



CZECH TECHNICAL UNIVERSITY IN PRAGUE  
Faculty of Nuclear Sciences and Physical Engineering



# **Implementation of components for distributed data management using the ADIOS2 library**

## **Implementace komponent pro distribuovanou správu dat pomocí knihovny ADIOS2**

Bachelor's Degree Project

Author: **Filip Borsodi**  
Supervisor: **Ing. Jakub Klinkovský, Ph.D.**  
Language advisor: **Darren Copeland, MSc.**  
Academic year: 2023/2024

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Borsodi** Jméno: **Filip** Osobní číslo: **509199**  
Fakulta/ústav: **Fakulta jaderná a fyzikálně inženýrská**  
Zadávací katedra/ústav: **Katedra matematiky**  
Studijní program: **Aplikovaná informatika**

## II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

**Implementace komponent pro distribuovanou správu dat pomocí knihovny ADIOS2**

Název bakalářské práce anglicky:

**Implementation of components for distributed data management using the ADIOS2 library**

Pokyny pro vypracování:

1. Seznamte se s projektem TNL-LBM (<https://gitlab.com/tnl-project/tnl-lbm>) a aktuální implementací pro ukládání dat ve formátu VTK.
2. Prozkoumejte knihovnu ADIOS2 (<https://adios2.readthedocs.io/en/latest/>) a její možnosti pro ukládání dat, zejména v distribuovaných výpočetních systémech.
3. Použijte knihovnu ADIOS2 pro správu dat v projektu TNL-LBM a porovnejte výhody oproti stávající implementaci.

Seznam doporučené literatury:

- [1] Dokumentace knihovny ADIOS2 <https://adios2.readthedocs.io/en/latest/index.html>
- [2] Dokumentace datových formátů VTK <https://examples.vtk.org/site/VTKFileFormats/>
- [3] Godoy, W. F., Podhorszki, N., Wang, R., Atkins, C., Eisenhauer, G., Gu, J., ... & Klasky, S. (2020). Adios 2: The adaptable input output system. a framework for high-performance data management. SoftwareX, 12, 100561. <https://doi.org/10.1016/j.softx.2020.100561>

Jméno a pracoviště vedoucí(ho) bakalářské práce:

**Ing. Jakub Klinkovský, Ph.D. katedra softwarového inženýrství FJFI**


Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **31.10.2023**

Termín odevzdání bakalářské práce: **05.08.2024**

Platnost zadání bakalářské práce: **30.09.2025**

  
Ing. Jakub Klinkovský, Ph.D.  
podpis vedoucí(ho) práce

  
prof. Ing. Zuzana Masáková, Ph.D.  
podpis vedoucí(ho) ústavu/katedry

  
doc. Ing. Václav Čuba, Ph.D.  
podpis děkana(ky)

## III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

28.11.2023

Datum převzetí zadání



Podpis studenta

*Acknowledgment:*

I would like to thank my supervisor Ing. Jakub Klinkovský, Ph.D. for his expert guidance and express my gratitude to Darren Copeland, MSc. for his language assistance.

*Author's declaration:*

I declare that this Bachelor's Degree Project is entirely my own work and I have listed all the used sources in the bibliography.

Prague, August 5, 2024

Filip Borsodi

*Název práce:*

**Implementace komponent pro distribuovanou správu dat pomocí knihovny ADIOS2**

*Autor:* Filip Borsodi

*Studijní program:* Aplikovaná Informatika

*Druh práce:* Bakalářská práce

*Vedoucí práce:* Ing. Jakub Klinkovský, Ph.D., České Vysoké Učení technické v Praze, Fakulta jaderná a fyzikálně inženýrská, Katedra softwarového inženýrství.

*Abstrakt:* Tato práce se zaměřuje na implementaci nového systému ukládání dat v rámci projektu TNL-LBM, s využitím knihovny ADIOS2. Hlavním cílem je představit nová vylepšení pro zvýšení efektivity a škálovatelnosti správy dat pro rozsáhlá vědecká simulační prostředí. Práce nejprve popisuje projekt TNL-LBM, a následně vysvětluje formát VTK, který se většinou používá pro vizualizaci. Dále představuje knihovnu ADIOS2 a zdůrazňuje její výhody pro vysoce výkonné datové vstupní/výstupní operace. Poté práce zkoumá starou a novou implementaci kódu a poskytuje srovnání jejich výkonu.

*Klíčová slova:* ADIOS2, distribuované výpočty, Paraview, správa dat, TNL-LBM

*Title:*

**Implementation of components for distributed data management using the ADIOS2 library**

*Author:* Filip Borsodi

*Abstract:* This thesis focuses on the implementation of a new data saving system within the TNL-LBM project, utilizing the ADIOS2 library. The main goal is to present new improvements for increasing the efficiency and scalability of data management for large-scale scientific simulation environments. First, the thesis describes the project TNL-LBM and explains afterwards the VTK format, which is mostly used for visualization. Next, it introduces the ADIOS2 library, highlighting its advantages for high-performance data I/O operations. The thesis then examines the old and new code implementations, providing a comparison of their performance.

*Key words:* ADIOS2, data management, distributed computing, Paraview, TNL-LBM

# Contents

<b>Introduction</b>	<b>6</b>
<b>1 TNL-LBM</b>	<b>7</b>
1.1 Design and Features . . . . .	7
1.2 Applications and Use Cases . . . . .	8
<b>2 VTK Format</b>	<b>9</b>
2.1 Structure and Features . . . . .	9
2.2 Examples of VTK Formats . . . . .	10
2.2.1 Example 1: ASCII VTK File for Structured Grid . . . . .	10
2.2.2 Example 2: Binary VTK File for Unstructured Grid . . . . .	10
2.2.3 Example 3: VTK File for PolyData . . . . .	11
2.3 Applications and Use Cases . . . . .	11
2.4 Disadvantages . . . . .	12
<b>3 ADIOS2 Library</b>	<b>13</b>
3.1 Core Features and Capabilities . . . . .	13
3.2 Integration with Scientific Workflows . . . . .	14
<b>4 Code structure</b>	<b>16</b>
4.1 Old implementation . . . . .	16
4.1.1 writeVTK Methods . . . . .	16
4.1.2 Code Examples . . . . .	17
4.2 New implementation . . . . .	20
4.2.1 MPI . . . . .	20
4.2.2 ADIOSWriter class . . . . .	20
4.2.3 writeADIOS methods . . . . .	23
4.3 Evaluation of the new implementation . . . . .	24
4.3.1 Code Quality . . . . .	24
4.3.2 Writing Performance . . . . .	25
<b>Conclusion</b>	<b>27</b>
<b>Bibliography</b>	<b>29</b>

# Introduction

Efficient data management is crucial for large-scale scientific simulations. Without proper data components, the simulation might not even be usable. The TNL-LBM project currently uses the VTK format to save its data. Compared to other solutions, current implementation of this library unfortunately comes with a lot of disadvantages, such as slower speed, hardly readable code or bigger file size. ADIOS2 (Adaptive IO System version 2) offers a versatile framework for high-performance data I/O, enabling seamless handling of vast datasets.

This thesis discusses the integration of the ADIOS2 library into TNL-LBM, highlighting the design, implementation, and performance benefits of this approach over the current one. The goals of this work are to speed up the data-saving process, reduce the file size, and clean up the code, making it more readable and extensible.

This thesis is organized as follows: In chapter 1, we discuss the TNL-LBM project and its design, features, and applications. In chapter 2, we cover VTK Format and its structure, features, applications and disadvantages. This chapter also provides a few examples. Chapter 3 focuses on the ADIOS2 Library, including its features and integration with Scientific Workflows. Chapter 4 presents old and new implementations of data-saving methods, as well as the comparison between them.

# Chapter 1

## TNL-LBM

The TNL-LBM (Template Numerical Library for Lattice Boltzmann Method) project is an advanced computational framework designed to facilitate large-scale simulations using the Lattice Boltzmann Method.[1][2][3] The LBM is a powerful computational technique for fluid dynamics simulations, which models fluid flow by tracking the evolution of particle distribution functions on a discrete lattice grid. This method is particularly effective for simulating complex fluid behaviors in various applications, including aerodynamics, porous media flow, and multiphase flows.

### 1.1 Design and Features

The TNL-LBM project emphasizes modularity, efficiency, and flexibility. It extends these principles to provide a robust platform tailored specifically for LBM simulations.

Key features include:

#### 1. Modular Architecture

The design of TNL-LBM is highly modular, allowing for easy integration and extension of different LBM models, boundary conditions, and collision operators. This modularity facilitates customization and experimentation with various LBM approaches without significant changes to the core codebase.[4]

#### 2. High Performance

The project leverages advanced programming techniques, including template metaprogramming and hardware-specific optimizations, to ensure efficient utilization of computational resources. This includes support for General-Purpose computing on Graphics Processing Units (GPGPU), which accelerates computations by offloading intensive tasks to GPUs. This focus on performance makes TNL-LBM suitable for running on high-performance computing (HPC) systems using Message Passing Interface (MPI).[5]

#### 3. Scalability

One of the primary objectives of TNL-LBM is to support large-scale simulations. It is designed to scale efficiently across multiple processors and compute nodes, enabling researchers to tackle problems that require significant computational power and memory.

## 4. Validation on Real-World Problems

Validating real-world problems is important for developing TNL-LBM. By applying it to diverse and complex fluid dynamics scenarios, TNL-LBM has demonstrated its robustness and accuracy. These validation efforts assure its dependability over numerous applications which encourages wider adoption in the fields of science and engineering.

### 1.2 Applications and Use Cases

TNL-LBM has been employed in numerous scientific and engineering applications, demonstrating its versatility and effectiveness. Some notable use cases include:

- **Environmental Science**

- Modeling pollutant dispersion in air and water, contributing to environmental impact assessments and mitigation strategies.[6]

- **Biomedical Engineering**

- Simulation of blood flow in arteries and veins, aiding in the study of cardiovascular diseases and the development of medical devices. [7]

- **Energy Sector**

- Analysis of air flow through a distributor plate in fluidized bed combustor. [8]

- **Aerospace Engineering**

- Simulation of airflow over aircraft wings and other aerodynamic surfaces to optimize design and performance.
- While this was not done specifically with TNL-LBM, LBM in general can be used for this problem.



## Chapter 2

# VTK Format

The Visualization Toolkit (VTK) format is an international standard for scientific data representation, supporting the realization of all types of analyses and visualizations. Initially being part of the library of VTK, developed for the effective treatment of complex data structures usually encountered in scientific computing. Such power and versatility make this format one of the most important tools available for researchers and practicing engineers today in areas such as fluid dynamics, computational physics, and medical imaging. [9] [10]

### 2.1 Structure and Features

The VTK format supports a variety of data types and structures, which are critical for accurately representing scientific data. This part is focused on VTK "legacy" file format.

Key features include:

#### 1. Data Types

VTK can handle scalar, vector, tensor, texture, and other types of data. This flexibility allows it to represent all kinds of different physical quantities and properties, enabling the widest spectrum of science applications.

#### 2. Grid Structures

VTK handles many different types of grids, including structured grids, unstructured grids, rectilinear grids, and polygonal data. It is highly desirable to have this capability so that complex geometries and spatial relationships can be modeled with a high degree of accuracy in a simulation.

#### 3. Data Representation

There are ASCII and binary representations available for VTK format. On one hand, ASCII files are human-readable and very useful during debugging. On the other hand, binary files have huge performance benefits since they are more compact and read/write operations are much faster.



**DATASET UNSTRUCTURED\_GRID** specifies that the dataset is an unstructured grid  
**POINTS** lists the coordinates of 8 points in binary format  
**CELLS** defines the connectivity of the points  
**CELL\_TYPES** specifies the type of cell used, with **12** representing a hexahedron

### 2.2.3 Example 3: VTK File for PolyData

```
1 # vtk DataFile Version 3.0
2 PolyData Example
3 ASCII
4 DATASET POLYDATA
5 POINTS 4 float
6 0.0 0.0 0.0
7 1.0 0.0 0.0
8 1.0 1.0 0.0
9 0.0 1.0 0.0
10 POLYGONS 1 5
11 4 0 1 2 3
```

**Listing 2.3:** VTK File for PolyData

**DATASET POLYDATA** indicates that the dataset is polydata  
**POINTS** lists the coordinates of 4 points  
**POLYGONS** defines a single polygon with 4 vertices, specified by their indices (0, 1, 2, 3) in the points list, with **4** indicating the number of vertices and **5** the number of entries in this section.

## 2.3 Applications and Use Cases

The VTK format is integral to numerous scientific and engineering workflows, providing several key benefits:

### 1. Interoperability

The VTK format is supported by a wide range of visualization and analysis tools, including ParaView[11], VisIt[12], and the VTK library itself. This interoperability allows researchers to seamlessly integrate VTK-based data into their workflows, facilitating comprehensive data analysis and visualization.

### 2. Visualization

High-quality visualization is very important for interpreting complex simulation results. The support provided in VTK for a wide range of data types and grid structures allows details and accuracy in visual representations, which will lead to an understanding of intricate phenomena.

### 3. Data Exchange

Because VTK format is standardized, it forms a very good basis for data exchange between different software tools and research groups. In this way, standardization makes it that data are consistently interpretable, reducing the possibility of errors during data transfer.

## 4. Performance

The binary format option in VTK enables large performance improvements by reducing file sizes and faster read/write operations. In reality, this performance boost is extremely important while dealing with the large datasets usually created by high-resolution simulations.

## 2.4 Disadvantages

Although the VTK format is pretty beneficial for a lot of reasons, such as flexibility in scientific visualization tasks and support of a large number of various data types, it has many drawbacks. These limitations can sometimes affect the effectiveness in certain applications and workflows.

### 1. Lack of Random Access

One of the major drawbacks of the VTK format is that it does not support random access to its data. This means that in many cases, parsing the whole file sequentially may be the only way to access parts from large datasets. This will slow down the time taken to retrieve the data, especially for huge data sets when only a small portion of data is required.

### 2. Absence of Parallel/Distributed Support

The VTK format does not natively provide any parallel or distributed computing environment. This can be a huge drawback for many modern scientific computing use cases when datasets are of a size that they no longer fit into the memory of a single machine. Additional layers of complexity are required for efficient data handling and processing in parallel or distributed systems.

### 3. Complex and Non-Robust Parsing

Parsing of VTK files is complex and often not very robust. Flexibility in the format and multiple, variously existing data types can easily introduce inconsistencies, leading to parsing errors. The structure may vary slightly from one VTK file to another, which makes the development of parsing tools that will ensure precise data extraction over various files rather difficult.

### 4. Lack of Formal Specification

Another major disadvantage is that no formal and complete specification documents are available for the VTK format. An implementation serves as a reference, but no formal specification is provided with the implementation. This may bring about ambiguities and variations in how different tools and implementations do interpret the VTK format, thus affecting compatibility and interoperability.

## Chapter 3

# ADIOS2 Library

The ADIOS2[13] library is a leading, high-performance data management framework developed to address the challenges of handling large-scale data volumes generated by scientific simulations. It is specifically designed to enable efficient, flexible operations on a scalable basis and allows any researcher to conduct effective input/output operations, thereby making it a very integral part in computational science.

Many high-performance computing applications and tools integrate ADIOS2. For example, Exascale Computing Project's application ExaAM [14] relies on ADIOS2 for productive data movement and storage, ensuring its scalable performance on large-scale simulations. Other examples include the molecular dynamics simulator LAMMPS[15] and the fusion simulation code XGC[16] which use ADIOS2 to realize their heavy-duty I/O tasks, thereby demonstrating its versatility in different scientific areas.

### 3.1 Core Features and Capabilities

ADIOS2 is built to provide for the diverse and demanding needs of scientific data management.

Its core features include:

#### 1. High Performance

ADIOS2 is tuned for high-performance data I/O operations. This kind of functionality is indispensable when dealing with the huge datasets arising from large-scale simulations. Advanced techniques are used in reducing I/O overhead and maximizing throughput, such as asynchronous I/O, collective buffering, data streaming, among others.

#### 2. Scalability

Designed with HPC environments in mind, ADIOS2 scales efficiently across thousands of nodes and processors. This scalability ensures that it can handle the increasing data volumes associated with modern scientific simulations.

### 3. **Flexibility**

ADIOS2 supports a wide range of data formats and storage backends, including HDF5, BP (Binary Packed), and SST (Staging Transport). This flexibility allows it to be integrated into various workflows and used in conjunction with different storage solutions, from local disks to parallel file systems.

### 4. **Data Model and APIs**

The library provides a rich data model supporting most complicated classic data types, such as multidimensional arrays and user-defined structures. It comes with easy-to-use APIs available in many programming languages: C++, Fortran, Python, and C.

### 5. **Fault Tolerance and Resilience**

ADIOS2 offers a variety of capabilities for redundancy and error detection to ensure the integrity and reliability of data in large-scale simulations.

## 3.2 **Integration with Scientific Workflows**

ADIOS2 is not just a standalone library but a critical component of various scientific workflows. It is designed to seamlessly integrate with simulation codes, data analysis tools, and visualization platforms. This integration capability is particularly important for multidisciplinary research projects that require coordinated data management across different stages of the computational pipeline.

The key benefits of integrating ADIOS2 into TNL-LBM include:

#### 1. **Efficient Data I/O**

The ADIOS2 environment will enable TNL-LBM to accomplish efficient data I/O, allowing saving of simulation time to be spent on tasks other than reading or writing data. Such high efficiency will make this software an indisputable need when simulating high-resolution datasets.

#### 2. **Scalable Data Management**

TNL-LBM making use of ADIOS2 would allow it to scale across several nodes for large-scale simulations and ensures that the management of data is handled in a manner that supports the analysis of problems in complex fluid dynamics.

#### 3. **Interoperability with VTK**

ADIOS2 supports the inclusion of VTK metadata commonly used for visualization. The capability thus provided enables TNL-LBM to write simulation results in a structure that can be later on mapped and visualized by the ParaView plugin for ADIOS2 data. This functionality, therefore, simplifies the post-processing workflow of the results from TNL-LBM-based research by being ready for analysis and visualization without any further need for data conversion.

#### 4. **Data Streaming and Staging**

ADIOS2's support for data streaming and in-situ data processing enables real-time analysis and visualization of simulation data. This feature is particularly valuable for monitoring long-running simulations and making timely adjustments based on intermediate results.

# Chapter 4

## Code structure

In the project, several methods are employed to periodically write simulation data, such as velocity fields and density distributions, to files using VTK metadata. These methods are invoked at regular intervals during the simulation to capture the evolving state of the system.

### 4.1 Old implementation

#### 4.1.1 writeVTK Methods

The project uses 5 main methods for saving different parts of the simulation once in a while. These are:

- **writeVTK\_3D**
  - Saving the entire simulation field
- **writeVTK\_3Dcut**
  - Saving 3D cut of the simulation, defined by cut offset and length
- **writeVTK\_2DcutX**
  - Saving 2D cut of the simulation on X axis, defined by X coordinate
- **writeVTK\_2DcutY**
  - Saving 2D cut of the simulation on Y axis, defined by Y coordinate
- **writeVTK\_2DcutZ**
  - Saving 2D cut of the simulation on Z axis, defined by Z coordinate



## 4.1.2 Code Examples

**Listing 4.1** depicts a code from `writeVTK_3D`, whose function is to first create the VTK-Writer class and open the file in append mode, followed by writing metadata describing the dataset, such as its dimensions, time, cycle and wall coordinates, into the file using `vtk` method `writeInt` or `writeFloat`.

```
1 VTKWriter vtk;
2
3 FILE* fp = fopen(filename.c_str(), "w+");
4 vtk.writeHeader(fp);
5 fprintf(fp, "DATASET RECTILINEAR_GRID\n");
6 fprintf(fp, "DIMENSIONS %d %d %d\n", (int)local.x(), (int)local.y(), (int)local.z());
7 fprintf(fp, "X_COORDINATES %d float\n", (int)local.x());
8 for (idx x = offset.x(); x < offset.x() + local.x(); x++)
9     vtk.writeFloat(fp, lat.lbm2physX(x));
10 vtk.writeBuffer(fp);
11
12 fprintf(fp, "Y_COORDINATES %d float\n", (int)local.y());
13 for (idx y = offset.y(); y < offset.y() + local.y(); y++)
14     vtk.writeFloat(fp, lat.lbm2physY(y));
15 vtk.writeBuffer(fp);
16
17 fprintf(fp, "Z_COORDINATES %d float\n", (int)local.z());
18 for (idx z = offset.z(); z < offset.z() + local.z(); z++)
19     vtk.writeFloat(fp, lat.lbm2physZ(z));
20 vtk.writeBuffer(fp);
21
22 fprintf(fp, "FIELD FieldData %d\n", 2);
23 fprintf(fp, "TIME %d %d float\n", 1, 1);
24 vtk.writeFloat(fp, time);
25 vtk.writeBuffer(fp);
26
27 fprintf(fp, "CYCLE %d %d float\n", 1, 1);
28 vtk.writeFloat(fp, cycle);
29 vtk.writeBuffer(fp);
30
31 fprintf(fp, "POINT_DATA %d\n", (int)(local.x()*local.y()*local.z()));
32
33 fprintf(fp, "SCALARS wall int 1\n");
34 fprintf(fp, "LOOKUP_TABLE default\n");
35 for (idx z = offset.z(); z < offset.z() + local.z(); z++)
36 for (idx y = offset.y(); y < offset.y() + local.y(); y++)
37 for (idx x = offset.x(); x < offset.x() + local.x(); x++)
38     vtk.writeInt(fp, hmap(x,y,z));
```

**Listing 4.1:** Metadata Writing

**Listing 4.1** continues with code from **Listing 4.2**, whose function is to write simulation data into the file. It is implemented using few nested cycles, which allow the method to go trough every dimension of either scalar or vector variable. The value of the variable is retrieved using the `outputData` function, and is written into the file using `vtk` method `writeFloat`. Every `writeVTK` method uses some variation of this code and code from **Listing 4.1**.

```

1 char idd[500];
2 real value;
3 int dofs;
4 int index=0;
5 while (outputData(*this, index++, 0, idd, offset.x(), offset.y(), offset.z(), value,
6     dofs)){
7     if (dofs==1){
8         fprintf(fp, "SCALARS %s float 1\n", idd);
9         fprintf(fp, "LOOKUP_TABLE default\n");
10    }
11    else
12        fprintf(fp, "VECTORS %s float\n", idd);
13
14    for (idx z = offset.z(); z < offset.z() + local.z(); z++)
15    for (idx y = offset.y(); y < offset.y() + local.y(); y++)
16    for (idx x = offset.x(); x < offset.x() + local.x(); x++){
17        for (int dof=0; dof<dofs; dof++){
18            outputData(*this, index-1, dof, idd, x, y, z, value, dofs);
19            vtk.writeFloat(fp, value);
20        }
21    }
22    vtk.writeBuffer(fp);
23 }
fclose(fp);

```

**Listing 4.2:** Data Writing

**Listing 4.3/4:** VTKWriter header declares metadata writing method `writeHeader` and data writing methods `writeInt` and `writeFloat`, the latter of which requires support methods `forceBigEndian` and `writeBuffer`. These data writing methods save single value at a time.

```

1 #ifndef __VTK_WRITER__
2 #define __VTK_WRITER__
3
4 #include "defs.h"
5
6 struct VTKWriter{
7     bool zip=false;
8     long buffer_len=64*1024*1024;
9     long buffer_pos=0;
10    float *buffer=0;
11
12    void forceBigEndian(unsigned char *bytes);
13    void writeHeader(FILE*fp);
14    void writeInt(FILE*fp, int val);
15    void writeFloat(FILE*fp, float val);
16    void writeBuffer(FILE*fp);
17
18    VTKWriter(){
19        buffer = (float*)calloc(buffer_len, sizeof(float));
20        buffer_pos=0;
21    }
22
23    ~VTKWriter(){
24        if (buffer) free(buffer);
25    }
26 };
27
28 #include "vtk_writer.hpp"
29 #endif

```

**Listing 4.3:** VTKWriter header

```

1 void VTKWriter::writeInt(FILE*fp, int val)
2 {
3     forceBigEndian((unsigned char *) &val);
4     fwrite(&val, sizeof(int), 1, fp);
5 }
6
7 void VTKWriter::writeFloat(FILE*fp, float val){
8     if (!fp) return;
9     forceBigEndian((unsigned char *) &val);
10    if (buffer_pos>=buffer_len){
11        printf("vtk.writeFloat::unexpected pos %ld vs. max %ld\n",buffer_pos,
12            buffer_len-1);
13        return;
14    }
15    buffer[buffer_pos] = val;
16    buffer_pos++;
17    if (buffer_pos == buffer_len){
18        writeBuffer(fp);
19    }
20 }

```

**Listing 4.4:** VTKWriter data writing methods

## 4.2 New implementation

In the new implementation, the methods responsible for saving data have been renamed from `writeVTK_...` to `writeADIOS_...`. They also use new `ADIOSWriter` class to save data, which uses the ADIOS2 library. From the subsection 3.2, the Efficient Data I/O, Scalable Data Management and Interoperability with VTK features were used. Overall shape of the dataset field does not change.

### 4.2.1 MPI

The Message Passing Interface (MPI) [17] used by the TNL-LBM project serves for distributed data and parallel computations. MPI enables partitioning of the computational domain among many processors, hence increasing the scalability and performance of applications.

### 4.2.2 ADIOSWriter class

The `ADIOSWriter` class exemplifies this distributed data model. It is initialized with an MPI communicator, allowing coordination among different processes. The constructor sets up the ADIOS2 I/O system using the provided communicator and configures the output file for either writing or appending based on the simulation cycle. The class handles global and local data extents, physical origins, and grid spacing, ensuring proper alignment and mapping of data across the distributed system.

```
1 ADIOSWriter(TNL::MPI::Comm communicator, std::string file_name, CoordinatesType
  global, CoordinatesType local, CoordinatesType offset, PointType physOrigin,
  real physDl, int cycle) : adios(communicator){
2     bpIO = adios.DeclareIO("bpIO");
3     bpIO.SetEngine("BP4");
4     fileName = file_name + ".bp";
5     if(cycle==0){
6         bpWriter = bpIO.Open(fileName, adios2::Mode::Write);
7     }else{
8         bpWriter = bpIO.Open(fileName, adios2::Mode::Append);
9     }
10    this->global = global;
11    this->local = local;
12    this->offset = offset;
13    this->physOrigin = physOrigin;
14    this->physDl = physDl;
15    bpWriter.BeginStep();
16 }
```

**Listing 4.5:** ADIOSWriter Constructor

**Listing 4.6:** The destructor finalizes the ADIOS2 writing step and constructs XML metadata for VTK visualization.

```

1 ~ADIOSWriter(){
2     const std::string extentG = "0 " + std::to_string(global.z()) + " 0 " + std::
3     to_string(global.y()) + " 0 " + std::to_string(global.x());
4     const std::string extentL = "0 " + std::to_string(local.z()) + " 0 " + std::
5     to_string(local.y()) + " 0 " + std::to_string(local.x());
6     const std::string origin = std::to_string(physOrigin.x()) + " " + std::to_string
7     (physOrigin.y()) + " " + std::to_string(physOrigin.z());
8     const std::string spacing = std::to_string(physDl) + " " + std::to_string(physDl
9     ) + " " + std::to_string(physDl);
10
11     const std::string imageData = R"(
12     <?xml version="1.0"?>
13     <VTKFile type="ImageData" version="0.1" byte_order="LittleEndian">
14     <ImageData WholeExtent=") + extentG + R("( Origin=") + origin + R("(
15     Spacing=") + spacing + R("(
16     <Piece Extent=") + extentL + R("(
17     <CellData Scalars="data">)
18     + DataArrays + R("(
19     </CellData>
20     </Piece>
21     </ImageData>
22     </VTKFile>";
23
24     bpIO.DefineAttribute<std::string>("vtk.xml", imageData);
25     bpWriter.EndStep();
26     varNames.clear();
27     bpWriter.Close();
28 }

```

**Listing 4.6:** ADIOSWriter Destructor

**Listing 4.7:** The recordVarName method saves the variable names into a vector varNames. Depending on the dimension (dim) of the variable, it constructs the appropriate XML <DataArray> element. Lastly, it saves it into a vector DataArrays, which is used in the destructor.

```

1 template< int D_, typename real, typename idx >
2 void ADIOSWriter<D_, real, idx>::recordVarName(std::string varName, int dim){
3     varNames.push_back(varName);
4     switch(dim){
5     case 0:
6         DataArrays += "<DataArray Name=\"" + varName + "\"> " + varName + " </
7         DataArray>\n";
8         break;
9     case 1:
10    case 3:
11        DataArrays += "<DataArray Name=\"" + varName + "\"/>\n";
12        break;
13    default:
14        throw std::invalid_argument("Invalid dimension of \"" + varName + "\"("
15        + std::to_string(dim) + ").");
16    }
17 }

```

**Listing 4.7:** Method for saving variable names

**Listing 4.8/9:** The write methods within ADIOSWriter facilitate both scalar and vector data writes, managing the data distribution across the specified global, local, and offset dimensions, while making sure no variable can be saved twice under the same name. Additionally, they save the variable name using recordVarName.

```

1  template<typename T>
2  void ADIOSWriter<D_, real, idx>::write(std::string varName, T val){
3      if(std::find(varNames.begin(), varNames.end(), varName) == varNames.end()){
4          adios2::Variable<T> value = bpIO.DefineVariable<T>(varName);
5
6          bpWriter.Put(value, val);
7          bpWriter.PerformPuts();
8
9          recordVarName(varName, 0);
10     }else{
11         throw std::invalid_argument("Variable \"" + varName + "\" is already defined
12         .");
13     }
14 }

```

**Listing 4.8:** Method for writing scalar data

```

1  template<typename T>
2  void ADIOSWriter<D_, real, idx>::write(std::string varName, std::vector<T>& val, int
3  dim){
4      if(std::find(varNames.begin(), varNames.end(), varName) == varNames.end()){
5          adios2::Dims shape({size_t(global.z()), size_t(global.y()), size_t(global.x
6          ())});
7          adios2::Dims start({size_t(offset.z()), size_t(offset.y()), size_t(offset.x
8          ())});
9          adios2::Dims count({size_t(local.z()), size_t(local.y()), size_t(local.x()
10         )});
11         adios2::Variable<T> values = bpIO.DefineVariable<T>(varName, shape, start,
12         count);
13
14         bpWriter.Put(values, val.data());
15         bpWriter.PerformPuts();
16
17         recordVarName(varName, dim);
18     }else{
19         throw std::invalid_argument("Variable \"" + varName + "\" is already defined
20         .");
21     }
22 }

```

**Listing 4.9:** Method for writing vector data

### 4.2.3 writeADIOS methods

**Listing 4.10:** At the beginning of the method, 2 vectors and a ADIOSWriter class are initialized, followed by saving of the walls of the simulation.

```
1 std::vector<int> tempIData;  
2 std::vector<float> tempFData;  
3 ADIOSWriter<3, float, int> adios(MPI_COMM_WORLD, filename.c_str(), global, local,  
4   offset, lat.physOrigin, lat.physDl, cycle);  
5 for (idx z = offset.z(); z < offset.z() + local.z(); z++)  
6 for (idx y = offset.y(); y < offset.y() + local.y(); y++)  
7 for (idx x = offset.x(); x < offset.x() + local.x(); x++)  
8   tempIData.push_back(hmap(x,y,z));  
9 adios.write<int>("wall", tempIData, 1);  
10 tempIData.clear();
```

**Listing 4.10:** Initialization in writeADIOS\_3D

**Listing 4.11:** During the data saving, the method first stores data into the temporary vector. Due to an issue[18] with saving vector variable as single variable, each dimension of vector variable is saved as separate variable with corresponding suffix.

```
1 while (outputData(*this, index++, 0, idd, offset.x(), offset.y(), offset.z(), value,  
2   dofs)){  
3   std::string IDD(idd);  
4   for (int dof=0;dof<dofs;dof++){  
5     for (idx z = offset.z(); z < offset.z() + local.z(); z++)  
6     for (idx y = offset.y(); y < offset.y() + local.y(); y++)  
7     for (idx x = offset.x(); x < offset.x() + local.x(); x++){  
8       outputData(*this, index-1, dof, idd, x, y, z, value, dofs);  
9       tempFData.push_back(value);  
10    }  
11    switch(dof){  
12      case 0:  
13        if(dofs>1){  
14          adios.write<float>(IDD + "X", tempFData, dofs);}  
15        else{  
16          adios.write<float>(IDD, tempFData, dofs);  
17        }  
18        break;  
19      case 1:  
20        adios.write<float>(IDD + "Y", tempFData, dofs);  
21        break;  
22      case 2:  
23        adios.write<float>(IDD + "Z", tempFData, dofs);  
24        break;  
25    }  
26    tempFData.clear();  
27  }  
28  
29 adios.write<float>("TIME", time);  
30 adios.write<int>("CYCLE", cycle);
```

**Listing 4.11:** Data writing in writeADIOS\_3D

## 4.3 Evaluation of the new implementation

The new implementation of the data management system within TNL-LBM has undergone several critical enhancements aimed at improving readability, efficiency, and writing performance. This section evaluates the key improvements made in the recent update.

### 4.3.1 Code Quality

- **Refactoring**

- Unused code from the prior implementation was removed, which helped reduce the overall size of the code. Not only has this reduction in redundant code made the implementation more readable, but also easier to maintain and debug. The writing methods have also been overloaded to allow for easier usage.

- **Optimization**

- The method for saving data has been hugely optimized. While in the old implementation, data is saved one value at a time, in the new implementation, it saves a whole vector of values. This change drastically reduces the overhead associated with frequent I/O operations, improving the performance of the data-saving process.

- **Self-Verifying**

- A new method to save variable names, `recordVarNames`, has been introduced, which, among helping to create metadata, can also be used to check if a variable is being saved more than once. This feature is most important in complex simulations where several variables are tracked and saved.

- **Consolidated Data Storage**

- With the new implementation, data are saved into a single file, significantly improving the visualization process. Previously, storing data in multiple files required the program to link those files together upon visualization, which many times introduced undesirable lines between sections. By consolidating all data into one file, this new approach eliminates the possibility of such visual artifacts, leading to cleaner and more accurate representations of the simulation results.



### 4.3.2 Writing Performance

The performance evaluation of the old and new implementation for saving the whole simulation field was conducted using two different configurations: one with a single process and another with four processes, both at resolution 4. The TNL class Timer was used to measure the time.

```
1 TNL::Timer timer;  
2 timer.start();  
3 block.writeVTK_3D(nse.lat, outputData, fname, nse.physTime(), cnt[VTK3D].count);  
4 timer.stop();  
5 std::cout << "write3D saved in: " << timer.getRealTime() << std::endl;  
6 timer.reset();
```

**Listing 4.12:** Code for measuring the write time for `writeVTK_3D` and `writeADIOS_3D`

#### • Parameters of the Simulation

All of the following simulations and measurements were done on a school computer gp1[19] with parameters:

- CPU: 2× Intel Xeon E5-2630 v3(8 cores @ 2.4-3.2 GHz, 20 MiB cache)
- Disk: 1 TB WD Caviar Black

The size of the lattice for both simulations was  $(X,Y,Z) = (512,128,128)$ . The fields stored for visualization were `lbm_density` (scalar variable) and `velocity`. (vector variable)

#### • Single Process Performance

For the single process configuration, the time taken to save the simulation field over the first five steps is presented in Table 4.1. The results show a significant improvement in the new implementation compared to the old one.

Both implementations resulted in the same file size of 827 MB, indicating that the new implementation achieves better performance without increasing the storage requirement.

Step #	Old	New
0	3.8070	2.8500
1	3.7395	2.6761
2	3.7346	2.6858
3	3.7485	2.6696
4	3.7432	2.6786

Table 4.1: Time (in seconds) to save the data using a single process, during first 5 steps, for old and new implementation

- **Multi-Process Performance**

For the configuration with four processes, the time taken to save the simulation field over the first five steps is shown in Table 4.2. Each process's time is recorded, highlighting the parallel performance improvements.

The file size for the old implementation was 822 MB, while the new implementation resulted in a slightly larger file size of 828 MB. This small increase in file size is more than compensated for by major efficiency gains in performance.

Step #	Old Implementation				New Implementation			
	Proc 0	Proc 1	Proc 2	Proc 3	Proc 0	Proc 1	Proc 2	Proc 3
0	0.7433	0.7544	0.8049	0.8056	0.5505	0.5448	0.5449	0.5499
1	0.8037	0.8042	0.8114	0.8131	0.5372	0.5372	0.5372	0.5375
2	0.7705	0.7990	0.8095	0.8097	0.5441	0.5443	0.5386	0.5445
3	0.8011	0.8016	0.8116	0.8121	0.5341	0.5342	0.5342	0.5344
4	0.7820	0.7840	0.8087	0.8095	0.5347	0.5350	0.5352	0.5352

Table 4.2: Time (in seconds) to save the data using four processes, during first 5 steps, for old and new implementation and for each process

# Conclusion

The main goal of this bachelor thesis was to implement a new data-saving system for the TNL-LBM project using the ADIOS2 library and compare it to the current implementation. This was achieved by first studying the project's old data management system, refactoring its code, and replacing it with new methods utilizing ADIOS2.

The first chapter discusses the design and features of the TNL-LBM project. It mentions topics such as Modular Architecture, High Performance or Validation on Real-World Problems. The chapter continues with applications of the project in practice, including Environmental Science, Biomedical Engineering, or Energy Sector.

The second chapter was devoted to the VTK legacy format, currently used by TNL-LBM. Following a brief introduction, it continues with the structure and features of this format, such as Grid Structures, Data Representation and Meta-Information. A few examples and applications of this format are then listed. The chapter finishes with a list of disadvantages.

The third chapter focuses on the ADIOS2 library. After explaining its function, it mentions core features and capabilities, including High Performance, Flexibility and Fault Tolerance and Resilience. The chapter then lists the benefits of integration of ADIOS2 into TNL-LBM, covering Efficient Data I/O, Scalable Data Management, or Interoperability with VTK.

The last chapter presents both old and new code structures while providing few examples. It then evaluates the new implementation, providing a number of improvements. The chapter ends by providing measured data on writing speed, coming to the conclusion that in both single and multiprocess simulations, the new implementation has significantly improved in writing speed while sacrificing little to no file size.

For future work, further improvements, such as solving the vector variable saving issue, can be implemented. Additionally, further data collection of reading speeds could be conducted.

# Bibliography

- [1] P. Eichler, R. Fučík, and R. Straka: *Computational study of immersed boundary-lattice Boltzmann method for fluid-structure interaction*, Discrete & Continuous Dynamical Systems-S 14.3 (2021), page 819. Science, 3(3):159-167, 2000.
- [2] R. Fučík, P. Eichler, R. Straka, P. Pauš, J. Klinkovský, and T. Oberhuber: *On optimal node spacing for immersed boundary–lattice Boltzmann method in 2D and 3D*, Computers & Mathematics with Applications 77.4 (2019), pages 1144–1162.
- [3] P. Eichler, V. Fuka, and R. Fučík: *Cumulant lattice Boltzmann simulations of turbulent flow above rough surfaces*, Computers & Mathematics with Applications 92 (2021), pages 37–47.
- [4] J. Klinkovský: *Data Structures and Parallel Algorithms for Numerical Solvers in Computational Fluid Dynamics*, Doctoral Thesis (2023).
- [5] T. Oberhuber, J. Klinkovský, R. Fučík: *TNL: Numerical library for modern parallel architectures*, Acta Polytechnica 61.SI (2021), 122-134.
- [6] J. Klinkovský, R. Fučík, A. C. Trautz, T. H. Illangasekare: *Lattice Boltzmann method–based efficient GPU simulator for vapor transport in the boundary layer over a moist soil: Development and experimental validation*, Computers & Mathematics with Applications 138 (2023), pages 65–87.
- [7] R. Fučík, R. Galabov, P. Pauš, P. Eichler, J. Klinkovský, R. Straka, J. Tintěra, and R. Chabiniok: *Investigation of phase-contrast magnetic resonance imaging underestimation of turbulent flow through the aortic valve phantom: experimental and computational study using lattice Boltzmann method*, Magnetic Resonance Materials in Physics, Biology and Medicine 33.5 (2020), pages 649–662.
- [8] M. Beneš, P. Eichler, R. Fučík, J. Hrdlička, J. Klinkovský, M. Kolář, T. Smejkal, P. Skopec, J. Solovský, P. Strachota, R. Straka, and A. Žák: *Experimental and numerical investigation of air flow through the distributor plate in a laboratory-scale model of a bubbling fluidized bed boiler*, Japan Journal of Industrial and Applied Mathematics 39.3 (2022), pages 943–958.
- [9] "VTK File Formats." VTK, [https://docs.vtk.org/en/latest/design\\_documents/VTKFileFormats.html](https://docs.vtk.org/en/latest/design_documents/VTKFileFormats.html). Accessed 22.7.2024
- [10] "VTK Documentation." VTK, <https://docs.vtk.org/en/latest/index.html>. Accessed 22.7.2024

- [11] "Paraview" Paraview, <https://www.paraview.org/>. Accessed 22.7.2024
- [12] "VisIt" Github, <https://visit-dav.github.io/visit-website/index.html>. Accessed 22.7.2024
- [13] "ADIOS2 Documentation" Adios, <https://adios2.readthedocs.io/en/v2.10.1/>. Accessed 22.7.2024
- [14] "ExaAM" ExascaleProject, <https://www.exascaleproject.org/research-project/exaam/>. Accessed 4.8.2024
- [15] "LAMMPS Molecular Dynamics Simulator" Lammps, <https://www.lammps.org/#gsc.tab=0>. Accessed 4.8.2024
- [16] "XGC1" High Fidelity Boundary Plasma Simulation, <https://epsi.pppl.gov/computing/xgc-1>. Accessed 4.8.2024
- [17] "MPI Forum" MPI Forum, <https://www.mpi-forum.org/>. Accessed 4.8.2024
- [18] "How to visualize vector fields on ImageData with Paraview?" Github, <https://github.com/ornladios/ADIOS2/discussions/4117>. Accessed 5.8.2024
- [19] "Hardware overview" Github, <https://mmg-gitlab.fjfi.cvut.cz/gitlab/mmg/gpx-cluster/-/blob/master/doc/hardware-overview.md>. Accessed 5.8.2024