Sindre Skåland Brun

Performance impact of porous media characteristics for lattice Boltzmann method

Master's thesis in Computer science
Supervisor: Jan Christian Meyer
July 2020

Master's thesis

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

Sindre Skåland Brun

Performance impact of porous media characteristics for lattice Boltzmann method

Master's thesis in Computer science
Supervisor: Jan Christian Meyer
July 2020

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

Acknowledgements
I would like to thank my supervisor Jan Christian Meyer for his support and guidance throughout the project.

Project Description

This study aims to model and evaluate the performance potential of exploiting porous rock cavities in lattice Boltzmann simulations, using a proxy application.

Summary

With compute resources being limited, there is a constant need to extract performance improvements from both new and existing applications. In this thesis we utilize performance modeling techniques and a proxy application to explore potential optimizations in a lattice Boltzmann fluid simulation.

The method and application is an iterative method, and we develop a model that is able to predict the iteration time with an average error of 20%. The model captures computational costs as well as synchronization costs during communication between ranks.

We develop an analytical model which is able to identify performance potential, though we are only able to extract this potential using characteristics extracted from the implemented application.

Further, we identify characteristics of porous media and techniques from pore space modeling which can be used to improve performance in future works.

Table of Contents

List of Tables

List of Figures

List of Source code

Chapter 1

Introduction

The lattice Boltzmann method (LBM) is an important method used in computational fluid dynamics. What sets LBM apart from other computational fluid dynamic methods is its inherent parallelism and support for complex boundaries. It also admits multiple fluids or phases naturally. It is used in the the oil and gas industry to determine properties of porous media such as permeability. It can also be used to determine the potential for CO_2 deposition in porous rocks.

In this thesis, we investigate properties of porous media to improve performance of the lattice Boltzmann method through modeling of a proxy application. Common reservoir rocks have a porosity of 5-40%, meaning the space in which we need to simulate fluid is small. This can be especially true if the computation is split across multiple machines as is common in high performance computing (HPC), and the porosity is not evenly distributed in the geometry. In this thesis, we use modeling to determine the potential of optimizations based on porosity to balance the computation when spread across multiple nodes. We consider the problem to be a partitioning and scheduling problem, and determine models which describe the performance potential of the application.

Our goal is to explore the approach of using modeling to expose and extract performance potential, and validating this approach on a proxy application. The proxy application computes fluid flow in a 2D geometry using the lattice Boltzmann method. It maintains the computational characteristics of a LBM solver while being simpler to develop and experiment on.

We investigate and utilize methods from the field of performance modeling to predict the iteration time. We also introduce methods from characterization of porous rocks, such as pore network modeling, laying the foundations for optimizations both in this and future works.

In Chapter 2 we review relevant background on proxy applications, high performance computing, programming and performance models and simulation. Chapter 3 reviews porous media and the lattice Boltzmann method. Chapter 4 describes the developed and modeled application. Chapter 5 describes the model of the application. In Chapter 6 we propose performance optimizations. Chapter 7 describes the platforms used and experimental setup. In Chapter 8 we discuss the results, and in Chapter 9 we conclude.

Chapter 2

Background

In this chapter we introduce the relevant background for this thesis. Section 2.1 describes proxy applications. In Section 2.2 we review relevant characteristics of system architecture for high performance computation. Section 2.3 introduces programming models for parallel computing. Section 2.4 describes performance modelling of computation and communication. Section 2.5 introduces the use of simulation in performance prediction.

2.1 Proxy Applications

A proxy application is an application that exhibits the same computational patterns as a larger application or class of applications, while being simpler in design. It often has a more limited input parameter space and lacks features expected from a complete application, such as error handling. The goal of a proxy application is to enable rapid prototyping and testing of modifications on a simpler code base.

2.1.1 Proxy characterization

Dosanjh et al. [12] classify different types of proxy applications based on characteristics, refactoring and performance modeling scope and code size. The types and their characteristics are presented in Table 2.1.

Proxy Type	Characteristics
Kernels	Small, self contained code fragments that represent key hot spots in an application
Benchmarks	Typically meant to be static code with precise input and usage restrictions.
Compact App	Simplified, but complete physics simulation.
Skeleton app	Accurate interprocessor communication model with synthetic computation.
Miniapp	Focused on one or a few performance-impacting aspects of the application.

Table 2.1: Proxy types and characteristics. Reproduced from [12]

Some proxy applications are developed to mirror a specific “full” application. In this case it is interesting to characterize how well a proxy application relates to the application it represents. Aaziz et al. [2] and Aaziz et al. [1] explore a methodology for characterizing this relationship.

2.2 High Performance Computing

In this section we review some basics of High Performance Computing (HPC).

2.2.1 Nodes

Modern HPC systems consist of hundreds or thousands of interconnected nodes. Each node contains one or more processors and dedicated memory. Each processor is a superscalar many core processor. Processors utilize speculative execution in terms of branch prediction to improve performance due to pipelining. This affects the performance of loops with irregular branching patterns. The proxy application exhibits this effect when it comes to irregular porosity.

Each node can also be equipped with accelerators which optimize specific workloads or computations. Many of the worlds largest systems today rely heavily on graphical processing units to increase the raw compute performance[37]. We review GPUs in Section 2.3.4.

2.2.2 Network

In this section, we review HPC network technologies. HPC networking distinguishes itself from traditional networking by having smaller stacks focused on low latency and high bandwidth. Section 2.2.2 review important characteristics of the network technology or *fabric*. Section 2.2.2 introduces network topologies.

Technology

While the network is often abstracted away by the programming model, its design and capabilities affect which models are applicable. The interconnect technology used in this thesis is Mellanox Infiniband. Remote Direct Memory Access (RDMA) is an important feature of Infiniband, and impacts communication performance and modeling.

Remote Direct Memory Access Remote Direct Memory Access (RDMA) is a feature of many modern interconnect technologies. RDMA can enable both read and write operations from remote memory. RDMA is supported by many data center interconnects. Examples are Infiniband, iWARP (RDMA over TCP and SCTP) and RoCE (RMA over Converged Ethernet). Correctly using RDMA without introducing memory corruption is handled by a communications library such as MPI, which allocates space that other processes can write into. When the recipient is ready, the data is then copied from this buffer into the actual receive buffer.

Infiniband Infiniband is a packet based interconnect specifically targeted at high performance computing(HPC) [24]. It is connected directly to the memory bus, and supports both read and write remote memory access. It also supports multiple transmission types similar to TCP/UDL, called reliable channel and unreliable datagram.

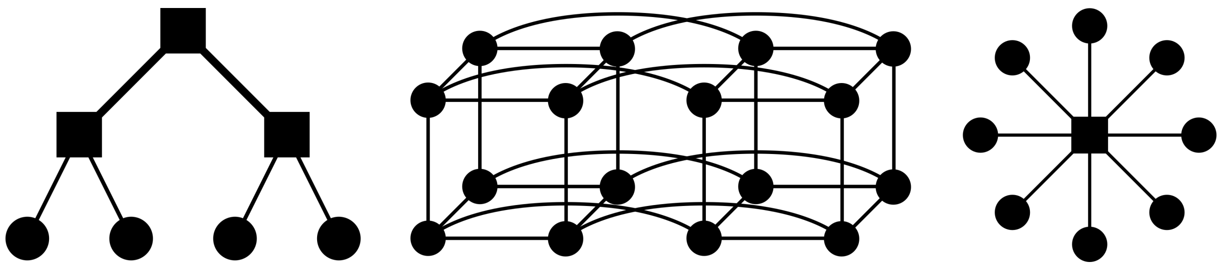


Figure 2.1: Interconnect topologies

The Infiniband standard does not have a definite interface specification. Rather it specifies a series of *verbs* which should be implemented, but the exact implementation is up to the provider. One such implementation is provided by the open framework alliance. Infiniband also supports another higher level API called the Direct Access Programming Library (DAPL), which wraps the low-level verbs API.

Topology

The interconnect topology can have a high impact on communication performance. An interconnect topology can be more suitable for specific communication patterns. The platform used in this work has a fat-tree topology. We also describe the hypercube due to its extended use in HPC. Figure 2.1 shows an illustration of the topologies discussed.

Fat-tree In an ideal fat-tree, nodes are organized in a tree. At each level, the switch has an upstream link equal to the sum of its downstream links. This balances the number of switches while maintaining full bisection bandwidth.

Star In a star network, all nodes are connected to a shared switch. We briefly review this, as the nodes used on Idun are connected to the same switch, and becomes a star.

Hypercube A hypercube is a highly connected topology, focusing on keeping easy addressing, and low hop count. It is used in many of the larges compute clusters in the world as per the supercomputer ranking[37]. One reason for this is that many algorithms naturally have communication patterns that are hypercubes. These algorithms are called *hypercube algorithms* or *d-cube algorithms*[16]. Examples of these algorithms include fast fourier transforms (FFT). These algorithms have the property that at any point a node will only need to communicate with its direct neighbors.

2.3 Programming models

The programming model impacts the achieved performance. We consider selecting a programming model to be a problem of finding a balance between programmer productivity, ease of developing a correct program, and utilizing the hardware.

Here we introduce four programming models. The Bulk Synchronous Parallel (BSP) model serves as an example of a model which shares characteristics with the LBM application, but has not been utilized directly. Shared Memory and Message Passing are both models utilized in the

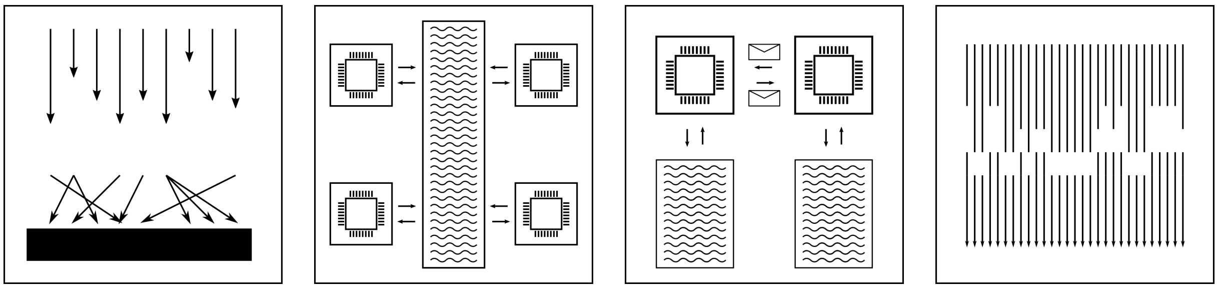

(a) Bulk Synchronous Parallel (b) Shared Memory (c) Message Passing (d) GPGPU

Figure 2.2: An illustration of the programming models described.

proxy application. General Programming Graphical Processing Unit (GPGPU) is included for reference as the LBM method is often computed using GPUs, though this is not used in this project. An illustration of the different models is shown in Figure 2.2

2.3.1 Shared Memory

In the Shared Memory model all compute resources share access to the same memory space. Modern multi-core or multi-processor machines are examples of systems which are suited for such a model. All processes on the machine share the same memory, and synchronization is realized through different mechanisms, such as support or atomic operations.

OpenMP

OpenMP is a compiler extension supported by most modern compilers for generating shared-memory parallel code[29]. OpenMP directives are included in source code, and the compiler generates parallel code. For example, `#pragma omp parallel for` can be used to parallelize a for-loop construct.

OpenMP originally targeted work-sharing constructs, where multiple threads cooperate to compute a known amount of work. Both static sharing, as well as dynamic sharing is supported. Starting with version 3, task-sharing constructs are also supported. In task-sharing the work is produced within the parallel region.

2.3.2 Message Passing

Message Passing is a model in which cooperation is done through the explicit sending and reception of messages. It is designed for multiple nodes with local memory. We treat this model and the Message Passing Interface (MPI) in details as it is important to the modeling of the communication in the proxy application.

MPI

The Message Passing Interface[15] is an application programming interface standard, providing a standardized interface for distributed memory programming using message passing. The MPI standard is governed by the MPI committee, and specific implementations provided by different vendors.

Communication types In MPI, processes known as *ranks* cooperate through passing messages between each other. The MPI specification primarily defines two types of operations, point to point operations and collective operations. Collective operations work on groups of participating ranks.

Communication protocols Most MPI implementations implement two main transmission protocols, eager and rendezvous.

In the eager protocol the message is sent directly to the recipient. This requires the recipient to be able to buffer the message until the receiving process is ready to receive the message, and copy it into the final memory destination. If the receive is posted before the message is received, the buffering and extra copy can be avoided.

In the rendezvous protocol a `Ready to Receive` request is sent with contains the size of the message about to be sent. The sender then await the `Cleared to Send` (`CTS`) before sending the actual message. This adds an extra round-trip of communication which needs to be taken into account when modeling the communication. Depending on the MPI implementation, the rendezvous protocol can be progressed independently, while others require a call to MPI which will allow the MPI engine to progress the message. In many implementations, this is also user configurable.

If the interconnect supports Remote Direct Memory Access (RDMA), the MPI implementation can utilize this to further speed up transmission. This can allow a transfer to be executed with minimal overhead. The points at which MPI utilizes RDMA capabilities and/or eager/rendezvous protocols are implementation specific.

Communication modes Using the protocols described above, MPI supports different sending modes. The modes are distinguished by which guarantee MPI gives about the state of the transmission. Each mode exists in a blocking and non-blocking version. In the non-blocking version the call returns immediately, and the request is `waited` or `tested` later. The non-blocking calls are prefixed with an `I` for `Immediate`.

- Standard send - Returns when the send buffer may be reused. It may buffer the message in a system buffer, or block until message is sent.
- Buffered send - Depending on the message size will buffer or send directly and return. The specification describes the buffered as being *local* while a standard mode send is *non-local*[26].
- Synchronous send - Returns when receiver has started reception (but not necessarily completed reception)
- Ready send - Can only be used then receive has been posted, otherwise its behavior is undefined.

There is only one receive mode, though it exists in both blocking and non-blocking versions. Note that non-blocking does not necessarily mean parallel. It was originally a method to avoid deadlocks.

Depending on the hardware, MPI can also support RDMA for accelerator devices, such as graphical processing units.

2.3.3 Bulk Synchronous Parallel

BSP is a programming model which splits computation into computational steps, called super steps [39]. Each super step consists of computation, followed by a communication and finally synchronization. For each super step, data written during the computation is only visible to other processes after the synchronization step. This means that all threads/processors work independently in the step, and the changed state is only visible to other processors in the subsequent step.

BSPlib and other uses

The BSP model has been implemented in many different libraries such as Oxford BSP Toolset[18] and BSPlib[40]. It can be implemented on top of both shared memory and message passing platforms. BSP is also used as a computational model in graph processing tools such as Apache Hama [34] and Google Pregel [25].

2.3.4 GPGPU

General purpose computing on graphics processing units (GPUs) has come into widespread use in high performance computing. GPUs are designed to have a large number of simple cores working in lockstep through having shared control logic. The simpler cores and shared logic enables a large number of cores per GPU. Modern data center GPUs such as the NVIDIA V100 has 5120 CUDA cores. GPGPUs can provide large speedups on parallel computations. The keys are large core numbers and high memory bandwidth. Highly parallel methods such as LBM are well suited for GPU acceleration.

CUDA

CUDA is NVIDIAs GPGPU interface which can be utilized by programmers. It allows a subset of C++ to be compiled and run on the GPU. The CUDA cores operate on the GPU memory under the shared memory model.

2.4 Performance modeling

In this section we review both applied and considered modeling techniques.

Barker et al. [4] introduce the fundamental theorem of performance modeling. The fundamental theorem expresses the total time of a program as a sum of computation and communication, minus the overlap. It can serve as the starting point for modeling the performance of an application. This is expressed in Equation 2.1.

$$T_{tot} = T_{comp} + T_{comm} - T_{overlap} \tag{2.1}$$

In this section, we review models that can be used to further expand the terms of this equation to produce accurate performance estimations. Section 2.4.1 introduces performance modeling of the computation and Section 2.4.2 introduces models for modeling communication time.

2.4.1 Computation models

Because of the diversity in programs, it may be difficult to express a program in terms of the operations it does. Such a model is practically the source code of the application. Furthermore, the time an operation takes is largely influenced by other factors such as memory access and pipelining, speculative execution etc.

Instead we often rely on benchmarking of the application itself and expressing the computation as a set of parameters known to influence computational load. This however requires that it is possible to identify such parameters, and that the computation follows a regular pattern.

2.4.2 Communication models

In this section we introduce relevant communication models. The Hockney model in Section 2.4.2 serves as a baseline model. The LogP and descendants described in Section 2.4.2 and 2.4.2 have been highly influential, and is also used to model the application. Rico-Gallego et al. [31] provides a review of communication performance models.

Hockney

The Hockney model is considered the fundamental model, and is also referred to as the postal model. It was first introduced by Hockney [19]. In this model, a transmission consists of a latency α and inverse bandwidth β. The model is stated in Equation 2.2 where m is the message size. α and β are constants, and t is the time of a transfer.

$$t = \alpha + \beta \cdot m \tag{2.2}$$

Generally speaking, most modern models can be considered specializations of this model, where the overhead is decomposed into more finer grained components, such as separately modeling sender and receiver overhead, expressing the bandwidth as a function of message size, and so forth. The Hockney model is still in use, and is used by the MPI implementation MPICH to select collective operations algorithm during runtime[31].

LogP

We review the LogP model in greater detail, as it is the foundation of many recent models, and has spawned a great number of other models. The LogP model was introduced by Culler et al. [11] and describes communication in terms of the following:

- L - Latency
- o - Overhead
- g - gap per byte
- P - processors involved

LogGP

LogGP[3] was is an important development of the LogP model. The LogGP extends the LogP model with a new parameter G which is the bandwidth for long messages. The LogGP model is the foundation of most of the later descendants of the LogP model.

LogGPS

The LogP model does not account for synchronization costs caused by different communication protocols employed by middleware such as MPI. LogGPS[22] is an extension of the LogGP model which tries to capture this synchronization cost.

The LogGPS model splits the overhead into a sender and receiver overhead. It also introduces software parameters s and S which are the message sizes for which the MPI implementation switches from short to eager and eager to rendezvous, respectively.

The required parameters are

- L - Latency
- o' - Unit overhead
- O_s - Sending overhead per byte
- O_r - Receive overhead per byte
- g - gap per byte
- G - gap per byte for large messages
- P - processors involved
- s, S - protocol thresholds for short, eager and rendezvous

The *short* protocol in LogGPS refers to transfer of a single package, *eager* and *rendezvous* are as described in Section 2.3.2.

LogGPO

LogGPO is an extension to LogGP which, similarly to LogGPS, tries to account for overlap in communication[10]. The main contribution of LogGPO is to account for the message progression methods. In the cases where we do not have independent message progression and transmit using the rendezvous protocol, the transmission will not be able to progress with overlapping computation. Instead it will stall until the next call to MPI which continues handling of control messages. By explicitly defining overheads for control messages, and including the progression mechanism and transmission pattern, LogGPO promises to provide better estimations for non blocking communication which overlaps computation.

2.5 Simulation

While the models can be used analytically, simulation can enable application of the previously mentioned models to more complex interactions.

Simulators often focus on a specific part of a computation. A key trade-off in simulation is speed versus accuracy. To balance this trade-off, some simulators, such as BigSim[43] and xSim[13] use an emulation guided simulation approach, where the program is emulated and the result of this emulation is used in further larger scale simulation.

Other simulators are more focused on providing accurate communication models, or are tied more directly to a single model. Examples of these are MPI-SIM[10] for LogGPO and LogGOPSim[21] which implements a modified LogGPS model.

Chapter 3

Lattice Boltzmann Method

In Section 3.1 we introduce porous media, in particular porous rocks. Section 3.2 introduces the lattice Boltzmann method used for simulating fluid dynamics.

3.1 Porous media

Porous media is any material containing pores. For our study, we are concerned with porous rock.

The main characteristic of porous media is the porosity. Porosity is the percentage of pore volume to the volume of the rock. This is expressed in Equation 3.1, and often presented as a percentage[17].

$$\phi = \frac{V_{pore}}{V_{total}} \tag{3.1}$$

In this thesis we mainly consider applications of the lattice Boltzmann method to reservoir rocks. Reservoir rocks are usually sandstone or limestone[36] which typically have a porosity of 5-30% and 0 to 40% [14] respectively.

Pores can take different shape, often depending on the type of rock. Figure 3.2 shows examples of pore/grain structure. In reservoir rocks we mainly find larger grains with smaller sediments filling the gaps. Fractures are more commonly found in harder rocks. Collectively the pores are described as the *pore space*, which is all of the non-solid regions of the rock.

3.1.1 Imaging

Bultreys et al. [7] provides an introduction to the imaging techniques used to acquire pore scale images. Most notable is scanning electron microscopy (SEM). After an image is captured, it needs to be segmented. This is often done through thresholding. A sample which still contains gray scale can be seen as a) in Figure 3.1 and a sample of an already segmented image is shown as b).

These images are what we use for LBM for porous media. We use one (2D) or more (3D) images of the rock, and use segmentation techniques to distinguish solid and pore space. The resulting image is used as a direct model, and methods such as LBM compute directly on it.

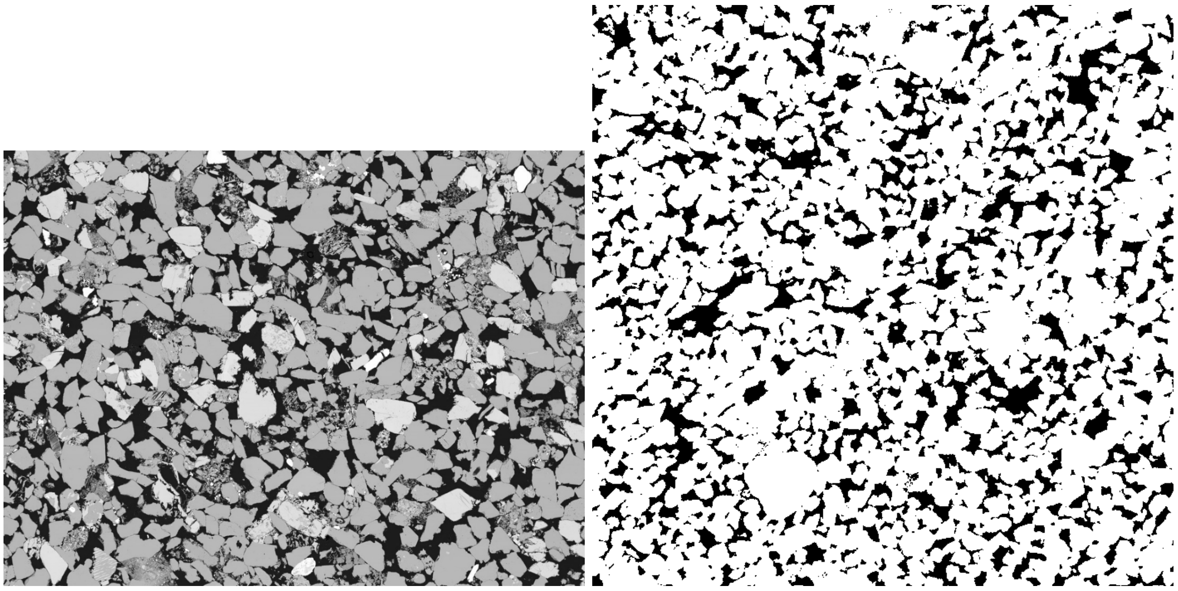
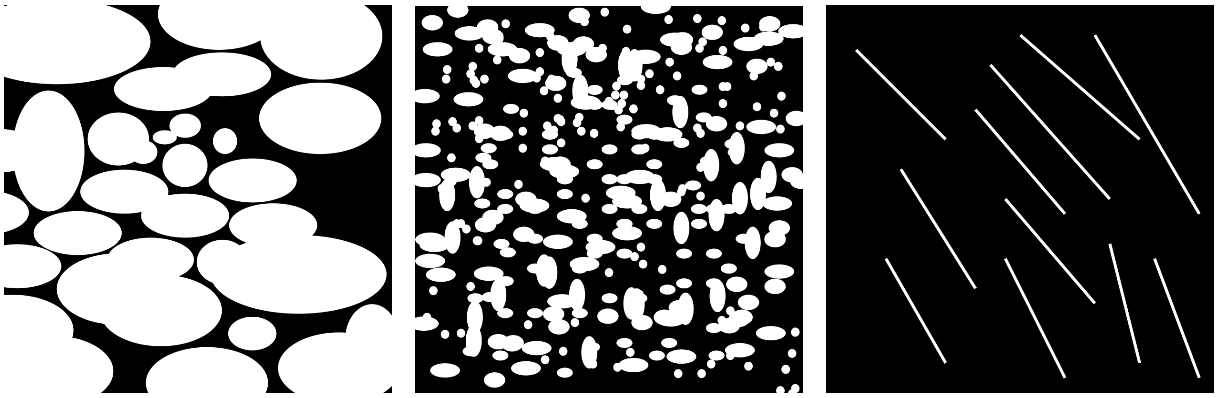


Figure 3.1: Two samples of unsegmented and segmented samples from SEM imaging. Samples are two different rocks, [33, 27]. Black indicate pore space in both samples.



(a) Large grains **(b)** Small grains **(c)** Fractures

Figure 3.2: Illustration of different pore types/grain sizes

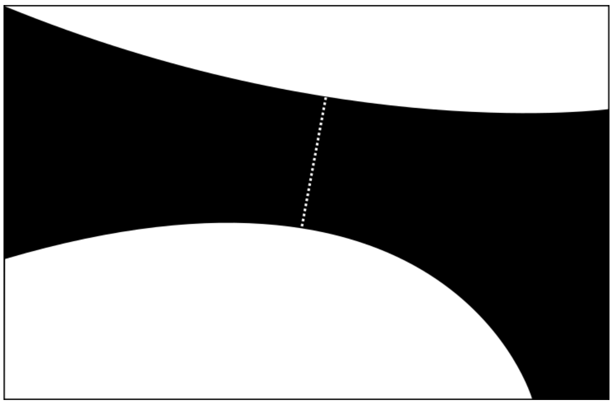


Figure 3.3: Illustration of pore body and throat. White on top and bottom are grains. Black to left and right are pore bodies, and the dashed white line marks the throat.

3.1.2 Pore network models

In network models, the pore space is considered as a network of pores connected by throats. Throats are the border between bodies, as illustrated in Figure 3.3. Originally such models were often lattice based network models stochastically generated from experimental data, but currently image-based network models are gaining in popularity[7].

The LBM which is the focus of our study depends on image models, but we review some network based models extracted from such images. Our goal is to identify techniques that extract features which can be used to characterize the pore network. Such a model can lay a foundation for partitioning techniques or other optimizations. We review some techniques used here.

Medial axis

Medial axis is an image transformation that is used to create a skeleton of the structure in the image. The branching points are then identified as pore body, and the line between the branching points are the throats. However, we often couple the medial axis method with another technique to fully determine the location of the throats. This technique is ineffective for noisy inputs, and can often identify irregular throats as multiple pores. The methods strength however is the direct relationship to the input image.

Watershed

Using a segmented image, a distance map is generated, and a watershed transformation is applied[32]. The starting markers for the watershed can be extracted from a medial axis analysis. The throats are identified as the surface between the pores.

Maximal balls

The maximal balls algorithm was originally introduced by Silin and Patzek [35]. For each point, a maximal ball for touching the pore walls is inscribed. Then, all balls fully contained within another is removed. The balls are then ordered by size, and starting form the largest any overlapping ball is described as a parent/child relationship based on the largest being the parent. When two such *parent trees* meet, it is a *common child*.

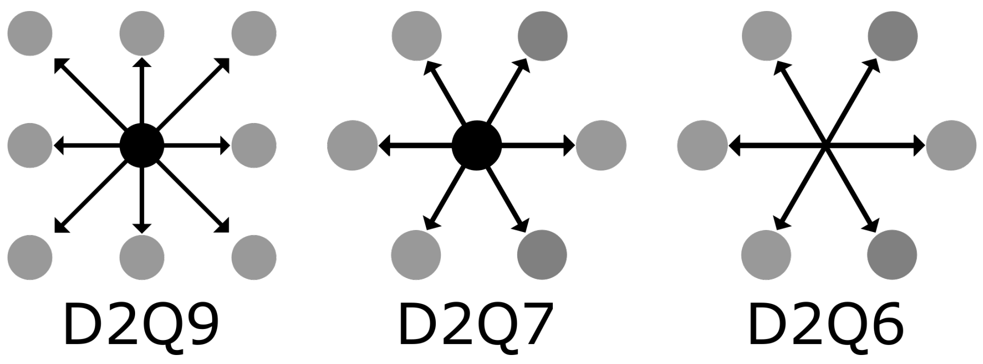


Figure 3.4: Lattice naming scheme

3.2 Lattice Boltzmann Method

The foundation of the lattice Boltzmann Method (LBM) is the lattice gas automata. In this model, instead of tracking particles in continuos space, they are fixed in a lattice. LBM evolves the gas automata by describing the particles as the density in the points, instead of enumerating all particles. Thus, we observe flow as a time series of moving densities in the lattice rather than particle flow. At each time step we compute how the densities evolve.

The LBM is flexible and can support different lattice structures, both different dimensions and number of neighboring points. To denote the lattice structure we use a $DmQn$ naming scheme where m refers to dimensions, and n refers to the number of components/directions the densities is described as. Normally this is the number of neighboring points in the lattice, plus a zero vector. For a Cartesian grid, each node has eight neighbors: north, south, east, west, the four diagonals, and a null vector or rest-particle. This results in a D2Q9 lattice. A cube would be a D3Q15 lattice, or a D3Q27 if all diagonal edges are included. In this work, a D2Q6 lattice has been used. Here, the zero (rest) is removed, and each point has six neighbors. See Figure 3.4.

3.2.1 Mathematical description

The lattice Boltzmann method computes the flow as a time series evolution. The computation naturally separates into a collision and propagation or streaming phase. This process is illustrated in Figure 3.5. The equations in this section are based on work by Chen and Doolen [9] on a D2Q7 lattice, with the rest particle removed to make a D2Q6 lattice, as described in Ragunathan and Valstad [30].

$$f_i(x + e_i, t + \delta t) = f_i(x, t) + \Omega_i(f(x, t)) \tag{3.2}$$

Equation 3.2 is the fundamental equation, with $\Omega()$ being the collision operator and i the direction. $f_i(\bar{x}, t)$ is the fluid density at point $\vec{x}$ at time t. Equation 3.2 states that the density moving in direction e_i at time $t + \delta t$ is the result of the current density, plus the collision. The collision operator is expanded in Equation 3.3.

$$\Omega_i(f(x, t)) = \frac{f_i^{eq}(x, t) - f_i(x, t)}{\tau} \tag{3.3}$$

The $f_i^{eq}(x, t)$ term in Equation 3.3 is an approximation of the equilibrium. This is the Bhatnagar Gross and Krook single order time relaxation towards an equilibrium[5].

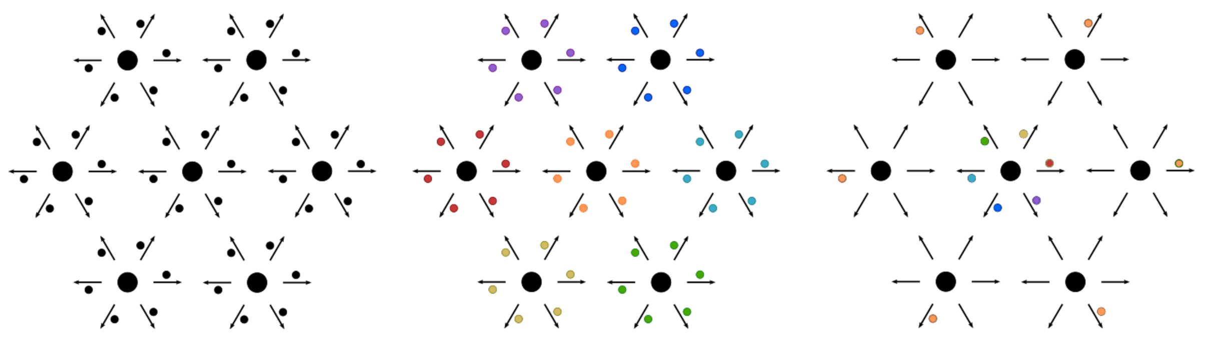


Figure 3.5: The steps of the lattice Boltzmann Method. *Left:* Initial state. *Middle:* Local to each point the collision is computed, which is illustrated by colors in each point. *Right:* Finally, the forces are propagated out to neighboring points. Only forces to and from the center has been included.

The density and momentum in each point is given in Equation 3.4 and 3.5. The implementation of these equations in code is shown in Listing 1.

$$\rho = \sum_{i=0}^{6} f_i \tag{3.4}$$

$$\rho v = \sum_{i=0}^{6} f_i e_i \tag{3.5}$$

Using this the equilibrium function is expanded in Equation 3.6. In Equation 3.7 and 3.8 the terms are written out as they will be implemented in code. The implementation is shown in Listing 2. The vectors for each direction e_i is described by Equation 3.9.

$$\sum_{i=0}^{6} f_i^{eq} = \frac{\rho}{6} \sum_{i=0}^{6} \left\{ 1 + 2 \underbrace{e_i v}_{(*)} + \underbrace{4(e_i v)^2 - 2v^2}_{(**)} \right\} \tag{3.6}$$

$$(*) = e_x v_x + e_y v_y \tag{3.7}$$

$$(**) = 4((e_x^2 - \frac{1}{2})v_x^2 + 2e_x e_y v_x v_y + (e_2^2 - \frac{1}{2})v_y^2) \tag{3.8}$$

$$e_i = \begin{cases} (0,0) & i = 0 \\ (\cos\theta_i c, \sin\theta_i c), \theta_i = (i-1)\frac{\pi}{3} & i = 1, 2, ..., 6 \end{cases} \tag{3.9}$$

3.2.2 Boundaries

Boundaries between solid and fluid can be implemented in different ways. In our application we use a bounce-back boundary for the solid/fluid interface. This means that any force applied to a solid point will be returned back. Listing 3 shows the implementation of this in code. For the edges, we use periodic boundaries both vertically and horizontally. This is to ensure conservation of mass.

```
for ( int i=0; i<6; i++ )
{
    rho += lattice->at(row, col).density[i][NOW];
    lattice->at(row, col).velocity[NOW] +=
        c[i][0] * lattice->at(row, col).density[i][NOW];
    lattice->at(row, col).velocity[NEXT] +=
        c[i][1] * lattice->at(row, col).density[i][NOW];
}
// rho*u = sum_i( Ni*ci ), so divide by rho to find u:
lattice->at(row, col).velocity[NOW] /= rho;
lattice->at(row, col).velocity[NEXT] /= rho;
```

Listing 1: Implementation of Equation 3.4 and 3.5

```
float
    qi_uaub,
    N_eq,
    delta_N;

qi_uaub =
    ( c[i][1] * c[i][1] - 0.5 ) *
        lattice->at(row, col).velocity[NEXT] *
        lattice->at(row, col).velocity[NEXT] +
    ( c[i][1] * c[i][0]       ) *
        lattice->at(row, col).velocity[NEXT] *
        lattice->at(row, col).velocity[NOW] +
    ( c[i][0] * c[i][1]       ) *
        lattice->at(row, col).velocity[NOW] *
        lattice->at(row, col).velocity[NEXT] +
    ( c[i][0] * c[i][0] - 0.5 ) *
        lattice->at(row, col).velocity[NOW] *
        lattice->at(row, col).velocity[NOW];

uc = lattice->at(row, col).velocity[NOW] *
    c[i][0] + lattice->at(row, col).velocity[NEXT] * c[i][1];


// Equilibrium, difference
N_eq = ( rho / 6.0 ) * ( 1.0 + 2.0 * uc + 4.0 * qi_uaub );
delta_N = LAMBDA * ( lattice->at(row, col).density[i][NOW] - N_eq );
```

Listing 2: Implementation of Equation 3.6, 3.7 and 3.8

```
if ( task.lattice->at(row, col).type == Point_type::Fluid )
{
    lattice->at(row, col).density[i][NEXT] =
        lattice->at(row, col).density[i][NOW] + delta_N;
}
else
{
    lattice->at(row, col).density[(i+3)%6][NEXT] =
        lattice->at(row, col).density[i][NOW];
}
```

Listing 3: Implementation of bounce back borders.

Chapter 4

Proxy application

This section describes the implementation details of the proxy application developed and used in this thesis.

The application is written in the MPI+X programming model [23] using C++11. The X in this case is OpenMP. While not being written as a BSP application, the program has similarities to the BSP model. Specifically it has separate compute and synchronization stages. The application is a *mini app* based on the classification described by Dosanjh et al. [12]. This is because it is *Focused on one or few performance-impacting aspect of the application*. In total, as counted by *SLOCCount*[41] the application is approximately 7300 lines of code.

Figure 4.1 shows the overall steps of the computation. The steps of the computation as well as other aspects of the application are described in the following sections.

4.1 Input

The application loads the geometry in the form of a PBM file[28]. PBM is an image format for monochrome images originally developed for email. The PBM format exists in both an ASCII and a binary version. In the ASCII format a '0' character is white and '1' is black. Representing each pixel using a 8-byte character is wasteful, and a binary version with the same extension was developed later. In this work, we have used the original ASCII based format as this allows for visual inspection of the geometry input. Using the program package `netpbm`, one can easily convert between PBM and other image formats.

Listing 4 shows an example of a PBM ASCII or *plain* PBM image. The first line contains a magic number, in this case P1 indicates that this is a plain PBM. The second line states the

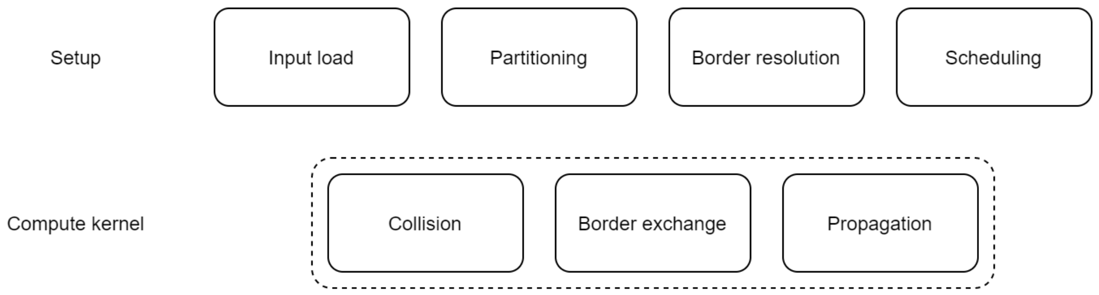


Figure 4.1: The main steps in the proxy application

```
P1
5 5
00000
01110
01110
01110
00000
```

Listing 4: Example of an image in the plain PBM format

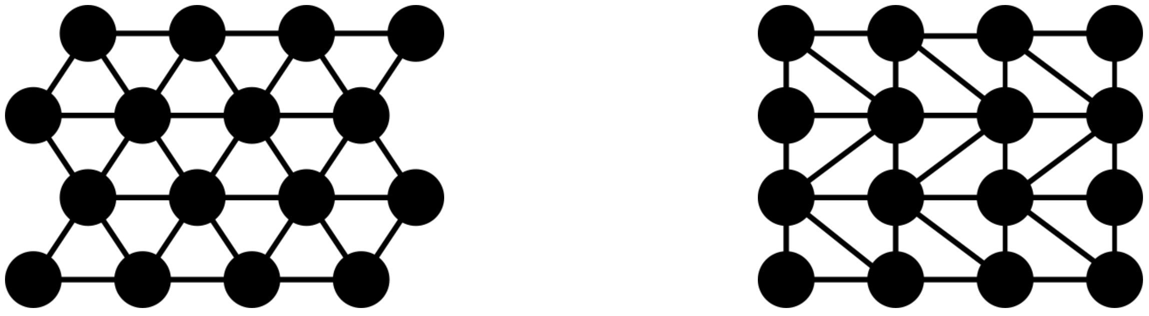

(a) Lattice structure **(b)** Corresponding grid layout

Figure 4.2: A lattice, rows shifted to fit regular memory layout.

dimensions of the image. After the dimensions, the image data follows. All whitespace is ignored.

The application considers black ('1') as solid and white ('0') as void. Note that this is inverted from the traditional images of pore space, where white are solid and black represent void.

4.2 Lattice representation

An *even row shift* converts the image into a lattice. Even rows are shifted to the right by 30°, or equivalently, half the pixel width. With the even row shift, we determine the lattice structure. However, the lattice is still represented as a grid in memory. Figure 4.2 show how the neighbor edges look when the lattice is stored in memory. The indexing of neighboring nodes depends on the y coordinate of the point.

Each point in the lattice contains the density in the point, and the velocity. Two sets of densities are stored, the *current* and *next*. The collision step reads the *current* densities and computes the *next* densities. During the propagation step, the *next* densities are moved from the next to the current state of the neighboring point. This enables perfect parallelism for each individual lattice point for the collision and propagate step.

4.3 Work representation

In a traditional MPI implementation of LBM, each participating process computes a single section of the lattice. The participating ranks are organized in a cartesian grid, and the sections are distributed according to the processor grid. In these cases, all nodes send and receive the

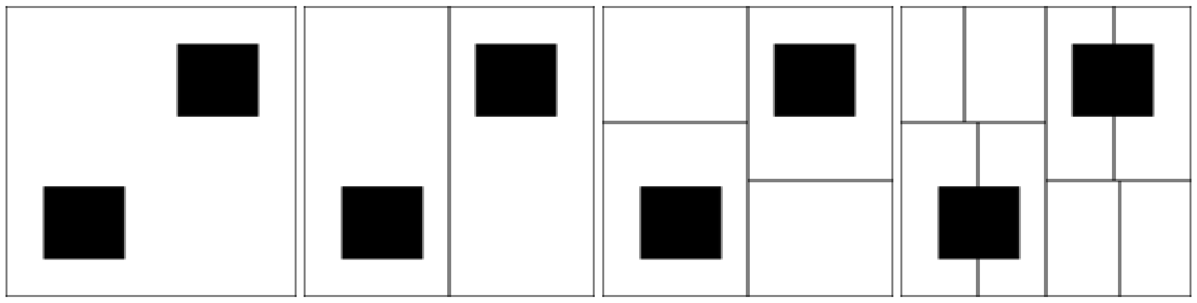


Figure 4.3: An example of a recursive bisection process to a depth of 3

borders of their section with the neighbors. The goal of this proxy application is to be able to evaluate more flexible partitioning and work scheduling. To enable this, we need to compute and keep more information about each section. A *section* refers to a subsection of the *lattice*. For each section a bounding box and a description of which borders neighbors with which section is kept.

By keeping this information per section, a section can be computed by any node. We also unlock the potential for exploring more overlap of communication and computation, as we can initiate communication of a section once it is finished, without waiting for the computation of other sections.

4.4 Work partitioning

In this section we review methods employed to partition the lattice into sections which are distributed onto different processes.

4.4.1 Grid

The lattice is split into rectangles with a predefined size. The size of each section is bounded above by the grid size, though smaller sections can occur if the section size does not evenly divide the lattice size.

4.4.2 Recursive balanced bisection

A recursive traversal is done, where a section is searched in alternating horizontal and vertical direction. The middle is determined and the section split along this axis. Then the process is continued up to a specified recursion depth. The first few steps of such a process are shown in Figure 4.3. This process wil create $2^n, n \in \mathbb{N}$ balanced sections.

When we say *balanced section*, balanced is determined by weights assigned to solid and fluid points which are passed as arguments to the application if this partitioning strategy is used.

4.5 Work scheduling

There are multiple ways to assign the lattice sections to the participating processes. This section describes the methods implemented in the proxy application. Depending on the partitioning

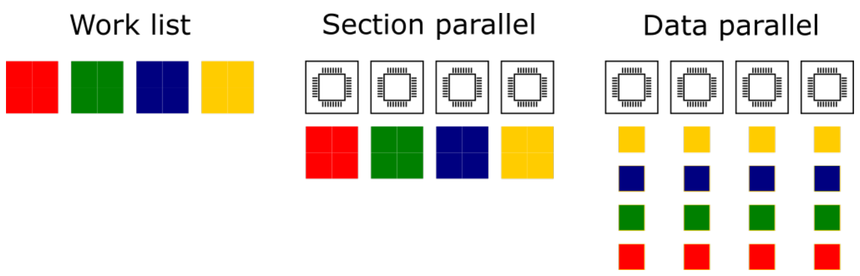


Figure 4.4: Illustration of MPI, MPI+SectionParallel and MPI+DataParallel models. The colored blocks illustrate sections, and how they are distributed to processors.

algorithm used, different scheduling schemes are relevant.

4.5.1 Round robin

Round robin scheduling assigns sections in a circular order. This scheduling is simple to implement. We expect it to be optimal when the sections are balanced and the number of sections evenly divide the number of ranks.

4.5.2 Greedy scheduling

A greedy scheduling algorithm can be used to balance the load when the partitioning does not provide balanced sections. First the cost of each section is computed by a cost function. We explore this cost function further in Section 6.3.

In the *list scheduling algorithm* we assign each section to the process with the least work. This algorithm, without sorting the initial list, is a 2-approximation. A 2-approximation guarantees to create a schedule no worse than 2 times the optimal schedule. If we add the *longest processing time rule* where we sort the list of work by decreasing cost, this becomes a $\frac{4}{3}$-approximation [42].

4.6 Compute kernel

As described in previous section, each section can be computed and exchanged on any rank. Each rank keeps a *work list* which is a list of sections it has been assigned. Collide and propagate are implemented as functions, and as part of the compute step, the application iterates over the work list and invoke the functions on each section. The border exchange is a function that takes the work list and exchanges all borders in parallel.

For the collide and propagate step, we support two different types of parallelism. *Section parallelism* utilizes OpenMP to compute each section in parallel. We refer to this model as *MPI+SectionParallel*. This is to support scenarios with many smaller sections. However, this will not be able to utilize all cores if the number of sections are smaller than the number of cores. Also, this will not utilize the cores if the sections are unbalanced. To enable efficient parallelism for single or few sections, we can allow *data parallelism* where the compute loop of each section is parallelized. We call this model *MPI+DataParallel*. Both can be enabled or disabled by compilation flags. If both are disabled, we describe the model as *MPI*. An illustration of the models is shown in Figure 4.4.

While it is possible to utilize both data parallelism and section parallelism at the same time, this will oversubscribe the cores. We could also utilize task parallelism, which dynamically schedules the task across multiple cores and avoids oversubscription which would result as a combination of data parallel and section parallel. This has however not been implemented in the proxy application.

4.7 Border exchange

While the borders are simple to determine for a traditional grid, they must be computed to allow more flexible partitioning methods. The following algorithm is employed to determine the exchanges.

1. All sections are numbered using integers starting with 0
2. An empty list of send and receive regions are initialized for all sections
3. A integer matrix B with size $W \cdot H$ is initialized with -1
4. For each section, all points along the border are iterated, and the section number the point belongs to is tagged in the matrix S.
5. For each section S, each external border is iterated in the matrix B. All contiguous regions with the same value T in B not equal to -1 is saved as an exchange region. A receive operation is saved for sections S. Conversely a send operation is saved for section B. The send and receive operations are tagged with a common tag value, used by the MPI library to match send and receive operations.

This algorithm will do explicit corner exchanges. When using a strict grid exchange, the corners can be exchanged by using a staged exchange, where one first does north south borders, and then east west. However, to allow for flexible sectioning we do not attempt to exploit such techniques. There is also a special case where a single section can share an entire border as well as corners due to the periodic boundaries. In this case, the first and last points of an edge wraps around because of the periodic edges of the simulation. This leads to situation where a border of n elements should be placed into $n + 2$ elements. Explicit corner exchanges simplify this case.

Also note that while horizontal edges are contiguous regions, the vertical edges are strided, which incurs a packing and unpacking cost on send and receive respectively. The effect of such packing and un-packing on communication is modeled in models such as Log_nP[8], though we ignore it in this case.

We implement periodic borders to ensure conservation of mass. Figure 4.5 shows the borders in the case of a grid partitioning.

4.8 Application validation

We validate the application by simulating flow in a pipe with a wedge. Depending on the angle of the wedge, we expect Moffatt eddies to form. We run the application across multiple nodes for 100.000 iterations and verify that the vortices have formed. Our goal is solely to verify that

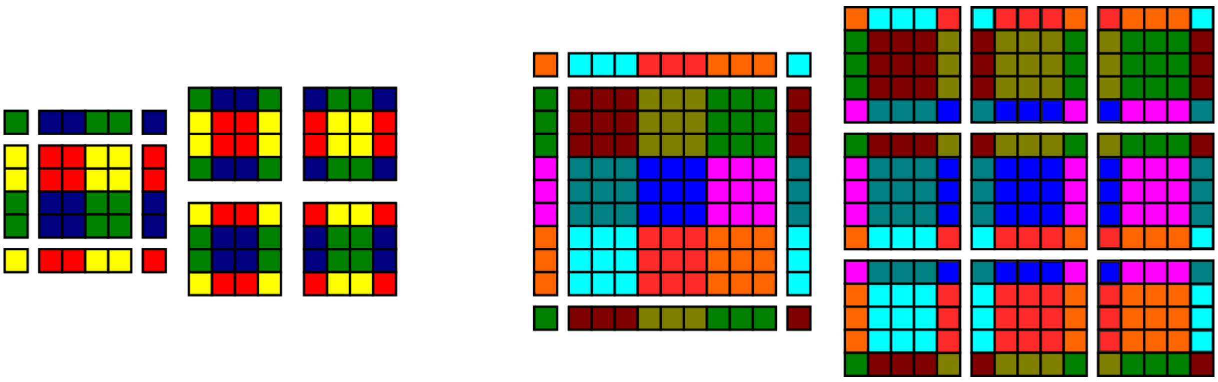
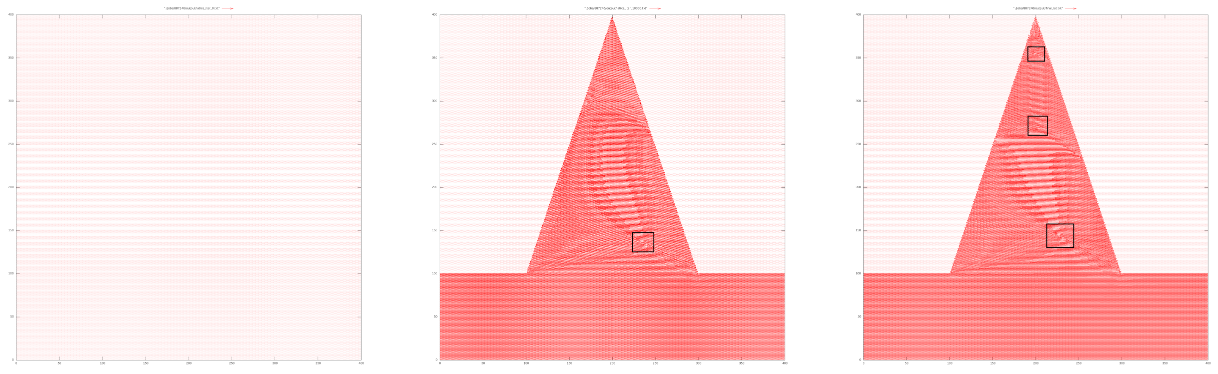


Figure 4.5: How the lattice is sectioned and the borders which will be exchanged for a 4x4 and 9x9 grid.



(a) Initial state

(b) After 10.000 time steps one vortex has formed

(c) After 100.000 time steps three vortices have formed

Figure 4.6: Simulation of fluid flow in a pipe with a wedge

the computation produces a valid result, and from that we conclude that the application captures the computational pattern of the LBM method. For later cases, we do not inspect the output of the simulation. A sample of the output from the simulation is shown in Figure 4.6.

Chapter 5

Models

In this chapter we develop the performance model of the proxy application. We apply a top down modeling approach, and start with the fundamental equation of modeling

$$T_{tot} = T_{comp} + T_{comm} - T_{overlap} \tag{5.1}$$

In Section 5.1 we consider the parameter space of the model. Section 5.2 develops a model for the computational aspects of the proxy application. In Section 5.3 the communication models are described. We do not have any overlap, and set $T_{overlap} = 0$.

5.1 Parameters

When modeling, there is a trade-off between accuracy and general applicability. To avoid tying the model to the implementation, and allow general applicability, we do not want to introduce too many variables. Because of this we use the approach of benchmarking the system to derive performance characteristics of the implementation on the platform. As fluid and solid points require different treatment in the computation we let the model distinguish between the two. The application also treats border points, and points where we apply force differently. This could imply adding another two parameters to the model. We do however ignore these terms, and do not apply force to any points.

This proxy application works on a list of sections per rank. To capture this, we could also extend the model with the number of sections and a overhead per section. This overhead can come from factors such as cache misses and more irregular data access patterns, as well as an extra function call for the propagate and collide steps. We do, however, not introduce a term for this effect, as we assume the compute time will dominate such factors. We validate this assumption in Section 8.2.1.

5.2 Computation

In previous work[30] a model using the number of lattice points were used, and the system benchmarked to determine a time per lattice point. In a pre-study to this thesis this model was was found to be too inaccurate if the ratio of fluid to solid points in the section differed between the processes. This also affects the communication time, as some processes spent most of their time waiting for the border exchange, while others were busy computing. We will try to capture this effect to enable better communication predictions as well.

5.2.1 Model

The computation time consists of the collision and propagate time.

$$T_{compute} = T_{collide} + T_{propagate} \tag{5.2}$$

Collide

The collide operator contains conditional branches for solid and fluid points. We express the model in terms of these two terms.

$$T_{collide} = n_{solid} \cdot C_{solid} + n_{fluid} \cdot C_{fluid} \tag{5.3}$$

This relates to section size width W and height H through the porosity ϕ as

$$n_{solid} = W \cdot H \cdot (1 - \phi) \tag{5.4}$$

and conversely

$$n_{fluid} = W \cdot H \cdot \phi \tag{5.5}$$

The parameters C_{fluid} and C_{solid} are the costs of computing a fluid or solid lattice point respectively.

Propagate

The propagate steps treat all points equally, and thus only needs one term. However, it also needs to propagate the points exchanged during border exchange. We express the propagation cost as

$$T_{propagate} = (W + 2) \cdot (H + 2) \cdot C_{propagate} \tag{5.6}$$

5.3 Communication

In this section we develop the communication models used. Our goal is to capture the communication performance, and be able to evaluate an application model. We do not attempt to develop analytical models for the communication, but express the cost of MPI calls in terms of a model for simulation.

Because we cannot assume network homogeneity, all models are given for a pair of nodes in the cluster. To simplify the testing, we will assume symmetric connections. We also assume a sequential mapping, in other words that given p processes per node, process i is mapped to node $\frac{i}{p}$ and processor $i \mod p$.

Routine	Cost
MPI_Send	0
MPI_Isend	0
MPI_Recv	$max\{t_r - t_s, \alpha + \beta \cdot m\}$
MPI_Irecv	0
MPI_Wait	T_{blk}

Table 5.1: MPI calls expressed in terms of the modified Hockney model. T_{blk} is the cost of the corresponding blocking call.

Name	Expression
T_1	$o' + kO_s$
T_2	$kG_s + L$
T_2'	$sG_s + (k - s)G_k + L$
T_3	$o' + kO_r$
T_4	$max\{o' + L, t_r - t_s\} + o'$
T_5	$o' + L + o'$

Table 5.2: Common sub expressions for LogGPS model

5.3.1 Hockney model

The Hockney model was originally used to model a single transfer. Our proxy application uses overlapping communication with multiple concurrent transfers. To model MPI non-blocking operations using the Hockney model, some assumptions must be made.

Our goal is to use the Hockney model as a benchmark and the key we wish to identify is the synchronization cost of ranks reaching the communication step of the computation at different times. To apply the Hockney model to non-blocking MPI operations we need to define `Isend`, `Irecv` and `Wait`. To model this, we define both the `Isend` and `Irecv` to take 0 time. Also, waiting for a send takes 0 time. Only waiting for a receive takes time. The wait for a receive computes the time between the call to wait and send, and either takes 0 time, or the time remaining of the blocking time. Modeling `Wait` as the corresponding blocking call is inspired by how the LogGPS models these transfers, though we ignore all overheads. `Recv` takes the remaining time of a transmission in the Hockney model. Doing this we expect the Hockney model to be able to capture synchronization of the ranks, but not model the actual time of a transfer. The resulting model for MPI calls are shown in Table 5.1.

5.3.2 LogGPS model

The Hockney model is not made for modeling of middleware such as MPI. The LogGPS is a model that attempts to capture the synchronization cost of the rendezvous protocol in MPI. Our goal is that the LogGPS model is able to more accurately capture the actual cost of the MPI routines.

The MPI operations used in the proxy application is described in terms of the LogGPS model in Table 5.2 and 5.3. The tables are reproduced from [22] for reference.

Routine	Condition	Cost
MPI_Send	$k \leq S$	T_1
	$k > S$	$T_4 + T_5 + T_1$
MPI_Isend		o'
MPI_Recv	$k \leq s$	$max\{T_1 + T_2 - (t_r - t_s), 0\} + T_3$
	$s < k \leq S$	$max\{T_1 + T_2' - (t_r - t_s), 0\} + T_3$
	$s > S$	$max\{o' + L - (t_r - t_s), 0\} + o' + T_5 + T_1 + T_2' + T_3$
MPI_Irecv		o'
MPI_Wait		$max\{T_{blk} - (t_i - t_w), o'\}$

Table 5.3: MPI Routines expressed in terms of the LogGSP model using expressions in Table 5.2. Reproduced from [22]

Chapter 6

Performance optimization

In this section we investigate and propose measures of the input geometry that can be used to improve the performance of the simulation. In Section 6.2 we determine the parameter space. Section 6.1 clarifies the notation used. Section 6.3 introduces cost functions that are utilized both by partitioning and scheduling and consider their applicability and potential. Section 6.4 discusses using porosity to optimize performance.

6.1 Notation

We briefly repeat and clarify some terms used. The *lattice* refers to the entire input geometry after it has been converted from a bitmap image to D2Q6 lattice through the even row shift described in Section 4.2. A *section* refers to a subsection of the *lattice*.

We define the *global porosity* to be the porosity $\phi = \frac{V_{void}}{V_{total}}$ of the lattice. The *local porosity* is the porosity for a single section determined by the partitioning. In the case where a single node compute multiple sections, we can note this as the *node porosity*. These measures all refer to the relationship of how many fluid points there are versus total points.

6.2 Parameters

We identify the parameter space for optimization to consist of the following aspects:

- Geometry
- Partitioning
- Scheduling

The geometry encompasses porosity, porosity distribution and other such factors. How this geometry is partitioned impacts the performance. Once partitioned, how the sections are scheduled onto nodes also affect the performance. The scheduling aspect also captures the number of nodes participating in the computation.

6.3 Section cost

Both the traditional grid partitioning as well as the possible balanced partitioning techniques depend on a notion of the section cost. We define three cost functions:

- *Size* - The size of the section as a cost function.
- *Analytical* - The local porosity or node porosity.
- *Model based* - The parameters of the compute model as cost for each point type.

An immediate observation is that all three can be implemented using the recursive balanced partitioning, but with different weights assigned. Size is simply assigning the same weight to all points, and is equivalent to a grid for grid sizes that evenly divide the lattice size.

6.3.1 Assumptions

The three functions above are applicable given different assumptions about the compute cost of each point.

Size

The assumption for partitioning by size is that

$$C_{fluid} \approxeq C_{solid} \tag{6.1}$$

It is also the simplest partitioning strategy. The structure/sectioning can be embedded in the program, which simplifies border exchange, loading input, writing output and more. The assumption is that the cost is sufficiently equal that the benefit of simplifying other program logic outweighs any difference.

Analytical

This assumes that

$$C_{fluid} \gg C_{solid} \tag{6.2}$$

In other words, this assumes that the cost of fluid points dominates the cost of solid points. This is based on observation of the LBM method, and the fact that the border points are not computed, but only reflect any incoming force back. We do not consider the case of solid point being more computationally costly than fluid, as we expect it should be at least no higher than the fluid cost.

Model based

The model based takes the model into account, which is based on the application. This assumes that though we are focused on the fluid points, the application as it is written and described in Section 4 still iterates all points. However, we still assume that the cost of fluid points is greater than the cost of solid points. Sufficiently so as to not be the *Size* case described above.

$$C_{fluid} > C_{solid} \gg 0 \tag{6.3}$$

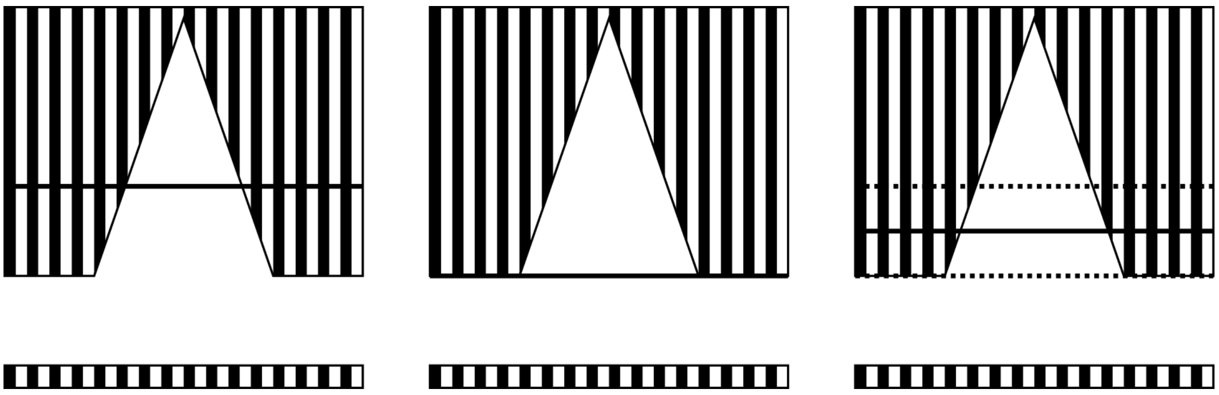


Figure 6.1: The three cost functions for partitioning applied to Moffatt geometry. Striped is solid, white is void.

6.4 Porosity as proxy for compute balance

Assuming a grid partitioning strategy is used, all sections will be equally sized. Traditionally, the grid size will be selected so that every node has one section to compute. Under these assumptions, and the analytical cost described above, we hypothesize that the local porosity can be used to describe the computational balance between nodes. If any node has a lower local porosity, it will have to remain idle waiting for the other nodes.

To illustrate this, we start with the Moffatt geometry used for validation of the application. The three sectioning techniques applied to the moffatt geometry is shown in Figure 6.1. Here, we assume that we are to split this geometry in two along the horizontal axis.

We see how the grid partitioning will create sections which are unbalanced in terms of number of fluid points, under the assumption that the LBM solver should only need to compute void space. Using a balanced partitioning where we balance the number of fluid points results in a sectioning where the size is uneven. Finally, we can imagine the cost being somewhere in between. Depending on the relative cost, we can see that the cut must be placed between the extremes of the grid and analytical assumption.

Chapter 7

Experimental setup

In this chapter, we review the experimental setup used in this thesis. In Section 7.1 the relevant characteristics of the compute platform used in this thesis is reviewed. Section 7.4 describes the simulator used to compute the expected run-time of the proxy from the models described in Chapter 5

7.1 Platform

Idun is a cluster owned by a collection of institutes at NTNU, and operated by NTNU IT. The cluster consists of several different types of nodes, with some providing access to NVIDIA GPUs. The cluster uses a Mellanox Infiniband interconnect with fat tree topology. In this thesis, a homogeneous subset of the nodes is used.

7.1.1 Nodes

The nodes are Dell PE630 machines. The Dell PE630 has two sockets, each having a Intel Xeon E5-2630 v4 10-core processor. The E5-2630 v4 has a base clock speed of 2.2GHz with 25MB of on chip cache.

The node is also equipped with 128GB ram, 64GB per NUMA region. A graphical description of the processors on Idun is shown in Figure 7.1.

7.1.2 Interconnect

Idun is organized as a fat-tree, though all nodes used in this thesis are connected to the same switch. This makes it effectively a star topology. The connections are all Mellanox Infiniband 4xFDR links, which have a theoretical bandwidth of 54.54Gbit/s.

7.1.3 MPI

Idun has both OpenMP and IntelMPI available. In this section, we review properties of both on Idun. This is because it has a large impact on the modeling and performance of the application.

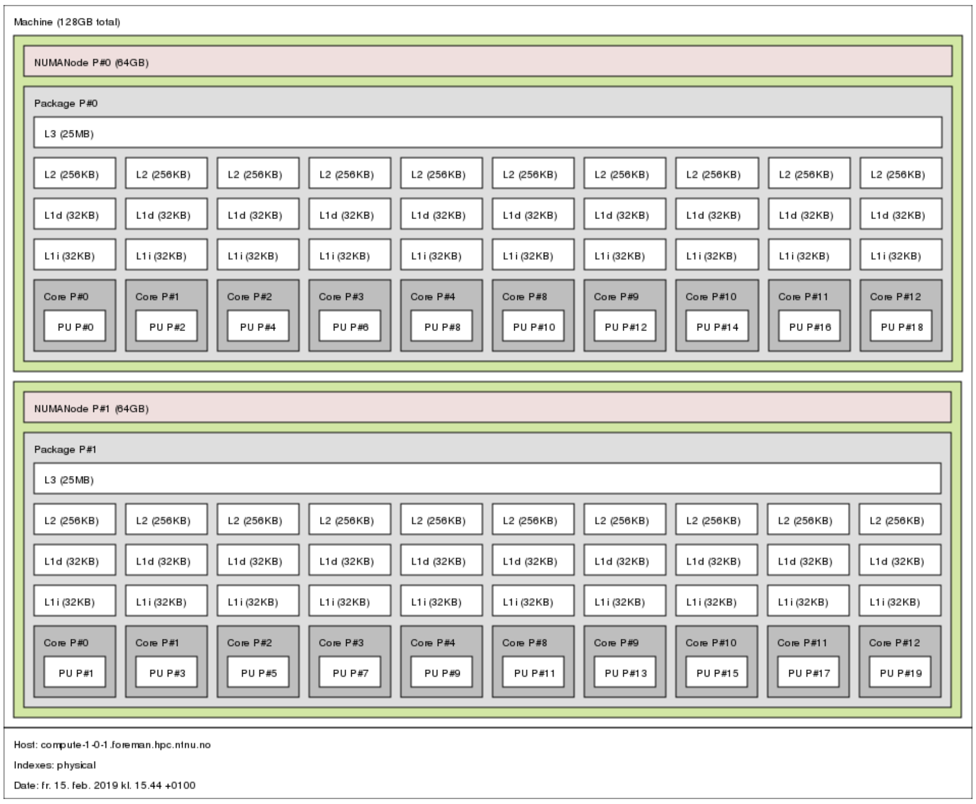


Figure 7.1: The organization of processors and cores on the Dell PE630 nodes on Idun. Image produced by *hwloc*[6]

OpenMPI

OpenMPI uses UCX, a communication library which supports RMA, tag based message matching and implementation of eager and rendezvous protocols. UCX is also used by other projects such as MPICH and Charm++, supports multi rail, and direct memory transfers for GPU-GPU[38].

However, OpenMPI is not built with support for the work manager Slurm on Idun. The result of this is that MPI+OpenMP application may experience issues where the application is unable to utilize the available cores for OpenMP.

Intel MPI

Intel MPI does not use UCX, but utilizes underlying communication APIs directly. Up to version 2019.x, Intel MPI utilizes the DAPL interface for communication on Infiniband. From version 2019, a new library provided by the open framework alliance will be used. The latest version available on Idun however is 2018.5.288, meaning the DAPL layer is used by default. One downside of this is that the DAPL layer introduces overhead in the communication. Figure 7.2 shows the transmission time of a ping-pong test. We see a significant jump in communication time at 128kB. Because of this, we configure Intel MPI to use the open framework interface fabric. This utilizes the verbs API directly and behaves more predictable in terms of eager and rendezvous protocol switches.

The threshold for the eager and rendezvous threshold is by default 256kB, but can be configured using the `I_MPI_EAGER_THRESHOLD` environment flag. We use this flag to explicitly

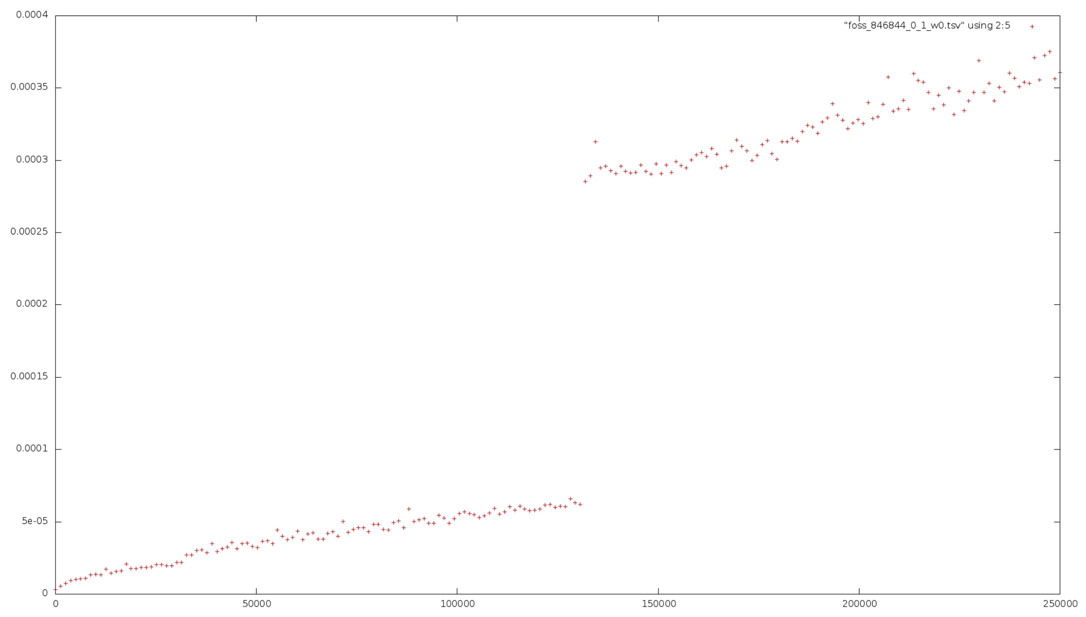


Figure 7.2: Transmission times for ping-pong test as part of determining LogGPS parameters over DAPL.

set it to the default value of 256kB to ensure it is a known value.

7.2 Timing

Timing the code has been done using `std::chrono::steady_clock`. This is part of `C++11` and provides a clock that is guaranteed to be steady, though it does not state a required precision. Irrespective of resolution, all measurements are stored as nanoseconds.

For timed sections, the current time is taken before the section, the code is executed, and then a second timestamp is taken. The difference is stored in a preallocated vector.

All timing code in the main loop can be enabled or disabled through compilation flags. To avoid including timing overhead, we only allow timing separate steps in the loop, or the entire loop, not both at the same time. We assume that all operations timed are sufficiently long for the overhead of the clock to be insignificant. When possible, we time multiple iterations, and compute the time per iteration using arithmetic means rather than individual measurements.

7.3 Determination of model parameters

This section describes the procedures used to determine parameters for the model.

7.3.1 Compute model

To ensure that we keep the physics correct, as we otherwise risk the simulation becoming unstable and ending up timing divide by zero faults in the compute kernel, we time the propagate and collide steps separately, while doing a border exchange.

To fit the model we run two different porosity configurations with 30% and 70% porosity. for both 400x400 and 4000x4000 lattice points. For each run 400 iterations are completed and each step, collide and propagate, is timed separately. We solve this as a set of linear equations, and fit using least squares method.

7.3.2 Communication parameters

In this section we review the methods used to determine the parameters for the communication models.

Hockney

A ping-pong test is run and the plot is inspected to detect cut-offs. We also know the eager rendezvous threshold from the documentation and expect a threshold here. Then, each identified segment is fitted to a line.

LogGPS

To determine the LogGPS parameters, a slightly modified ping-pong test is conducted. Between the send and receive on the initiator, a configurable delay is inserted. The ping-pong test is conducted with the delay w set to 0 and to a number greater than the communication time of a single message. The value of w should be chosen large enough to exceed the time of a single message. A too large w will result in the measurement procedure taking a significant amount of time. In this work we have used a value of $w = 0.005$.

The resulting times are fitted to lines for each interval of $0, s$, s, S and $S, \inf$. The gradient and intersection of these lines are used together with a set of equations from the model, and solved as a linear set of equations. For further details, we refer to [22].

7.4 Simulation

To compute the communication times using the models, we develop a simple simulator. The simulator takes a list of events as input, and computes the time of the computation and each MPI call. The simulator works by computing events for a single rank at a time. If simulation reaches an unresolved dependency, a *non-local* MPI call, it stops computation of that rank. It then proceeds with the next rank. This repeats until all events have been computed. This simulation only computes the time of a single iteration.

Chapter 8

Results

In this chapter we review the results from the experiments described in the previous section. Section 8.1 notes experiments used to characterize the performance of the proxy on the platforms used. Section 8.2 builds and evaluates the performance model. Section 8.3 applies the model to evaluate the assumptions discussed in Section 6.

8.1 Benchmarking proxy application

In this section, we benchmark the proxy application to determine which factors impact performance.

8.1.1 Setup

In this section we benchmark the three steps of the setup phase. The four key steps are image load, partitioning, border resolution and scheduling.

Image load

The image is loaded by the main rank from a network location. It is the most expensive operation, but it is important to note that it is not optimized in any way. The image format was chosen to be human readable and not optimized for speed. Figure 8.1 shows how it scales with the image size, and thus the file size. We do not attempt to determine how much of the loading times that is network versus compute time.

Partitioning

The partitioning is mainly influenced by two factors. We see that the grid partitioning has a fixed overhead which is not significantly impacted by the input size.

The recursive bisection on the other hand is affected by input size and the depth. From Figure 8.2 we see that the depth influence appears to be negligible.

Border resolution

Figure 8.3 shows the time to determine the border exchanges for two different lattice sizes as the number of sections increase. We see that the size of the lattice impacts the time, but that the

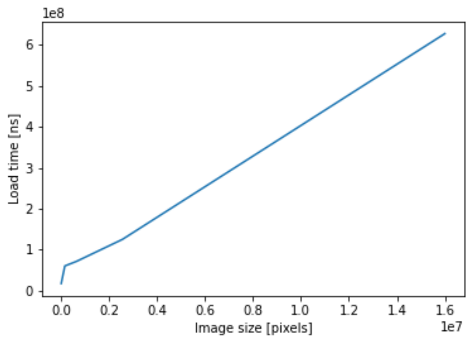
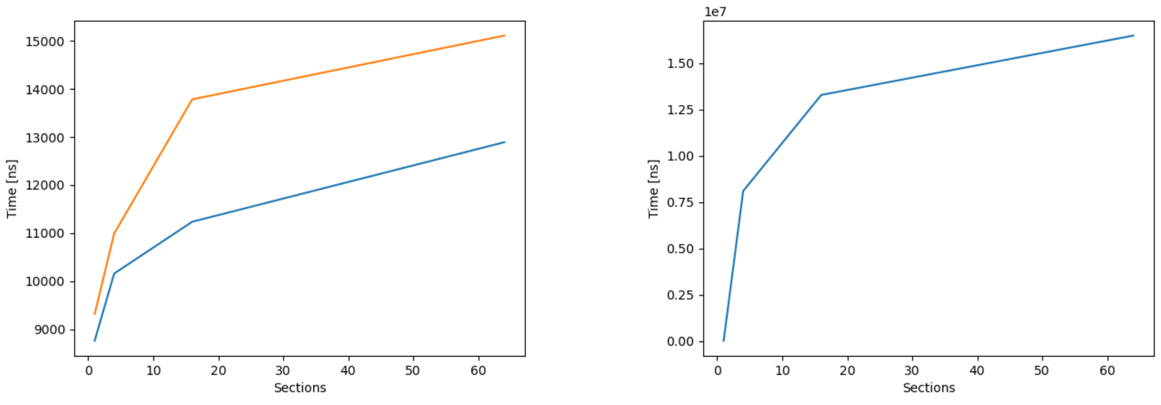
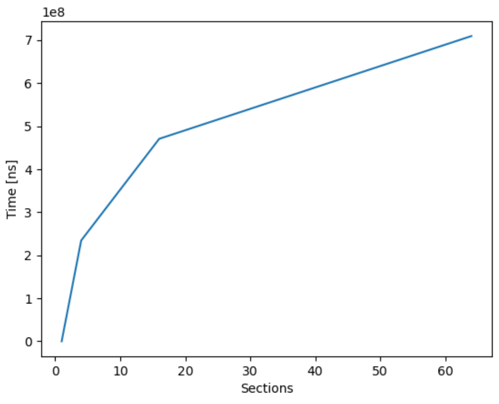


Figure 8.1: The load time of the image as it scales with the image size. Note the scale in the upper left-hand corner.



(a) Grid partitioning

(b) Recursive partitioning on 400x400 input



(c) Recursive partitioning on 400x400 input

Figure 8.2: Times for partitioning. Note the difference in y-axis scales, and the scale in the upper left-hand corner.

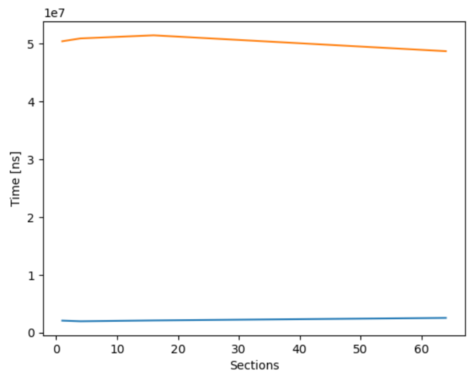


Figure 8.3: The time to determine border exchanges as section count increase. Note the scale in the upper left-hand corner.

number of sections does not contribute to the time.

Scheduling

The time to schedule the sections onto ranks for the three scheduling methods are shown in Figure 8.4. We see that the time is negligible for both round robin and greedy based on size. However, the time is significant for the model based. This is in large due to the fact that it must iterate the entire lattice in order to assign costs.

8.1.2 Computation

In this section we characterize the performance of the steps of the compute kernel.

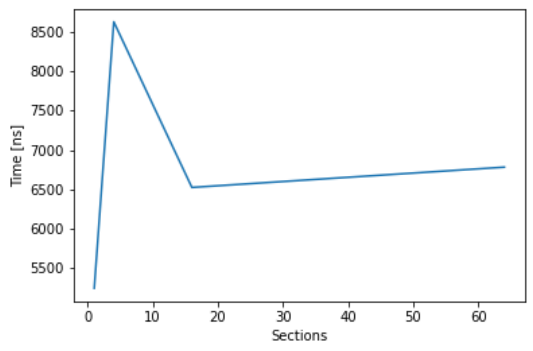
Parallel model and section size

We start by investigating how the number of sections impact the compute performance. As described in Section 4.6 we consider three different parallel models.

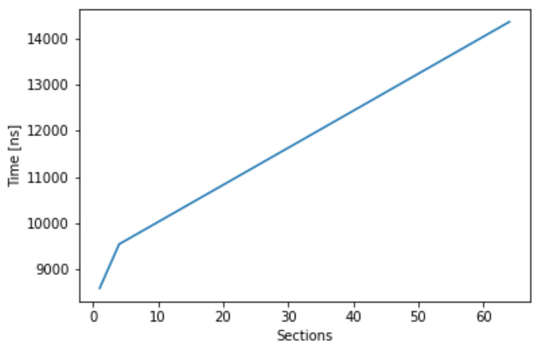
- MPI - This model relies only on message passing
- MPI+DataParallel - This model parallelize the computation of the points in each section.
- MPI+SectionParallel - This model parallelizes the computation of sections. Each core processing a single section.

Figure 8.5 show MPI, MPI+DataParallel and MPI+SectionParallel for both one and two ranks on a single node. When using two ranks, processes, each rank has access to 10 cores. When using a single rank, it has access to all 20 cores. We study this variation as we expect the NUMA regions to impact performance.

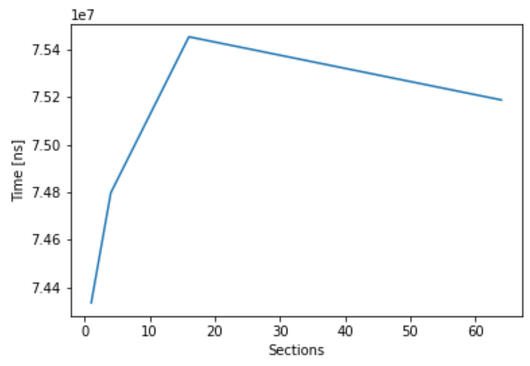
The y-axis shows the computation time in nanoseconds. All runs use a lattice with 4000x 4000 fluid points. The tested section counts all evenly divide the lattice dimensions. We see that for all sections shown, the data parallel version performs the best. We see that we need at least two sections to utilize two ranks across all configurations. However, only MPI+DataParallel



(a) Round robin

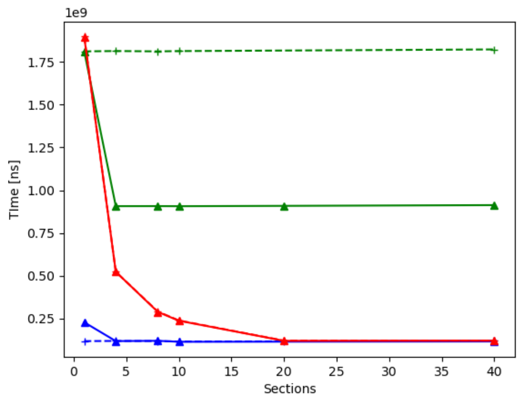


(b) Greedy based on size

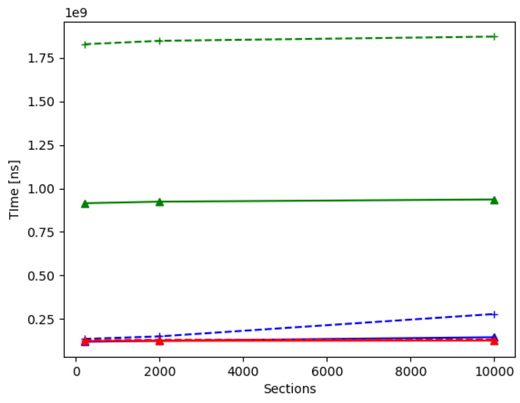


(c) Greedy based on model parameters

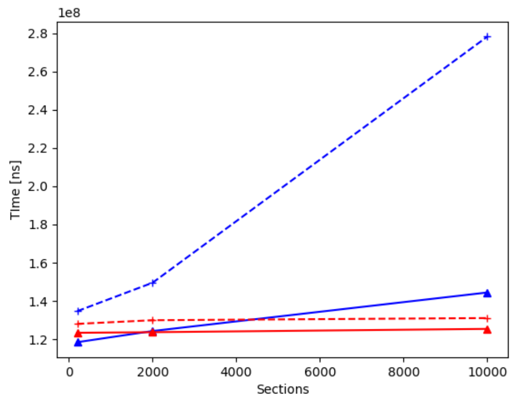
Figure 8.4: The time to schedule sections onto ranks for: round robin, greedy scheduling based on size, and greedy scheduling based on model weights.



(a) Scaling for small number of sections



(b) Scaling for large number of sections



(c) Scaling for large number of sections. Only MPI-+DataParallel and MPI+SectionParallel plotted.

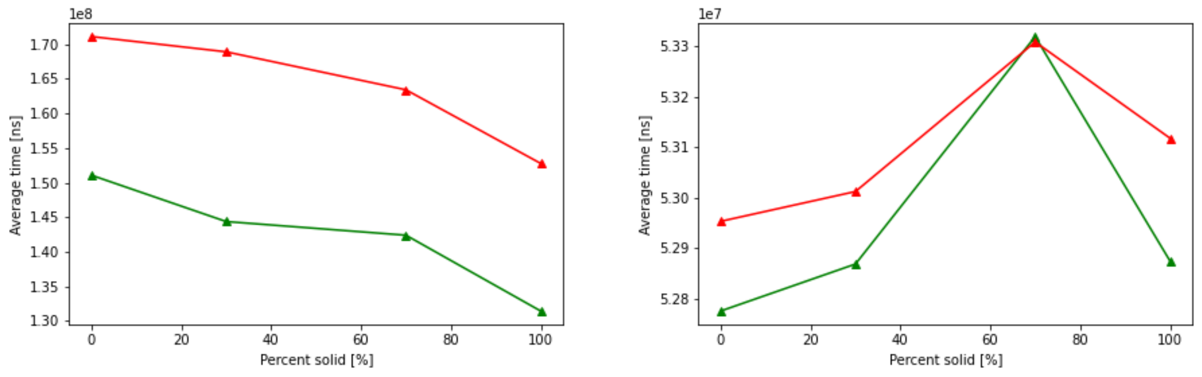
Figure 8.5: The compute times of different configurations. Green is MPI, blue is DataParallel, red is SectionParallel. Dashed with cross is single rank, solid with triangle is two ranks.

is able to fully utilize all cores for low sections counts. Once the section count is 20, we see MPI+SectionParallel reaching the same performance.

When we take this to very small sections we see that eventually, the overhead of fan out/in for each section is noticeable for the MPI+DataParallel, while the MPI+SectionParallel stays constant, with little performance penalty for small sections. Figure 8.5 shows the scaling to 10.000 sections (40x40 lattice points). The performance penalty of many small sections is more pronounced for the single rank configuration where all 20 cores are used for each section.

8.2 Model Validation

In this section, we validate the models ability to predict the runtime of the proxy application. Section 8.2.1 reviews the computational model. Section 8.2.2 reviews the communication models described in Section 5.3. Finally, Section 8.2.3 evaluates the accuracy of computation and communication together.



(a) Collision **(b)** Propagate

Figure 8.6: Green line show the time when force is applied to all points. Red line does not have force applied to any points.

Parameter	Value [s/lattice point]
C_{fluid}	9.28023964e-09
C_{solid}	8.53028999e-09
$C_{propagate}$	2.54048376e-09

Table 8.1: Compute model parameters on Idun

8.2.1 Computational model

The computational model estimates the running time of the propagation and collision steps of the LBM method. In this section, we review the experiments conducted to validate the computational model described in Section 5.2.

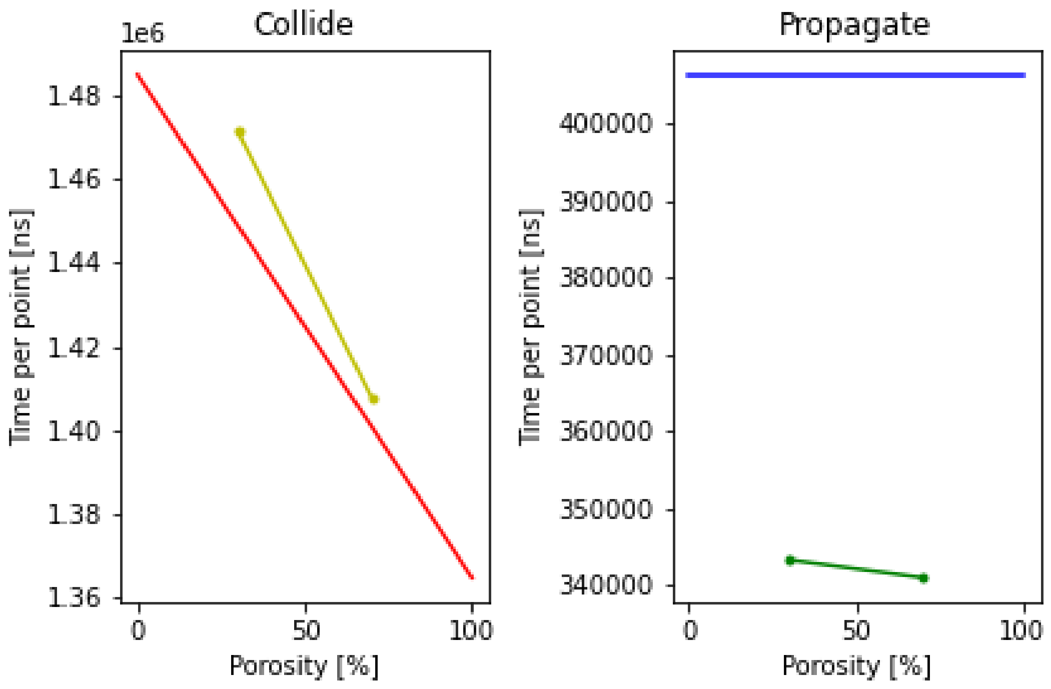
From the tests in Section 8.1.2 we found the data parallel to be on par with other solutions except for a high number of small regions. We determine to use the MPI+DataParallel version, as this is most likely to perform well across a range of configurations. If we were to use a large number of small sections, we could instead use the MPI+SectionParallel configuration. In this case, we would determine compute parameters for that configuration as well.

Figure 8.6 show the compute time, for varying percentage of solid to fluid lattice points, with force applied to all points and no points. We find that both force, and point type are significant for the computational time. We see that only solid points with force is equal to only fluid points, but with no force. The model in this work does not include this factor, but the model could be extended with an additional parameter, and fitted using the same method.

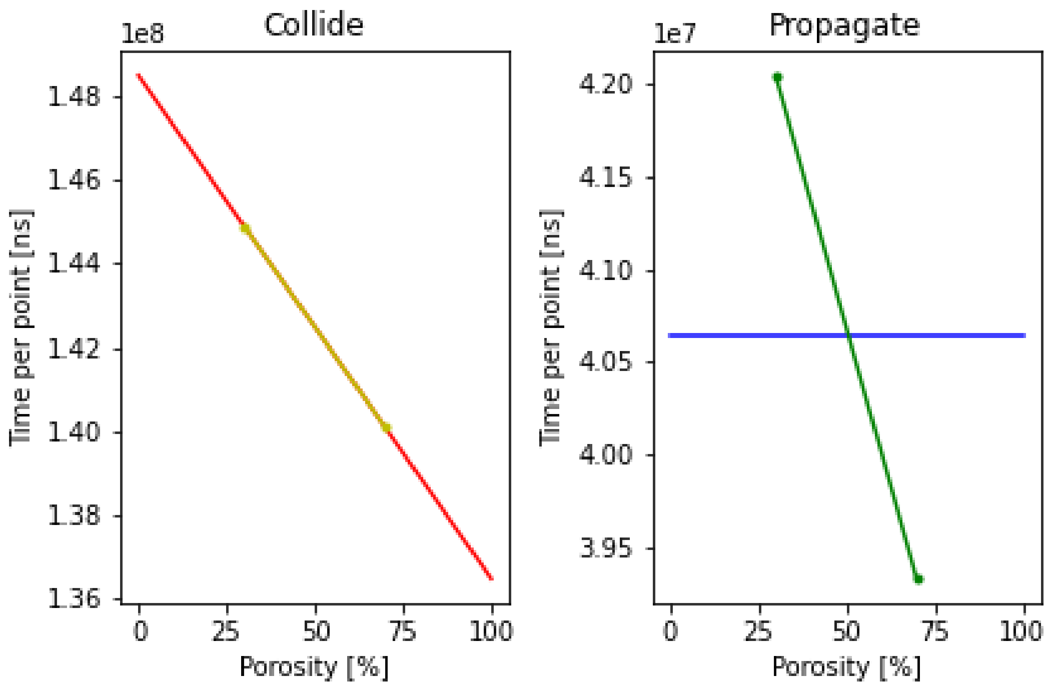
Using the method described in Section 7.3 we determine the parameters of the compute model to be as shown in 8.1.

8.2.2 Communication model

Depending on the input size, relatively little time is spent in communication compared to compute. The communication phase does, however, impact performance in two important ways. While partitioning the input into many small sections can enable better balancing, it also increases the need for border exchange. Additionally, the communication effectively synchronizes all ranks. Because of this, the communication time is in large the synchronization time.



(a) Fit for 400x400 lattice points



(b) Fit for 4000x4000 lattice points

Figure 8.7: Compute model parameter fit

Start	End	α	β
0	65536	1.1975672273539768e-06	2.1076591315635432e-10
65536	+inf	7.8348843121862e-11	5.735000605451329e-07

Table 8.2: Model parameters for intra node communication on Idun

Start	End	α	β
0	8192	3.143603959120105e-06	3.1255977885458683e-10
8192	32768	5.195300855509677e-06	1.2879242631433513e-11
32768	131072	1.381784036151631e-07	1.6183214427981522e-10
131072	+inf	9.000180299942776e-05	2.0385484230679632e-10

Table 8.3: Model parameters for inter node communication on Idun

In this section we review the communication models described in Section 5.3 and evaluate if they are able to capture the synchronization as well as accurately capture the impact of many sections of the communication time.

Hockney

Figure 8.8 shows the communication times for eight ranks running on four nodes. Each row i show communication tests originating from rank i. Columns correspond to process $i + 1$. We find two distinct communication patterns. They are the inter and intra node times. The Hockney model parameters derived from these measurements are listed in Table 8.2 for intra node communication and Table 8.3 for inter node. Since the interconnect effectively is a star, and all nodes are equal, we only determine parameters for inter and intra node, rather than for every pair.

LogGPS model

As described in Section 7.1.3 we use the ofa fabric instead of using the default DAPL interface of Intel MPI. Figure 8.9b show the communication time for the w_0 case on Idun. Figure 8.9a show the case for intra node communication. From the two graphs we see the S at 256kB.

The LogGPS parameters for intra and inter node on Idun is shown in Table 8.4.

What we see there is that the LogGPS measurement procedure does not work well for the intranode case. While intra node communication still used MPI it does not use the ea-

	Intranode	Internode
s, S	1024, 264144	2048, 262144
L	-3.7721341976527483E-07	1.07691911298802E-06
o’	5.7322726392455359E-07	9.1712012742516981E-08
o_s	2.8705119913329383E-10	2.0590874495359647E-10
o_r	-1.499123815304E-10	-6.7664361390937842E-12
g_s	1.5755309856477086E-10	4.9346921813160516E-10
g_l	1.5093994794941992E-10	6.6792565312476043E-12

Table 8.4: LogGPS parameters for intra and inter node

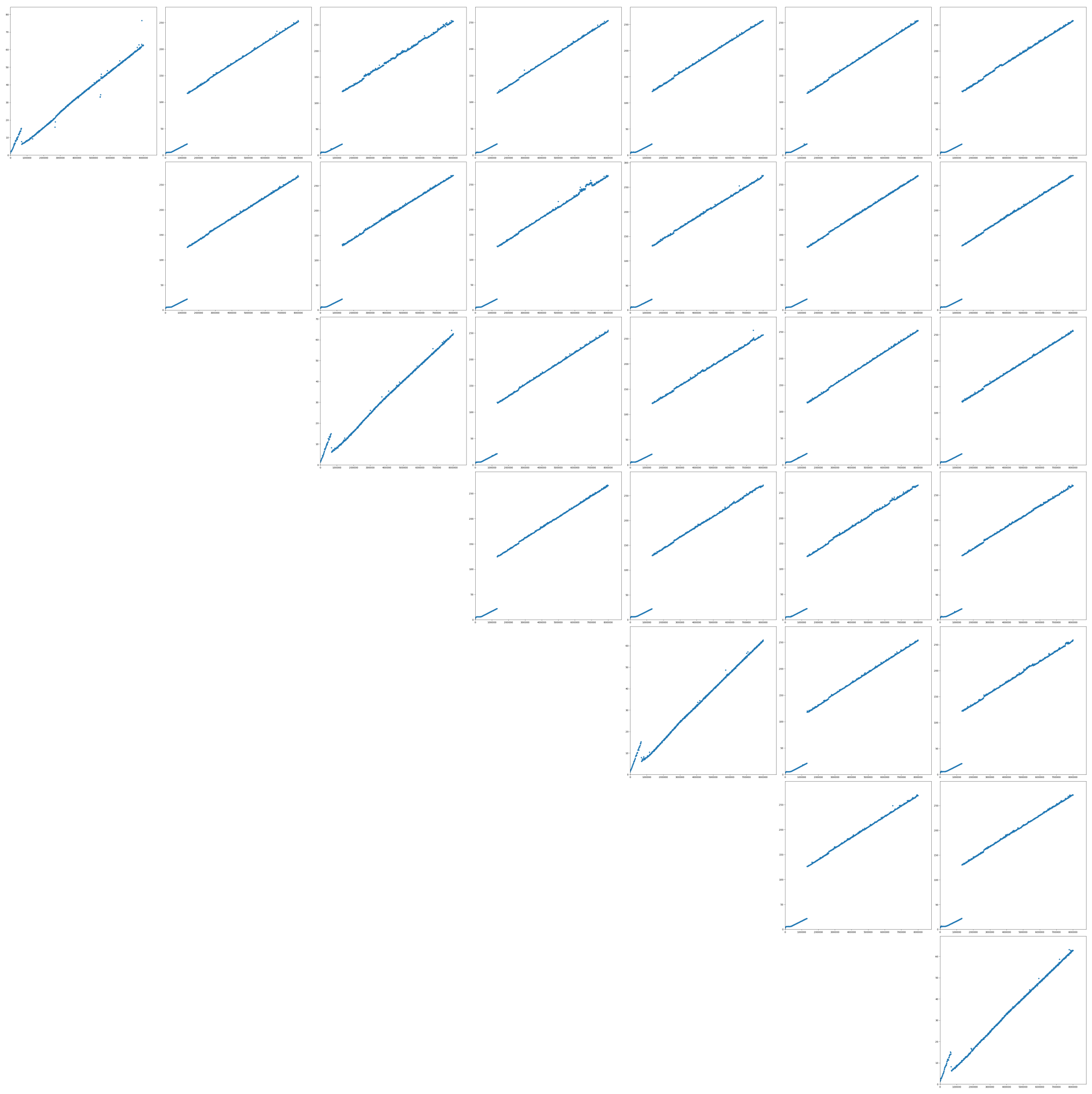
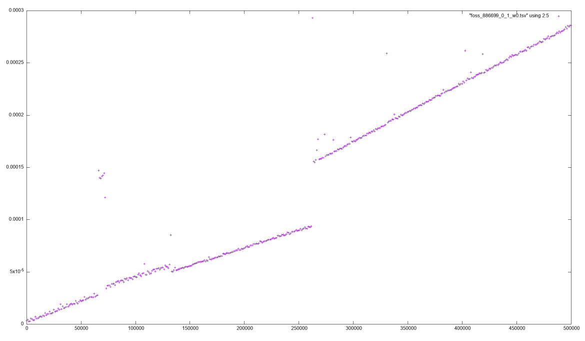
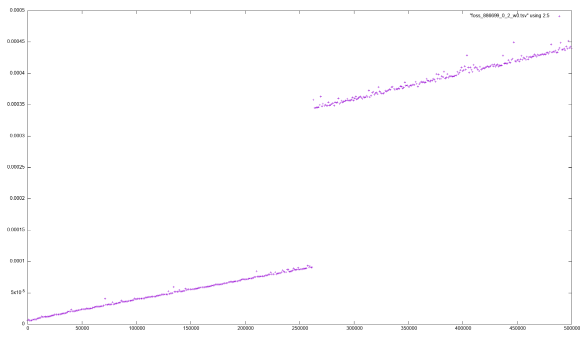


Figure 8.8: Communication matrix for 4 nodes on idun, with two ranks on each. We find the communication times are similar for all internode and intra node communication



(a) LogGPS benchmark intra node



(b) LogGPS benchmark inter node

Figure 8.9: Communication times for a LogGPS benchmark with $w = 0$.

ger/rendezvous transfer mechanisms. Instead a copy through shared memory is used. This makes the transfer progress in a way that causes the L to become negative. We also see that the measurement of o_r becomes negative for both inter and intra node.

Communication model accuracy

We evaluate the accuracy of the communication predictions by inserting a barrier before the communication. This will not perfectly synchronize the processes, but sufficiently to assume equal starting time. The actual versus predicted time using the Hockney and LogGPS model for a single rank (rank 0) for the following configurations are shown in Figure 8.10. All configurations are run with two nodes with two ranks per node.

- Quarter geometry
  - Four sections of size 400x400 per rank - Grid partitioning
  - One section of size 800x800 per rank - Grid partitioning
  - One section per rank, balancing fluid in each section
- Moffatt geometry
  - Four sections of size 400x400 per rank - Grid partitioning
  - One section of size 800x800 per rank - Grid partitioning
  - One section per rank, balancing fluid in each section

We see that the predicted time is underestimated by both models, but the LogGPS model does provide slightly better estimations than the Hockney model. We see that the grid partitioning behaves the same way, and that increasing the number of sections also increases the actual communication time by a factor of four. This leads us to believe that the communication may be dominated by send and receive overhead, rather than bandwidth.

Both configurations 1 and 4 and 2 and 5 have the same number of sections of the same size. However, configurations 3 and 6 are partitioned according to balancing fluid points in each section. Because of this, configuration 6 receives a larger section than configuration 3, and we see that the result is longer communication time. However, it is not possible to determine if it is caused by bandwidth or larger sending overhead.

This effect could also be due to contention, as many messages are posted at the same time. This effect can be countered by the network, as multiple consecutive small messages in Infiniband networks has a lower cost. This is modeled and described in the LogfP model[20].

While we see that the models underestimates the communication times with what may appear as a constant, we do not want to determine such a constant and multiply by it. This is because one of the important aspects of the model is to capture the synchronization time, which cannot be multiplied by such a constant.

8.2.3 Complete run

To predict the time of the application, we use the simulator developed and simulate an iteration of the program, this is the *predicted* value. We compare the predicted value with the average of all iterations of an actual run. We call this average time for each rank the *actual* time.

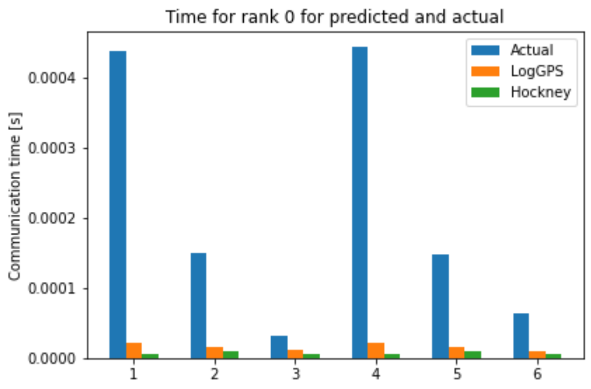


Figure 8.10: The communication time for the described scenarios versus actual.

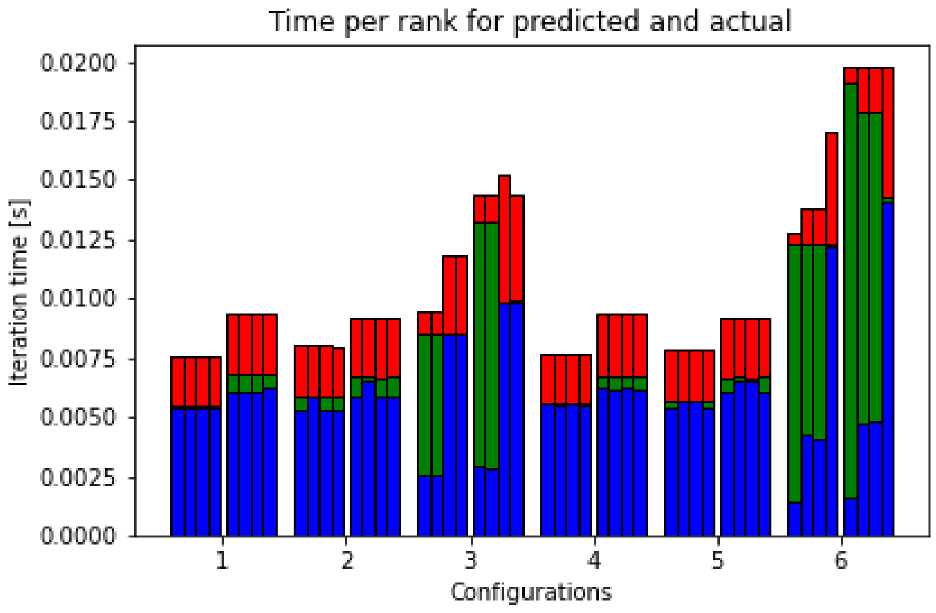
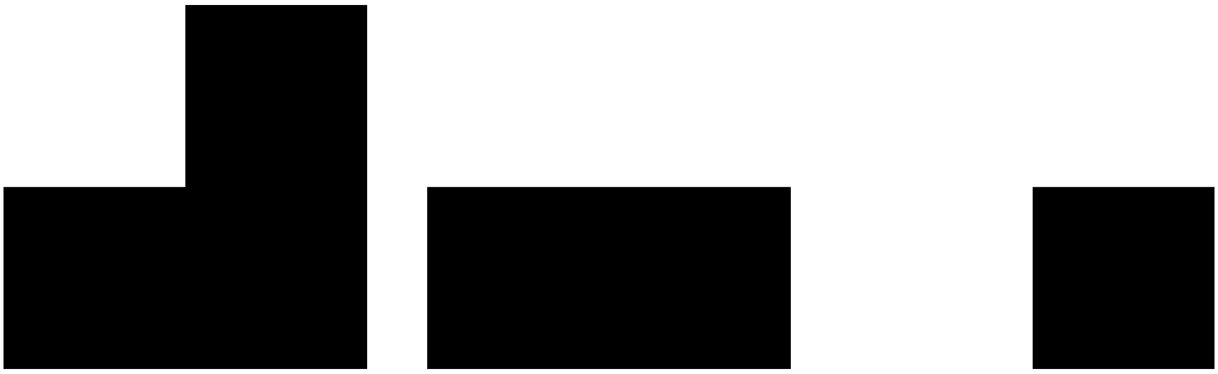


Figure 8.11: Predictions for both Moffatt and quarter for grid and recursive partitioning with 4 processes on 2 nodes.

We predict the time of each rank for a few different configurations. The result of this is shown in Figure 8.11. The x-axis show multiple runs. For each run, the predicted time per rank is shown as individual bars to the left of the marker. The four bars to the right of each marker indicate the actual time for each rank. Each bar consists of collision shown in blue, border exchange shown in green, and propagation shown in red.

One flaw of the simulation is that we only simulate a single iteration. The effect of this is that the predictions shown in Figure 8.11 are most accurate for the slowest rank. For other ranks, the synchronization time is underestimated. Simulating multiple iterations can reduce this error.

We see that while while we do not capture the communication time, we capture the synchronization cost and identify the overall time of an iteration. For the runs shown in Figure 8.11, the average prediction error was 20.08%



(a) Quarter fluid **(b)** Half fluid **(c)** Three quarter fluid

Figure 8.12: Three sample geometries illustrating imbalanced loads. Black is solid, white is void.

8.3 Performance optimizations

In this section we review the results above and their impact to the characteristic described in Chapter 6.

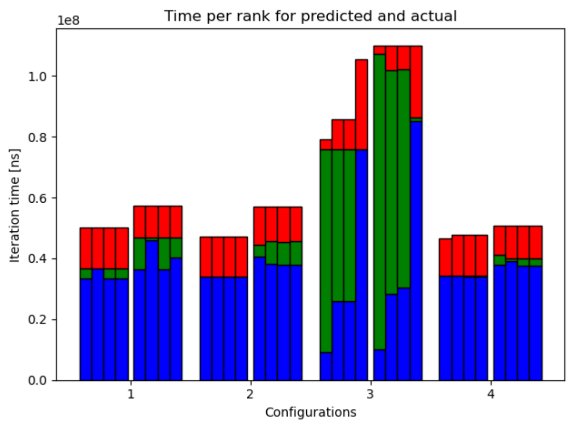
8.3.1 Porosity as proxy for balance

Using the model, we can identify a key observation. In Section 6.3.1 we identified the assumptions underlying different section costs and thus partitioning. From the model parameters found above, we find that $C_{fluid} > C_{solid}$. This is contrary to the traditional grid partitioning assumption, and the analytical assumption. To illustrate the consequences of this, we define three sample geometries shown in Figure 8.12. For each of the three geometries, we apply four different configurations.

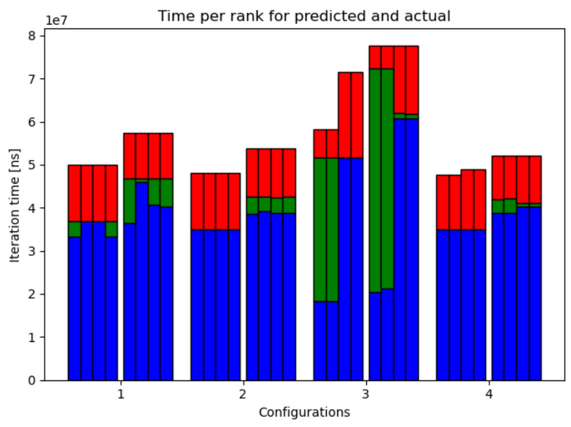
- Grid partitioning into 4 sections
- Grid partitioning into 16 sections
- Recursive balanced partitioning using the analytical parameters
- Recursive balanced partitioning using model parameters

All four configurations use round robin scheduling. Figure 8.13 shows the predicted time versus the actual time for each of these configurations and geometries.

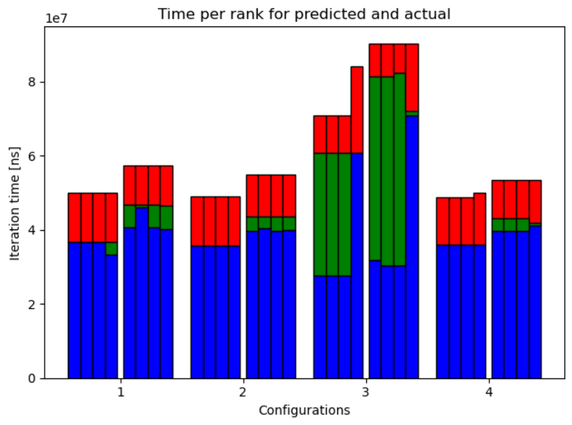
Using many sections, and distributing these across the ranks (second configuration) does improve the balance in this case, but at the expense of increased communication. The slight increase in propagation caused by the increased number of sections is also outweighed by the improved balancing. For all three configurations, we also find that using the analytical assumption (the third from the right and tallest set of bars) as a partitioning strategy creates highly unbalanced workloads and leads to the slowest iteration time. Using a balanced partitioning strategy that uses model weights however shows the best performance across all samples.



(a) Quarter fluid



(b) Half fluid



(c) Three quarter fluid

Figure 8.13: For each plot, predicted and actual time for the four configuration described. Input size is 4000x4000 and four ranks on two nodes.

Chapter 9

Conclusion

In this thesis, we have reviewed characteristics of porous media, and evaluated their potential application to improve performance of a lattice Boltzmann simulation. We state three potential assumptions on the relationship between the computational cost of fluid and solid points and determine their impact on performance though modeling of a proxy application.

We develop a performance model and simulator that predict the iteration with an error of 20.08% for complete runs. While we do not produce accurate communication time prediction, we capture the synchronization cost of the border exchange step of the application. We suspect this is the result of multiple factors, with mainly the overhead of MPI calls being underestimated. There are also potential effects of contention. For many configurations, the communication time is sufficiently small for this to have little impact.

We find that predicting performance of real applications is still a difficult task. While many models are proposed and shown to be effective on select applications, they are often not applicable to real world applications. Many models may only support one or a few concurrent transfers or support only blocking send and receive operations. In the case they support multiple concurrent, they may be targeted specifically at only small concurrent messages. Publicly available simulators are often trace-driven, which reduces the potential for exploring design through models rather than implementation.

The analytical model is able to reveal performance potential, though we can not realize this potential without relating the model to the application. The analytical model predict porosity to impact performance. However, partitioning based on the analytical model reduces performance. This is predicted by the application model and verified in runs of the application. Using the model parameters, which determine the relative difference of the cost of fluid and solid points to be 10%, we are able to extract the performance potential identified by the analytical model.

We find that while analysis of the underlying physics of the application can reveal potential for optimization, we cannot rely on analytical assumptions to provide speedup. However, using a model we can predict the impact of such changes, and realize the expected performance improvement.

Future work

In addition to the porosity, we identified other potential characteristics of porous media. While the proxy application in this work only considered solid and fluid points, we can also identify *internal points*. These points are the points that do not neighbor any fluid points, and thus do not need to be computed for either propagation nor collision. In future work, exploring such an optimization and how it increases performance given the number of grains and grain size is a potential avenue for further study.

The partitioning and scheduling techniques in this thesis can be extended to improve performance on heterogenous clusters. The balanced partitioning can be extended to create sections with a balance determined by the compute node resources as well as the model. The greedy scheduling can also be extended to schedule according to per node performance. Further, we can explore dynamic scheduling and redistribution of the sections during runtime.

Bibliography

[1] O. Aaziz, J. Cook, J. Cook, T. Juedeman, D. Richards, and C. Vaughan. A Methodology for Characterizing the Correspondence Between Real and Proxy Applications. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 190–200, 10. doi: 10.1109/CLUSTER.2018.00037.

[2] O. Aaziz, J. Cook, J. Cook, and C. Vaughan. Exploring and Quantifying How Communication Behaviors in Proxies Relate to Real Applications. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 12–22, 12. doi: 10.1109/PMBS.2018.8641569.

[3] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the LogP model - one step closer towards a realistic model for parallel computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 95–105, Santa Barbara, California, USA, July 1995. Association for Computing Machinery. ISBN 978-0-89791-717-9. doi: 10.1145/215399.215427.

[4] 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. *Computer*, 42(11):42–49, 2009. doi: 10.1109/MC.2009.372.

[5] P. L. Bhatnagar, E. P. Gross, and M. Krook. A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems. *Physical Review*, 94(3):511–525, May 1954. doi: 10.1103/PhysRev.94.511.

[6] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. Hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, February 2010. doi: 10.1109/PDP.2010.67.

[7] Tom Bultreys, Wesley De Boever, and Veerle Cnudde. Imaging and image-based fluid transport modeling at the pore scale in geological materials: A practical introduction to the current state-of-the-art. *Earth-Science Reviews*, 155:93–128, April 2016. ISSN 0012-8252. doi: 10.1016/j.earscirev.2016.02.001.

[8] Kirk Cameron, Rong Ge, and Xian-He Sun. lognP and log3P: Accurate Analytical Models of Point-to-Point Communication in Distributed Systems. *IEEE Transactions on Computers - TC*, 56:314–327, January 3. doi: 10.1109/TC.2007.38.

[9] Shiyi Chen and Gary D. Doolen. Lattice Boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, 30(1):329–364, 1998. doi: 10.1146/annurev.fluid.30.1.329.

[10] Wenguang Chen, Jidong Zhai, Jin Zhang, and Weimin Zheng. LogGPO: An accurate communication model for performance prediction of MPI programs. *Science in China Series F: Information Sciences*, 52:1785–1791, October 2009. doi: 10.1007/s11432-009-0161-2.

[11] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, 1993. ISSN 0362-1340. doi: 10.1145/173284.155333.

[12] S. S. Dosanjh, R. F. Barrett, D. W. Doerfler, S. D. Hammond, K. S. Hemmert, M. A. Heroux, P. T. Lin, K. T. Pedretti, A. F. Rodrigues, T. G. Trucano, and J. P. Luitjens. Exascale design space exploration and co-design. *Future Generation Computer Systems*, 30:46–58, January 2014. ISSN 0167-739X. doi: 10.1016/j.future.2013.04.018.

[13] C. Engelmann and F. Lauer. Facilitating co-design for extreme-scale systems through lightweight simulation. In *2010 IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*, pages 1–8, 20. ISBN null. doi: 10.1109/CLUSTERWKSP.2010.5613113.

[14] Charles R. Fitts. 2 - Physical Properties. In Charles R. Fitts, editor, *Groundwater Science (Second Edition)*, pages 23–45. Academic Press, Boston, January 2013. ISBN 978-0-12-384705-8. doi: 10.1016/B978-0-12-384705-8.00002-9.

[15] MPI Forum. MPI Documents. https://www.mpi-forum.org/docs/.

[16] A. Gonzalez, M. Valero-Garcia, and L. Diaz de Cerio. Executing algorithms with hypercube topology on torus multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):803–814, August 1995. ISSN 1558-2183. doi: 10.1109/71.406957.

[17] Boyun Guo. Chapter 2 - Petroleum reservoir properties. In Boyun Guo, editor, *Well Productivity Handbook (Second Edition)*, pages 17–51. Gulf Professional Publishing, January 2019. ISBN 978-0-12-818264-2. doi: 10.1016/B978-0-12-818264-2.00002-6.

[18] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, December 1998. ISSN 0167-8191. doi: 10.1016/S0167-8191(98)00093-3.

[19] Roger W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389–398, March 1994. ISSN 0167-8191. doi: 10.1016/S0167-8191(06)80021-9.

[20] Torsten Hoefler, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. LogfP - a model for small messages in InfiniBand. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, page 319, Rhodes Island, Greece, April 2006. IEEE Computer Society. ISBN 978-1-4244-0054-6.

[21] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. LogGOPSim: Simulating large-scale applications in the LogGOPS model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 597–604, Chicago, Illinois, June 2010. Association for Computing Machinery. ISBN 978-1-60558-942-8. doi: 10.1145/1851476.1851564.

[22] Fumihiko Ino, Noriyuki Fujimoto, and Kenichi Hagihara. LogGPS: A parallel computational model for synchronization analysis. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, volume 36, pages 133–142, June 2001. doi: 10.1145/379539.379592.

[23] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary Devito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles H. Still. Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 919–932, May 2013. doi: 10.1109/IPDPS.2013.115.

[24] Patrick MacArthur, Qian Liu, Robert D. Russell, Fabrice Mizero, Malathi Veeraraghavan, and John M. Dennis. An Integrated Tutorial on InfiniBand, Verbs, and MPI. *IEEE Communications Surveys Tutorials*, 19(4):2894–2926, Fourthquarter 2017. ISSN 1553-877X. doi: 10.1109/COMST.2017.2746083.

[25] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, Indianapolis, Indiana, USA, June 2010. Association for Computing Machinery. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807184.

[26] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.1, June 2015.

[27] Chul Moon and Matthew Andrew. Bentheimer networks. 2019. doi: 10.17612/1a36-rn45.

[28] Netpbm. The PBM Format. http://netpbm.sourceforge.net/doc/pbm.html, November 2013.

[29] OpenMP. OpenMP. https://www.openmp.org/, July 2020.

[30] Janusa Ragunathan and Jørgen Valstad. *Performance Modeling of CFD Application Scalability Using Co-Design Methods*. Master Thesis, Norwegian University of Science and Technology, 2018.

[31] Juan A. Rico-Gallego, Juan C. Díaz-Martín, Ravi Reddy Manumachu, and Alexey L. Lastovetsky. A Survey of Communication Performance Models for High-Performance Computing. *ACM Computing Surveys*, 51(6):126:1–126:36, January 2019. ISSN 0360-0300. doi: 10.1145/3284358.

[32] Jos B.T.M. Roerdink and Arnold Meijster. The Watershed Transform: Definition, Algorithms and Parallelization Strategies. *Fundamenta Informaticae*, 41(1-2):187, January 2000. ISSN 01692968. doi: 10.3233/fi-2000-411207.

[33] Gilbert Scott. North sea sandstone SEM images. 2020. doi: 10.17612/36F2-8Q45.

[34] Kamran Siddique, Zahid Akhtar, Edward J. Yoon, Young-Sik Jeong, Dipankar Dasgupta, and Yangwoo Kim. Apache Hama: An Emerging Bulk Synchronous Parallel Computing Framework for Big Data Applications. *IEEE Access*, 4:8879–8887, 2016. ISSN 2169-3536. doi: 10.1109/ACCESS.2016.2631549.

[35] Dmitriy Silin and Tad Patzek. Pore space morphology analysis using maximal inscribed spheres. *Physica A: Statistical Mechanics and its Applications*, 371(2):336–360, November 2006. ISSN 0378-4371. doi: 10.1016/j.physa.2006.04.048.

[36] Store norske leksikon. reservoar – petroleum. *Store norske leksikon*, May 2020.

[37] TOP500. TOP 500 June 2020. https://www.top500.org/lists/top500/2020/06/, June 2020.

[38] Unified Communication X. UCX. https://www.openucx.org/, July 2020.

[39] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8): 103–111, 1990. ISSN 0001-0782. doi: 10.1145/79173.79181.

[40] Mick van Duijn, Koen Visscher, and Paul Visscher. BSPLib: A fast, and easy to use C++ implementation of the Bulk Synchronous Parallel (BSP) threading model. 2016.

[41] David A. Wheeler. SLOCCount. https://dwheeler.com/sloccount/, August 2001.

[42] David P Williamson and David B Shmoys. *The Design of Approximation Algorithms*. GB: Cambridge University Press - M.U.A, GB, 2011. ISBN 0-521-19527-6. doi: 10.1017/CBO9780511921735.

[43] G. Zheng, Kakulapati Gunavardhan, and L. V. Kale. BigSim: A parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 78, April 2004. ISBN null. doi: 10.1109/IPDPS.2004.1303013.