103867. ISSN 18755100. doi: 10.1016/j.jngse.2021.103867. URL https://linkinghub.elsevier.com/retrieve/pii/S1875510021000743.

[15] C. Kessler and J. Keller. Models for Parallel Computing: Review and Perspectives. 24:13–29. URL https://www.ida.liu.se/~chrke55/papers/modelsurvey.pdf.

[16] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-Driven, Highly-Scalable Dragonfly Topology. In *2008 International Symposium on Computer Architecture*, pages 77–88. IEEE. ISBN 978-0-7695-3174-8. doi: 10.1109/ISCA.2008.19. URL http://ieeexplore.ieee.org/document/4556717/.

[17] F. B. Kjolstad and M. Snir. Ghost Cell Pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns - ParaPLoP '10*, pages 1–9. ACM Press. ISBN 978-1-4503-0127-5. doi: 10.1145/1953611.1953615. URL http://portal.acm.org/citation.cfm?doid=1953611.1953615.

[18] Y.-H. Lee, L.-M. Huang, Y.-S. Zou, S.-C. Huang, and C.-A. Lin. Simulations of turbulent duct flow with lattice Boltzmann method on GPU cluster. 168:14–20. ISSN

00457930. doi: 10.1016/j.compfluid.2018.03.064. URL https://linkinghub.elsevier.com/retrieve/pii/S0045793018301683.

[19] G. Liu, S. Patel, R. Balakrishnan, and T. Lee. IMEXLBM 1.0: A Proxy Application based on the Lattice Boltzmann Method for solving Computational Fluid Dynamic problems on GPUs. URL http://arxiv.org/abs/2201.11330.

[20] K. Mattila, T. Puurtinen, J. Hyväluoma, R. Surmas, M. Myllys, T. Turpeinen, F. Robertsén, J. Westerholm, and J. Timonen. A prospect for computing in porous materials research: Very large fluid flow simulations. 12:62–76. ISSN 18777503. doi: 10.1016/j.jocs.2015.11.013. URL https://linkinghub.elsevier.com/retrieve/pii/S1877750315300478.

[21] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. pages 19–25.

[22] S. Melchionna, M. Bernaschi, S. Succi, E. Kaxiras, F. J. Rybicki, D. Mitsouras, A. U. Coskun, and C. L. Feldman. Hydrokinetic approach to large-scale cardiovascular blood flow. 181(3):462–472. ISSN 00104655. doi: 10.1016/j.cpc.2009.10.017. URL https://linkinghub.elsevier.com/retrieve/pii/S0010465509003385.

[23] P. Min. Binvox. URL http://www.patrickmin.com/binvox.

[24] C. Nita, L. M. Itu, C. Suciu, and C. Suciu. GPU accelerated blood flow computation using the Lattice Boltzmann Method. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE. ISBN 978-1-4799-1365-7 978-1-4799-1364-0. doi: 10.1109/HPEC.2013.6670324. URL http://ieeexplore.ieee.org/document/6670324/.

[25] F. Nooruddin and G. Turk. Simplification and repair of polygonal models using volumetric techniques. 9(2):191–205. ISSN 1077-2626. doi: 10.1109/TVCG.2003.1196006. URL http://ieeexplore.ieee.org/document/1196006/.

[26] OpenMP Architecture Review Board. OpenMP Application Program Interface. URL https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

[27] T. Ramstad, P.-E. Øren, and S. Bakke. Simulation of Two Phase Flow in Reservoir Rocks Using a Lattice Boltzmann Method. In *All Days*, pages SPE–124617–MS. SPE. doi: 10.2118/124617-MS. URL https://onepetro.org/SPEATCE/proceedings/09ATCE/All-09ATCE/SPE-124617-MS/147204.

[28] R. Rew, G. Davis, S. Emmerson, C. Cormack, J. Caron, R. Pincus, E. Hartnett, D. Heimbigner, L. Appel, and W. Fisher. Unidata NetCDF. URL http://www.unidata.ucar.edu/software/netcdf/.

[29] M. Schulz, M. Krafczyk, J. Tölke, and E. Rank. Parallelization Strategies and Efficiency of CFD Computations in Complex Geometries Using Lattice Boltzmann Methods on High-Performance Computers. In M. Breuer, F. Durst, and C. Zenger, editors,

High Performance Scientific And Engineering Computing, volume 21, pages 115–122. Springer Berlin Heidelberg. ISBN 978-3-540-42946-3 978-3-642-55919-8. doi: 10.1007/978-3-642-55919-8_13. URL http://link.springer.com/10.1007/978-3-642-55919-8_13.

[30] M. Själander, M. Jahre, G. Tufte, and N. Reissmann. EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure.

[31] Stepniak et al. Phantom Coronary Artery Models. URL https://3d.nih.gov/entries/3DPX-012589.

[32] K. Suga, Y. Kuwata, K. Takashima, and R. Chikasue. A D3Q27 multiple-relaxation-time lattice Boltzmann method for turbulent flows. 69(6):518–529. ISSN 08981221. doi: 10.1016/j.camwa.2015.01.010. URL https://linkinghub.elsevier.com/retrieve/pii/S0898122115000346.

[33] C. Sun and L. L. Munn. Lattice-Boltzmann simulation of blood flow in digitized vessel networks. 55(7):1594–1600. ISSN 08981221. doi: 10.1016/j.camwa.2007.08.019. URL https://linkinghub.elsevier.com/retrieve/pii/S0898122107006402.

[34] C. A. Taylor and D. A. Steinman. Image-Based Modeling of Blood Flow and Vessel Wall Dynamics: Applications, Methods and Future Directions: Sixth International Bio-Fluid Mechanics Symposium and Workshop, March 28–30, 2008 Pasadena, California. 38(3):1188–1203. ISSN 0090-6964, 1573-9686. doi: 10.1007/s10439-010-9901-0. URL http://link.springer.com/10.1007/s10439-010-9901-0.

[35] T. Tomczak. Data-Oriented Language Implementation of Lattice-Boltzmann Method for Dense and Sparse Geometries. URL http://arxiv.org/abs/2108.13241.

[36] T. Zeiser, G. Hager, and G. Wellein. BENCHMARK ANALYSIS AND APPLICATION RESULTS FOR LATTICE BOLTZMANN SIMULATIONS ON NEC SX VECTOR AND INTEL NEHALEM SYSTEMS. 19(04):491–511. ISSN 0129-6264, 1793-642X. doi: 10.1142/S0129626409000389. URL https://www.worldscientific.com/doi/abs/10.1142/S0129626409000389.

[37] Q. Zou and X. He. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. 9(6):1591–1598. ISSN 1070-6631, 1089-7666. doi: 10.1063/1.869307. URL https://pubs.aip.org/aip/pof/article/9/6/1591-1598/260464.

Appendix A

Selected code

A.1 Collision kernel

```
/* Collision/relaxation operator:
 * redistribute the densities at each lattice point so that they
 * approach equilibrium.
 */
void
collide(void)
{

#pragma omp parallel for
#ifdef OCOLLIDE_LOOP_INDIRECT
  for (int_t n = 0; n < fluid_node_count; n++) {
    int_t k = p_coords[n].z;
    int_t i = p_coords[n].y;
    int_t j = p_coords[n].x;
    if (i < 0 || k < 0 || i > height - 1 || k > depth - 1)
      continue;
#else
  for (int_t k = 0; k < depth; k++)
    for (int_t i = 0; i < height; i++)
      for (int_t j = 0; j < width; j++) {
#endif
    int_t global_j = j /* + cart_coords[1] * width */;
    int_t global_i = cart_coords[0] * height;
    real_t rho = 0.0, uc = 0.0;
    V(i, j, k, 0) = 0.0;
    V(i, j, k, 1) = 0.0;
    V(i, j, k, 2) = 0.0;

    /* Disregard solid points */
    if (MAP(i, j, k) == FLUID || MAP(i, j, k) == WALL) {

      /* Calculate the current velocity
       * field in the fluid */
      if (MAP(i, j, k) == FLUID) {
```

```
35          for (int_t d = 0; d < D_COUNT; d++) {
36            rho += D_now(i, j, k, d);
37            V(i, j, k, 0) += c[d][0] * D_now(i, j, k, d);
38            V(i, j, k, 1) += c[d][1] * D_now(i, j, k, d);
39            V(i, j, k, 2) += c[d][2] * D_now(i, j, k, d);
40          }
41          V(i, j, k, 0) /= rho;
42          V(i, j, k, 1) /= rho;
43          V(i, j, k, 2) /= rho;
44        }
45
46        /* Calculate collision operator,
47           inspired by "A Coupled Approach
48           for Fluid Dynamic Problems
49            Using the PDE Framework
50           Peano", Neumann et al.
51         */
52        for (int_t d = 0; d < D_COUNT; d++) {
53          real_t N_eq, delta_N;
54          uc = c[d][0] * V(i, j, k, 0) + c[d][1] * V(i, j, k, 1) +
55               c[d][2] * V(i, j, k, 2);
56          N_eq =
57            w[d] * rho *
58            (1.0 + (uc / (c_s * c_s)) +
59             ((uc * uc) / (2.0 * c_s * c_s * c_s * c_s)) -
60             ((V(i, j, k, 1) * V(i, j, k, 1) + V(i, j, k, 0) * V(i, j, k, 0)
      +
61               V(i, j, k, 2) * V(i, j, k, 2)) /
62              (2 * c_s * c_s)));
63
64          delta_N = lambda * (D_now(i, j, k, d) - N_eq);
65
66          /* External force at j=1 */
67          if (global_j == 1 && MAP(i, j, k) == FLUID)
68            delta_N += w[d] * (c[d][0] * force[0] + c[d][1] * force[1] +
69                               c[d][2] * force[2]); // 1e-3;
70
71          switch (MAP(i, j, k)) {
72            /* Redistribute fluid according to density/velocity */
73            case FLUID:
74              D_nxt(i, j, k, d) = D_now(i, j, k, d) + delta_N;
75              break;
76            /* Walls reflect incoming mass in opposite direction */
77            case WALL:
78              if (d != 0)
79                D_nxt(i, j, k, bounce(d)) = D_now(i, j, k, d);
80              break;
81            /* No work to do on solid points */
82            case SOLID:
83              break;
84          }
85        }
86      }
87    }
88  }
```

code/lbm_d3q27.c

A.2 MPI type creation

```
1  void
2  create_types(int ndims,
3               int* local_size,
4               int_t* local_ghost_size,
5               int* local_origin,
6               int* global_size)
7  {
8
9    MPI_Type_contiguous(D_COUNT, MPI_DOUBLE, &lattice_pt);
10   MPI_Type_commit(&lattice_pt);
11
12   /*
13    * Depth, column and row types. In
14    * order of major: z -> y -> x
15    */
16   MPI_Type_vector(local_ghost_size[2],
17                   1,
18                   local_ghost_size[0] * local_ghost_size[1],
19                   MPI_INT64_T,
20                   &z_row);
21   MPI_Type_vector(
22     local_ghost_size[0], 1, local_ghost_size[1], MPI_INT64_T, &column);
23   MPI_Type_vector(
24     1, local_ghost_size[1], local_ghost_size[1], MPI_INT64_T, &row);
25   MPI_Type_vector(
26     1, local_ghost_size[1], local_ghost_size[1], MPI_INT, &row_INT);
27
28   MPI_Type_commit(&column);
29   MPI_Type_commit(&row);
30   MPI_Type_commit(&z_row);
31   MPI_Type_commit(&row_INT);
32
33   // x-y plane type
34   MPI_Type_vector(
35     1, local_ghost_size[0], local_ghost_size[0], row, &row_col_plane);
36   MPI_Type_vector(
37     1, local_ghost_size[0], local_ghost_size[0], row_INT, &
       row_col_plane_INT);
38
39   // x-z plane type
40   MPI_Type_vector(
41     local_ghost_size[2], 1, local_ghost_size[0], row, &row_dep_plane);
42   MPI_Type_vector(
43     local_ghost_size[2], 1, local_ghost_size[0], row_INT, &
       row_dep_plane_INT);
44
45   // y-z plane type
46   MPI_Type_vector(
47     local_ghost_size[2], 1, local_ghost_size[1], column, &col_dep_plane);
48
49   MPI_Type_commit(&row_col_plane);
50   MPI_Type_commit(&row_dep_plane);
51   MPI_Type_commit(&row_dep_plane_INT);
52   MPI_Type_commit(&row_col_plane_INT);
53   MPI_Type_commit(&col_dep_plane);
```

```
54
55    // Map domain subarray
56    MPI_Type_create_subarray(DIMS,
57                             global_size,
58                             local_size,
59                             local_origin,
60                             MPI_ORDER_C,
61                             MPI_INT,
62                             &map_domain);
63    MPI_Type_commit(&map_domain);
64 }
```

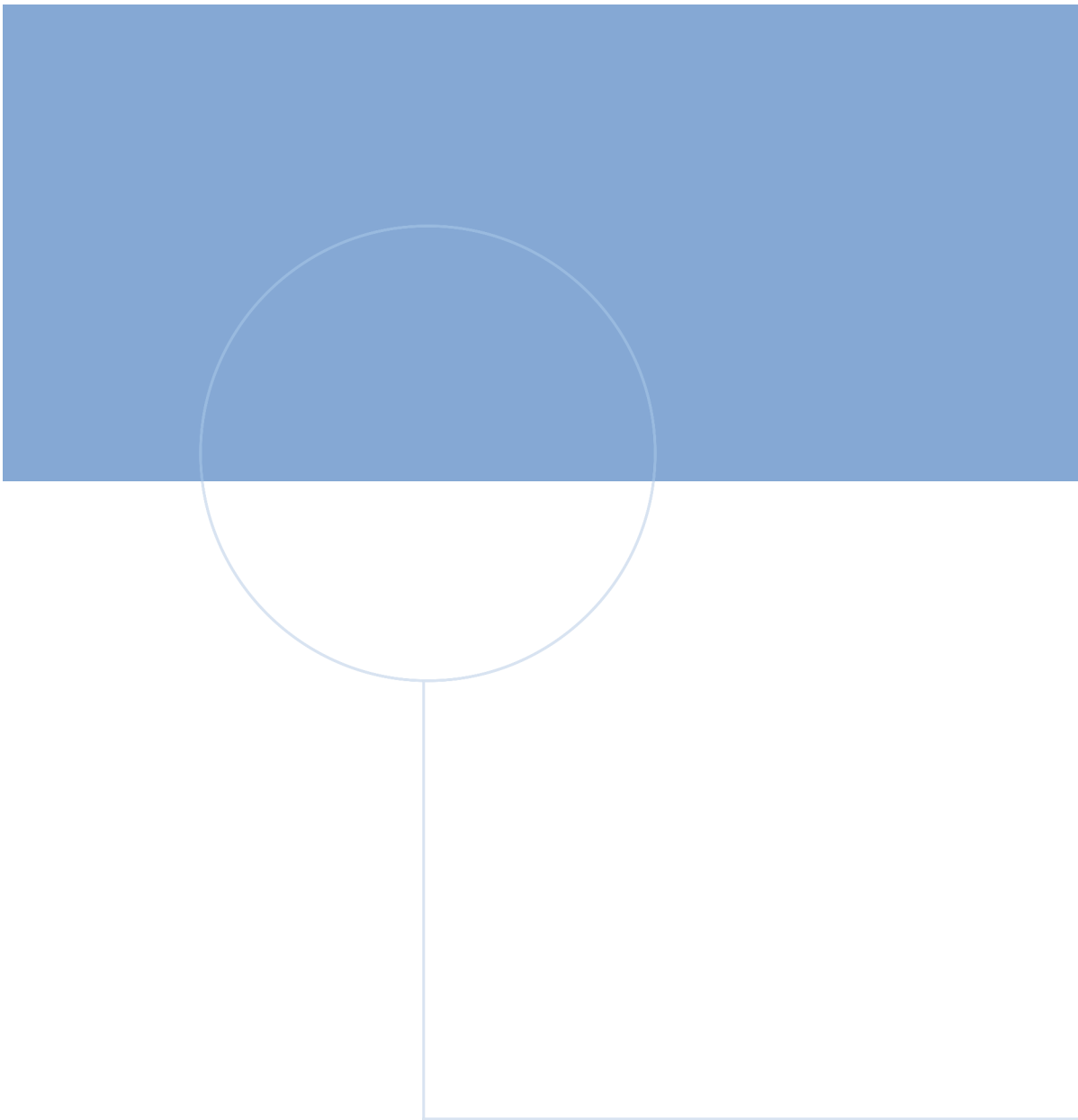
code/lbm_d3q27.c

A.3 Border exchange

```
1  void
2  border_exchange(void)
3  {
4    // South
5    //   ^
6    // North
7    MPI_Sendrecv(density[1],
8                 1,
9                 mpi_t_plane_N,
10                NB_N,
11                0,
12                density[1],
13                1,
14                mpi_t_plane_S_ghost,
15                NB_S,
16                0,
17                comm_cart,
18                MPI_STATUS_IGNORE);
19
20   // South
21   //   v
22   // North
23   MPI_Sendrecv(density[1],
24                1,
25                mpi_t_plane_S,
26                NB_S,
27                0,
28                density[1],
29                1,
30                mpi_t_plane_N_ghost,
31                NB_N,
32                0,
33                comm_cart,
34                MPI_STATUS_IGNORE);
35
36   // East <- West
37   MPI_Sendrecv(density[1],
38                1,
39                mpi_t_plane_W,
40                NB_W,
41                0,
42                density[1],
43                1,
44                mpi_t_plane_E_ghost,
45                NB_E,
46                0,
47                comm_cart,
48                MPI_STATUS_IGNORE);
49
50   // East -> West
51   MPI_Sendrecv(density[1],
52                1,
53                mpi_t_plane_E,
54                NB_E,
55                0,
```

```
56                      density[1],
57                      1,
58                      mpi_t_plane_W_ghost,
59                      NB_W,
60                      0,
61                      comm_cart,
62                      MPI_STATUS_IGNORE);
63  }
```

code/lbm_d3q27.c



NTNU

Norwegian University of
Science and Technology