BERGISCHE
UNIVERSITÄT
WUPPERTAL

Faculty of Mathematics and Natural Sciences

M A S T E R T H E S I S

Multi GPU usage in ARTSS

A thesis submitted for the degree of

Master of Science (M.Sc.)

in the course of studies

Computer Simulation in Science

Student: Max Joseph Böhler, B. Eng.

First examiner: Prof. rer. nat. Lukas Arnold
Second examiner: Assist. Prof. Sunita Chandrasekaran

Wuppertal, January 2020

Statement in Lieu of an Oath

I hereby declare that I have written this master thesis independently, without the help of third parties and without using sources and tools other than those indicated. All sources used, whether taken literally or by analogy, are individually marked as such. Furthermore, I declare that all images and graphics appearing in this thesis were created by me personally. This thesis has not yet been submitted to any other examination board and has not been published.

Wuppertal, 04th January 2020

[illegible]

Max Joseph Böhler

Abstract

ARTSS is a CFD code written in C++ that simulates buoyancy-driven turbulent smoke propagation based on finite differences and a large-eddy simulation turbulence model, using a GPU for accelerated computation. In the context of this work, the *Message Passing Interface* is implemented in *ARTSS*. This allows *ARTSS* to use multiple GPUs simultaneously. In addition to faster computation, this also permits more grid cells to be used in a simulation, since there are theoretically no longer any memory constraints. In order to avoid changing the structure of the existing code, a separate class `MPIHandler` is implemented. A decentralized approach is chosen as the parallelization strategy. This means that each process reads the configuration file and performs a domain decomposition for itself. Communication is non-blocking to avoid deadlocks.

For the analysis of the MPI implementation a simple tunnel setup is used which has a constant velocity profile in positive x-direction. A maximum of 16 GPUs or CPUs are used for the runtime analysis. The evaluation of these measurements shows that the MPI implementation provides the desired speedup.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Science is based on observation, theory and physical experimentation. An observation leads to a hypothesis, which in turn leads to a theory, which is proven or disproved by an experiment. Depending on the theory, those experiments may be too expensive, time-consuming or impossible to perform. In modern science, therefore, numerical simulations that embody the proposed theory are carried out and then compared with the observation of the real world. If observation and simulation are in agreement, the theory is considered proven.
Some numerical simulations are so complex that they are listed in the so-called *Grand Challenges*, a set of complex scientific problems that require extremely powerful computers to approximate the solution [Levin, 1989]. To utilize the computing resources of the so-called supercomputers, special application programming interfaces must be implemented in simulation software. This thesis focuses on the implementation and analysis of the *Message Passing Interface* in an existing numerical solver for smoke spread simulations. In principle, the more computing resources can be used and the higher the performance of a single computing resource, the faster a simulation can be calculated.

A way to quantify the performance capability of a computer is to measure the number of floating-point arithmetic calculations a systems can perform on a per-second basis. The measurement, floating-point operations per second, is abbreviated as *FLOPS*. The *FLOPS* performance is determined by two factors. On the one hand by increasing the clock frequency of a single processing unit and on the other hand by the parallel execution of tasks on different processing units. The increase in clock frequency is mainly achieved by reducing the size of the components of the processing units (i.e. the size of a transistor), thus reducing the capacitance and natural time constants [Sterling et al., 2018]. For the execution of parallel tasks, a distinction must be made between the microscopic and macroscopic view.

The microscopic view is limited to the parallelization within a processor with several processing units sharing the same memory. Within this view two approaches are pursued [Hwu et al., 2008]. One is the *multicore* approach, which minimize the execution latency of a single thread. This is done by using large cache memory, low-latency arithmetic units, and sophisticated operand delivery logic. However, this low-latency deign consumes chip area and power that could be otherwise used to provide more arithmetic execution units [Kirk and Hwu, 2016]. A current example is the AMD Ryzen 3900x CPU with 12 cores, which delivers 132.54 *gigaFLOPS* in double precision [University of Washington, 2020]. The second approach is the *many-thread* strategy, which optimize for the execution throughput of massive numbers of threads while allowing individual threads to have a potentially much longer execution latency. This throughput-oriented design saves chip area and power which also allows an increased number of arithmetic units [Kirk and Hwu, 2016]. A current example for throughput-oriented design is the NVIDIA Tesla P100 graphics processing unit (GPU) with 3584 cores, which delivers 4700 *gigaFLOPS* in double precision [Nvidia, 2020*a*].

Besides the microscopic view, the macroscopic view looks at many independent processors, each of which uses its own memory. This distributed memory approach is used especially for supercomputers and allows the use of any number of processors. The de facto standard for communication between processors is the so called *Message Passing Interface*. The combination of macroscopic and microscopic view, also called hybrid parallelization, allows supercomputers to achieve computational performance in the *petaFLOPS* range. In the context of this work, the JURECA system of the Forschungszentrum Jülich is used, which delivers 3.54 (CPU) + 14.98 (GPU) *petaFLOPS* in peak performance [Forschungszentrum Jülich GmbH, 2020].

1.1 ARTSS

The implementation of the *Message Passing Interface* is done in *ARTSS*, which is short for Accelerator-based Real Time Smoke Simulator. It is a CFD code base, written in *C++*, simulating buoyancy-driven turbulent smoke spread based on finite differences and a large eddy simulation turbulence model [Küsters, 2018]. It is based on *JuROr*, which was originally developed within the ORPHEUS project by Dr. Anne Küsters. In addition, ARTSS has the option of Dynamic Domain Adaptation, which allows the calculation of a partial section of the entire geometry. If the remaining geometry sections are required later in the simulation process, they are automatically loaded [Würzburger, 2019].

Unlike other CFD solvers that specialize in smoke propagation, such as the *Fire Dynamics Simulator* (*FDS*) or *fireFoam*, ARTSS has the ability to calculate simulation results in real time, while maintaining high accuracy of simulation results. Figure 1 shows the comparison between the calculation time for a simulation with *FDS*, using a Intel Xeon Haswell E5-2680 v3 @ 2.2 GHz CPU and *ARTSS*, using a NVIDIA Pascal P100 (PCIe, 12 GB) GPU. A simple simulation setup with three different grid resolutions is investigated [Küsters, 2018]. A real-time simulation is achieved as soon as the solution is calculated faster than the simulation time, here 500 seconds.

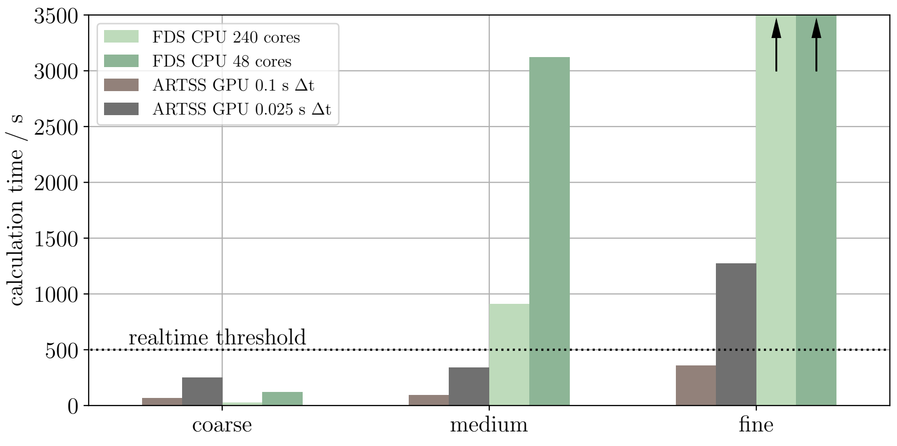


Figure 1: *Comparison between Fire Dynamics Simulator and ARTSS (GPU Version)*

It can be seen that *ARTSS* as well as *FDS* can calculate the simulation results in real time for small problem sizes (coarse). In order to calculate the simulation results with a higher accuracy, the grid resolution must be refined (medium and fine). In doing so, it can be seen that unlike *ARTSS*, *FDS* is no longer able to calculate the results in real time. To achieve this, *ARTSS* uses a GPU in addition to the CPU, which is significantly more performant in parallel tasks due to the throughput-oriented design. In the area of the smallest problem size, it can clearly be seen that *FDS* computes the simulation results faster than *ARTSS*. This is due to the fact that the data exchange between CPU and GPU in *ARTSS*, which does not occur in *FDS*, has a significant influence on the calculation time. However, this effect is relativized as the problem size increases, since the computation time on the GPU clearly exceeds the time for data exchange.

The GPU usage in ARTSS is implemented by the *OpenACC* Application Programming Interface. *OpenACC* is a collection of compiler directives and runtime routines that allows to specify loops and code areas in standard C, C++ and Fortran [Nvidia, 2020*b*].

1.2 Motivation

At the time before this work, *ARTSS* is limited to the locally available hardware. That is, *ARTSS* can either use a CPU core and a GPU simultaneously or multiple CPU cores sharing the same memory. This leads to two fundamental problems. First, only the computing power of one GPU can be accessed, and second, only the memory of one GPU can be used. In particular, the last problem leads to serious limitation, since the size of the simulation is restricted to the size of a GPU memory unit. The goal of this work is therefore the implementation of the *Message Passing Interface* within *ARTSS* to be able to use multiple GPUs and CPUs simultaneously. This leads to a reduction of the computation time as well as to the removal of the memory limitation and thus the possibility of the computation of an arbitrary problem size.

Chapter 2

Implementation

The implementation of the *Message Passing Interface* in *ARTSS* is done under consideration of two goals. On the one hand, the code base must not be changed in its structure and, on the other hand, it must be ensured that *ARTSS* also functions without MPI, thus serially. The latter is realized by the use of conditional inclusion statements, which are activated or deactivated when compiling the code.

The structure of *ARTSS* can be divided into four segments [Küsters, 2018]. Figure 2 shows the relationship between these segments in the form of a schematic class diagram. The first segment (1. Time Integration) serves as an introduction to the simulation process. Here, among other things, the solver and the required fields, e.g. temperature, pressure and speed are initialized. The second segment (2. Problem and solution method definition) is dedicated to the exact definition of the solver. Here, the module-based approach of *ARTSS* is elucidated. The pure virtual class `ISolver` is realised by a descriptive class. Depending on the problem at hand, an existing combined problem solver can be used here. These combined problem solvers consist of individual solvers which are described in the third segment (3. Numerical Methods). Each of these individual solvers is again represented by a virtual class. This allows the use of different numerical methods for one individual problem. For example, for the calculation of diffusion, three different numerical methods (*Jacobi*, *Explicit*, *colored Gauss-Seidel*) are available, which can be freely selected by the user at the beginning of a simulation. The fourth segment (4. Auxiliaries) contains all the elements of *ARTSS* that are needed to complement the other three segments. This includes, among other things, the visualization, the description of the domain and the boundary conditions. The utility group contains functions for reading in XML configuration files and other auxiliary functions.

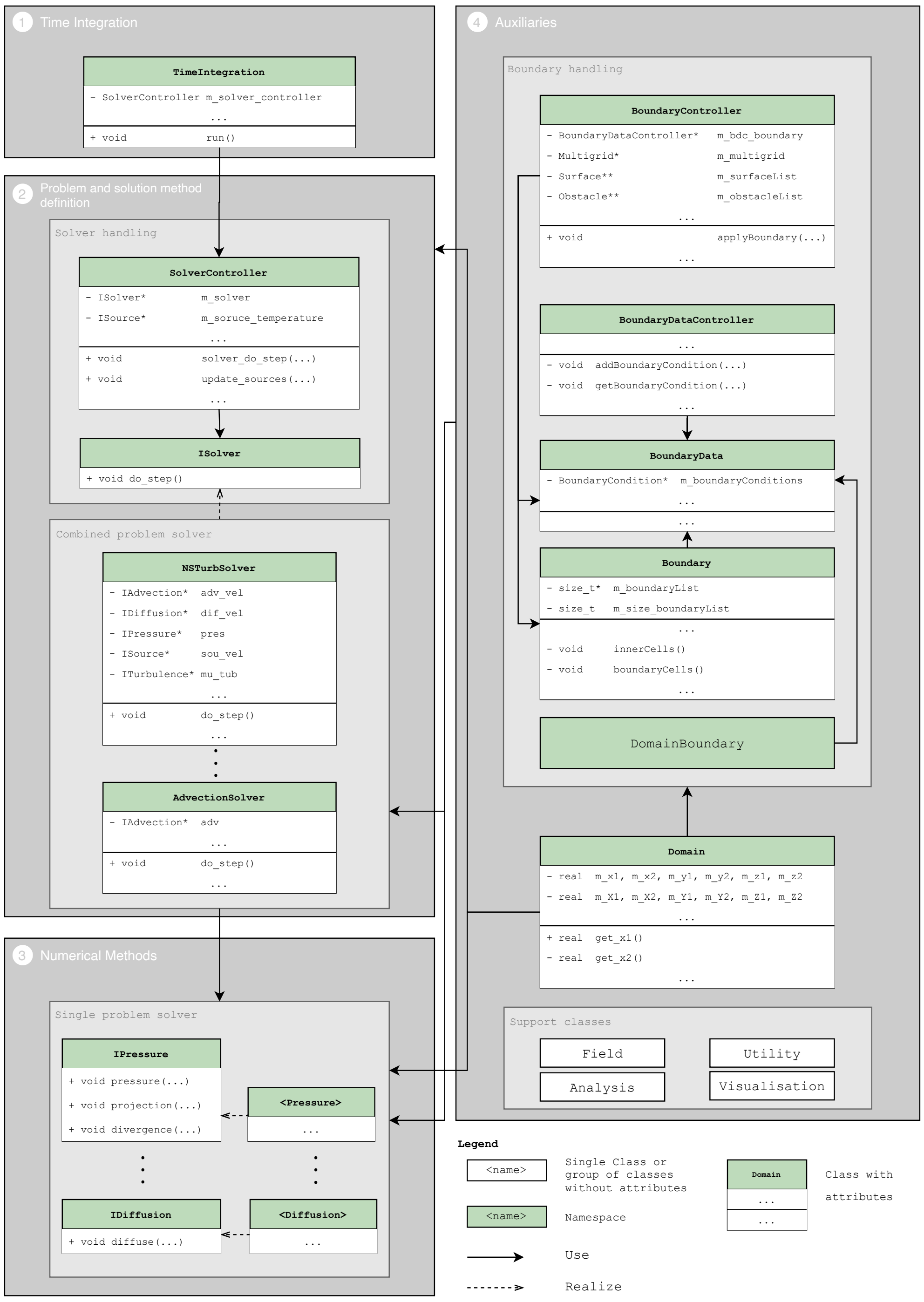


Figure 2: *Class diagram of ARTSS*

2.1 Message Passing Interface

The *Message Passing Interface* is a *message-passing library interface specification* which is developed and maintained by the MPI Forum, a public body with representatives from various organisations. The first version was released in 1994, while the most recent version 3.1, at the time of this work, was released in 2015. Since MPI is a library interface specification, the implementation is realized by other organizations, such as *Open MPI* or *MPICH*. [MPI Forum, 2015]
MPI primarily targets the message-passing parallel programming model, where data is moved from the address space of one process to that of another process. It was originally developed for distributed storage architectures where data was sent over a dedicated network. In the current version, however, shared memory systems can also be used efficiently [Gropp et al., 2014]. Figure 3 illustrates the use of MPI within *ARTSS*.

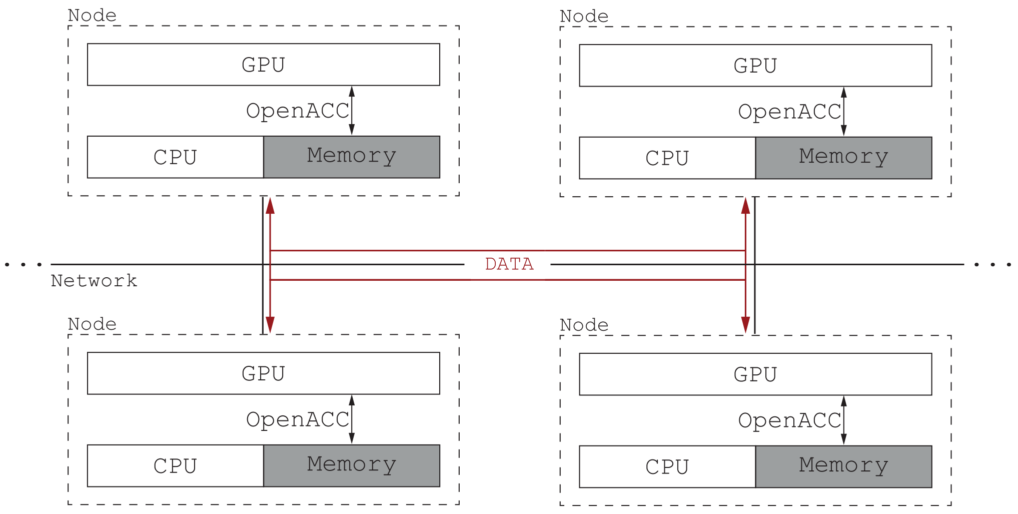


Figure 3: *MPI usage within ARTSS*

Each node consists of a CPU, a Random-Access-Memory unit (RAM) and a GPU. The communication between the CPU and the GPU is done using the *OpenACC* standard within a node. The communication between the nodes takes place via the point-to-point communication of MPI. A distinction is made between two types of point-to-point communication.

A blocking send call blocks until the send memory can be safely reused. Similarly, a blocking receive call blocks until the receive memory actually contains the intended message. Under certain circumstances, this can lead to the software experiencing a deadlock. To prevent this, the MPI standard also defines so-called non-blocking communication. This has the advantage that a send or receive function does not have to wait for the response of the respective partner function and therefore no deadlock can occur. If this strategy is used, it must be ensured that the data in the send memory is not changed before the process is completed. MPI provides special functions to check the status of a non-blocking communication. For point-to-point communication in *ARTSS*, the non-blocking variant is used.

Within MPI, the communication pattern of a set of processes can be represented by a topological pattern or more generally by a graph. This arrangement is called *virtual topology*. The structure of the *virtual topology* is defined with MPI. However, the mapping between the *virtual topology* and the underlying hardware is outside the capabilities of MPI and is done by the accompanying operating system. The *virtual topology* for the MPI implementation in *ARTSS* uses a 3-dimensional hypercube, which is shown in Figure 4. [MPI Forum, 2015]

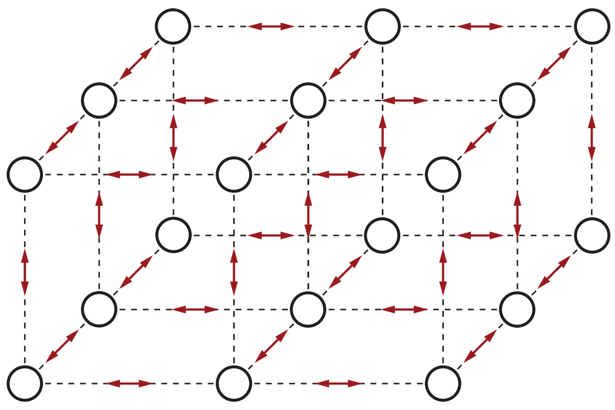


Figure 4: *Virtual Topology of MPI implementation in ARTSS. Red arrows indicate the communication links (non-periodic)*

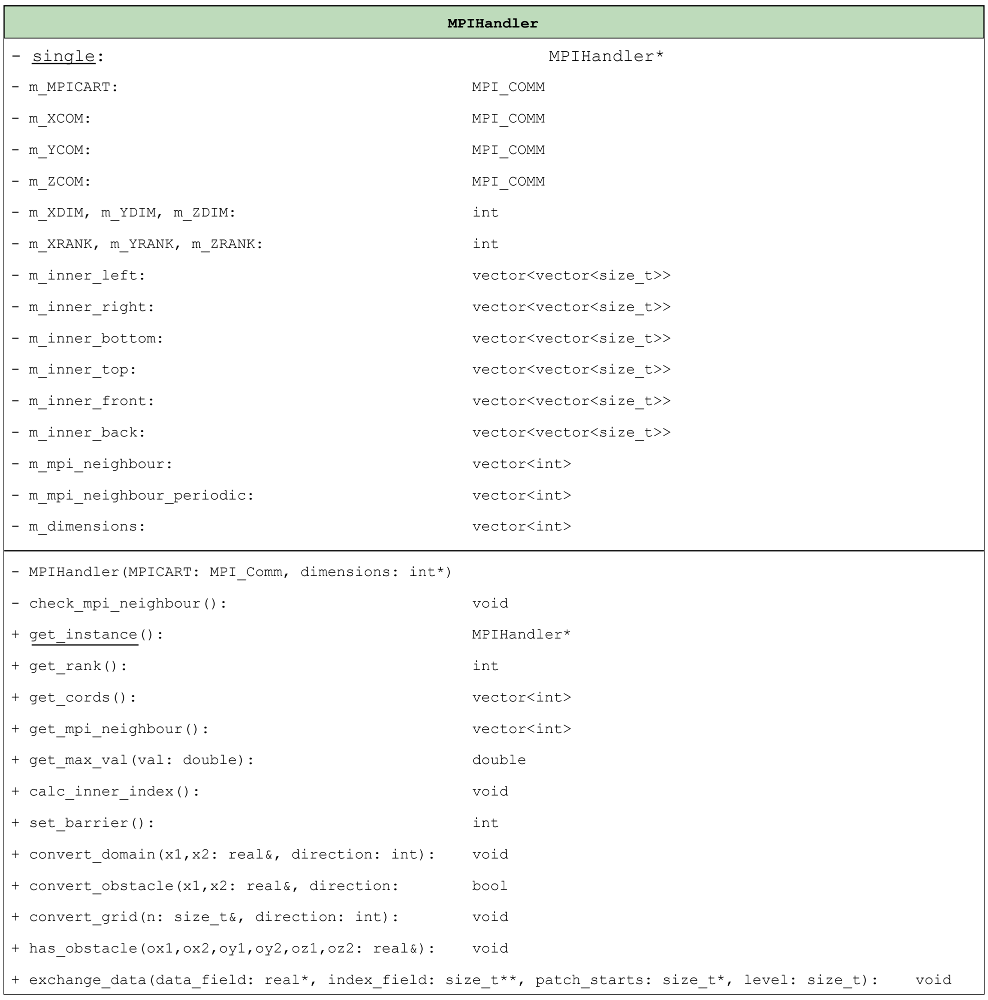
A MPI program is compiled with a compiler wrapper. They do not actually perform the compilation and linking steps themselves, but add the appropriate compiler and linker flags and call the underlying compiler and linker.

In order to run an MPI program, a special MPI command must be called and the number of processes to be used must be set.

```
mpiexec -n <number of processes> ...
```

2.2 MPIHandler class

To keep the code structure as independent as possible from the MPI implementation, the separate class `MPIHandler` is introduced within the utility group. This class serves as an intermediary between *ARTSS* and the MPI standard. Figure 5 shows the corresponding UML class diagram.



```
MPIHandler
- single:                                              MPIHandler*
- m_MPICART:                                 MPI_COMM
- m_XCOM:                                    MPI_COMM
- m_YCOM:                                    MPI_COMM
- m_ZCOM:                                    MPI_COMM
- m_XDIM, m_YDIM, m_ZDIM:                    int
- m_XRANK, m_YRANK, m_ZRANK:                 int
- m_inner_left:                              vector<vector<size_t>>
- m_inner_right:                             vector<vector<size_t>>
- m_inner_bottom:                            vector<vector<size_t>>
- m_inner_top:                               vector<vector<size_t>>
- m_inner_front:                             vector<vector<size_t>>
- m_inner_back:                              vector<vector<size_t>>
- m_mpi_neighbour:                           vector<int>
- m_mpi_neighbour_periodic:                  vector<int>
- m_dimensions:                              vector<int>

- MPIHandler(MPICART: MPI_Comm, dimensions: int*)
- check_mpi_neighbour():                     void
+ get_instance():                            MPIHandler*
+ get_rank():                                int
+ get_cords():                               vector<int>
+ get_mpi_neighbour():                       vector<int>
+ get_max_val(val: double):                  double
+ calc_inner_index():                        void
+ set_barrier():                             int
+ convert_domain(x1,x2: real&, direction: int):    void
+ convert_obstacle(x1,x2: real&, direction:        bool
+ convert_grid(n: size_t&, direction: int):        void
+ has_obstacle(ox1,ox2,oy1,oy2,oz1,oz2: real&):    void
+ exchange_data(data_field: real*, index_field: size_t**, patch_starts: size_t*, level: size_t):    void
```

Figure 5: *UML class diagram of MPIHandler - implementation level details*

In order for the *Message Passing Interface* to be used within *ARTSS*, the flag `-DUSEMPI` must first be set at the time of compilation.
Subsequently, the number of processes for a simulation can be configured within the XML file or via the command line. If the configuration is done using an XML file, the following parameters must be set within the `domain_parameters` tag.

```
...
<domain_parameters>
           ...
   <MESHX> ... </MESHX>
   <MESHY> ... </MESHY>
   <MESHZ> ... </MESHZ>
           ...
</domain_parameters>
...
```

In certain cases, it is useful to carry out the configuration via the command line. For this purpose, the flags `-x`, `-y` and `-z` are appended to the end of the running command.

```
mpiexec -n 8 artss ./simulation.xml -x 2 -y 2 -z 2
```

If the configuration is done via the command line, undefined flags automatically obtain the value 1. If the configuration is carried out through an XML file, all three parameters must be defined. As soon as the configuration is done from the command line, the values within the XML file are ignored.

2.3 Domain decomposition

Domain decomposition refers to the splitting of computational work among multiple processors by dividing the computational domain of a problem into smaller subdomains. This is referred to as a *graph partitioning problem.* The aim of solving this problem is to perform the decomposition in such a way that each subdomain has the same computational effort, therefore a balanced load, and that the communication for data exchange is minimized. To achieve this, several options are available. [George and Sarin, 2011]

- **Geometric Approaches**
  These approaches rely mainly on the physical mesh coordinates to partition the problem domain into subdomains corresponding to regions of space.

- **Coordinate-Free Approaches**
  These approaches are used in particular when the focus is on minimizing the communication between subdomains and the problem is not embedded in a geometric space.

- **Dynamic Approaches**
  These techniques are used when the structure of the problem changes dynamically. They focus on dynamically repartitioning the problem domain to ensure load balancing.

A key aspect of domain decomposition is the mapping of subdomains to the underlying hardware. The most efficient way is to assign a separate process to each subdomain. To keep communication times as short as possible, *virtual topology* and physical hardware should be similar.

A decentralized geometric approach is chosen as the method for domain decomposition in *ARTSS*. In this context, decentralized refers to the fact that each process reads in the configuration and initializes the corresponding data. This decentralized approach has the advantage that initial parameters do not have to be distributed across all processes. At the same time all subdomains in *ARTSS* have the same size. This makes configuration especially easy for the end user, since load balancing is automatically given by constant problem size and only three parameters have to be defined inside the simulation configuration.

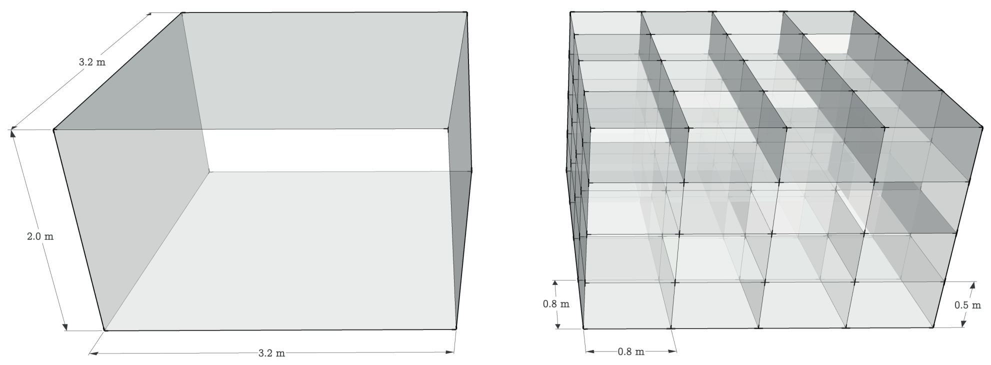


Figure 6: *Left: One physical domain with no subdomains; Right: One physical domain with 64 subdomains*

Figure 6 shows an example of a domain decomposition in *ARTSS*. The left side shows a physical domain with the dimensions 3.2 m × 3.2 m × 2.0 m and no subdomain. Therefore one single process computes the entire domain. The right side, on the other hand, shows the same domain with 64 subdomains. This is represented by the following configuration.

```
...
<domain_parameters>
           ...
   <MESHX> 4 </MESHX>
   <MESHY> 4 </MESHY>
   <MESHZ> 4 </MESHZ>
           ...
</domain_parameters>
...
```

By dividing the number of subdomains in each direction by the total length of the respective direction, you get for each domain a dimension of 0.8 m × 0.8 m × 0.5 m.

Due to the decentralized nature of domain decomposition in ARTSS, it is necessary that each subdomain computes its own physical dimensions, i.e. `X1`, `X2`, `Y1`, `Y2`, `Z1` and `Z2`. The `MPIHandler` class implements this behavior with the `convert_domain` function.

```
void MPIHandler::convert_domain(real& p1, real& p2, int direction)
```

This function receives three parameters. `p1` and `p2` are the respective start and end points of the entire physical domain. The integer value `direction` indicates one of the three spatial directions (0 = x, 1 = y, 2 = z). In the example above, there are a total of four subdomains in x-direction. Thus, the local number of the current subdomain is a value between zero and three. Defining the number of subdomains in the respective spatial direction as `total_domains`, the number of the current subdomain with `local_domain` and the physical local length with

$$local\_length = \frac{(p1 - p2)}{total\_domains}, \tag{2.1}$$

one gets with

$$local\_p1 = p1 + local\_domain \cdot local\_length \tag{2.2}$$

and

$$local\_p2 = p2 + (local\_domain + 1) \cdot local\_length \tag{2.3}$$

the new start and end points of the respective subdomain.
The `convert_grid` function is used to calculate the grid resolution in the corresponding subdomain.

```
void MPIHandler::convert_grid(size_t& n, int direction)
```

The parameter `n` is the number of grid points for the whole domain in the given `direction`. Defining the number of subdomains in the respective spatial direction as `total_domains` one gets with

$$local\_n = \frac{n - 2}{total\_domains} + 2 \tag{2.4}$$

the grid solution of the respective subdomain.

ARTSS uses *ghost cells* for the implementation of the boundary condition. Equation 2.4 ensures that each subdomain still has these *ghost cells*. If a subdomain has an adjacent subdomain, the data exchange between them is done through the *ghost cells*. If a subdomain does not have a neighbor, i.e., it is an outermost subdomain, then the *ghost cells* are still used for the implementation of the boundary conditions. Figure 7 illustrates the use of *ghost cells* within *ARTSS*.

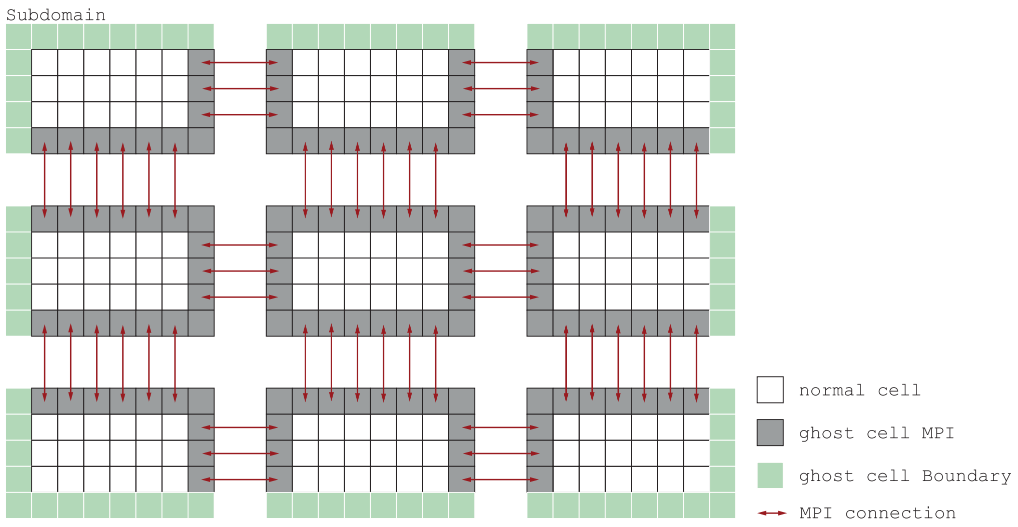


Figure 7: *Usage of ghost cells within ARTSS*

It should be noted that the *ghost cells* in the corner of a subdomain are not considered. This is due to the fact that *ARTSS* uses the *Finite- Difference-Method* for discretization. This method only considers the directly adjacent cells for the solution. For easier implementation, these corner cells are nevertheless taken into account during data exchange within this implementation. The influence on the communication time is negligible.

2.3.1 Obstacles

In the initialization phase of a simulation, *ARTSS* checks whether the calculation domain contains obstacles. The detection is done by checking whether obstacle parameters have been set in the configuration file.

This check is extended by the function `convert_obstacle`.

```
bool MPIHandler::convert_obstacle(real& o1, real& o2, int direction)
```

This function receives three parameters. `o1` and `o2` are the respective physical start and end points of the entire obstacle. The integer value `direction` indicates one of the three spatial directions (0 = x, 1 = y, 2 = z). Furthermore, it retrieves the physical start (`s1`) and end (`s2`) points of the subdomain and then checks six different cases shown in Figure 8. Depending on the case, physical parameters of the obstacle are then adjusted.

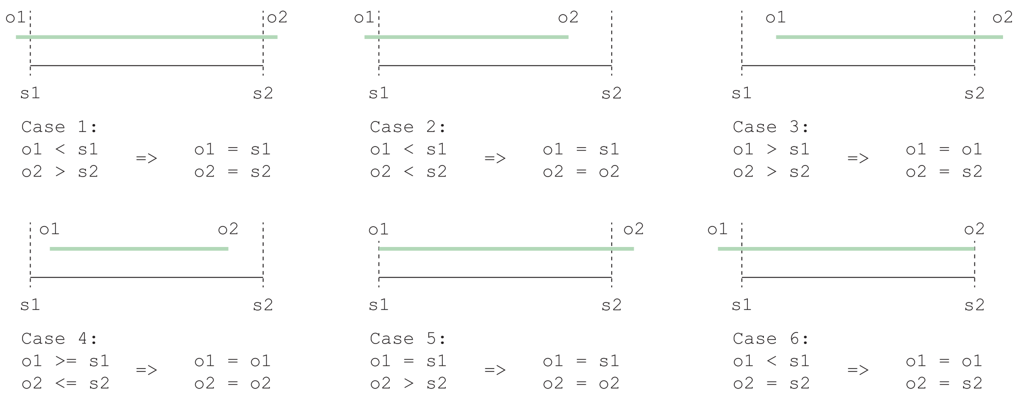


Figure 8: *Obstacle decomposition in ARTSS*

2.3.2 Sources

ARTSS supports temperature, concentration and momentum sources at the time of this work. These can be either point-shaped or volumetric, as cuboids. The point sources require only one `x`, `y` and `z` coordinate for orientation in space. The *Message Passing Interface* implementation must check in which subdomain the point source is located. If a point source is located directly between two subdomains, it is ignored. The check is done outside of the `MPIHandler` class using the physical dimensions of the respective subdomain.
The volumetric source works identically to the obstacles. The functionality of the obstacle decomposition, as shown in Figure 8 and explained in Chapter 2.3.1, can therefore be adopted.

2.3.3 Visualization and Logging

Depending on the needs, *ARTSS* generates different files during execution. For visualization the `vtk` file format is used, which can be opened with an external program and evaluated visually. Furthermore *ARTSS* generates `csv` files, which output the simulation data formatted to the defined time steps. At the end of a simulation, various `dat` files are created, which provide the raw cell value as well as the one-dimensional index of each cell. In addition to the visualization, *ARTSS* saves the progress of a simulation and the corresponding output in a log file.

A decentralized approach continues to be used for visualization and logging. This means that each subdomain generates its own output. To realize this and to avoid filename conflicts, each file gets a prefix to the actual filename.

```
<PREFIX>_<FILENAME>_<FILECOUNTER>.vtk
Example: 121_Test_0000001.vtk
```

The prefix is composed of the index of the respective subdomain. The first position is the x-direction, the second position is the y-direction and the third position is the z-direction. Figure 9 shows the numbering of the subdomains. It should be noted that the coordinate system used conforms to the MPI standard.

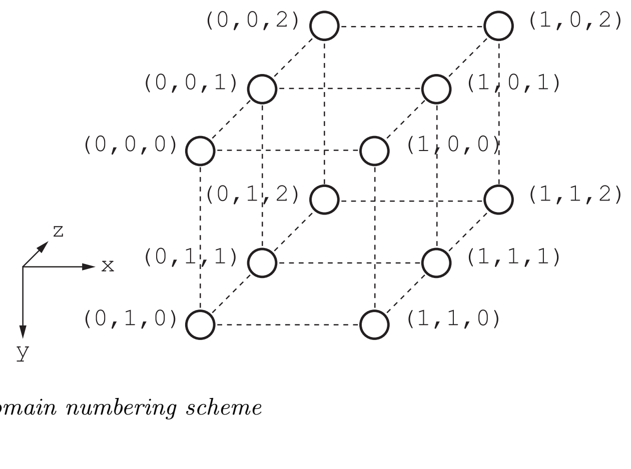


Figure 9: *Subdomain numbering scheme*

Ghost cells are also displayed in the visualization. In the serial version these contain the boundary conditions. However, since the *ghost cells* are also used for communication in the MPI implementation, an additional check must be implemented here. The ghost cells may only be displayed if they contain the boundary conditions. *Ghost cells* for data exchange must not be written to the `vtk` file.

2.4 Communication

ARTSS mainly uses the *Finite- Difference-Method* (FDM) to solve the governing equations. To approximate the derivatives at a given grid point, FDM uses a *7-Point-Stencil* in three dimensions or a 5-Point-Stencil in two dimensions. Thus, the approximation at a point depends on six, respectively four neighboring points. The left side of Figure 10 shows a two-dimensional grid with an imprinted *5-Point-Stencil.*

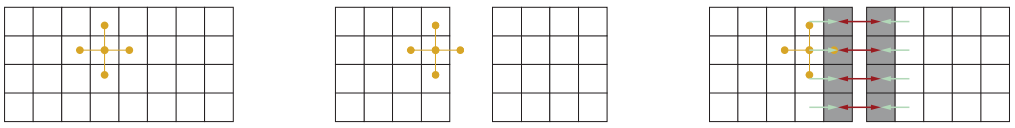


Figure 10: *Left: 5-Point-Stencil on computational grid; Middle: Computational grid after domain decomposition; Right: Communication via ghost cells*

After a domain decomposition, the problem arises that cells on the outside no longer have a neighbor, since this neighbor is located within the opposite subdomain. The middle image in Figure 10 illustrates this problem. The right neighbor of the *5-Point-Stencil* is missing. This problem can be solved with the use of *ghost cells.* As already described in Chapter 2.3, each subdomain has its individual *ghost cells.* These are initialized in each iteration step with the data of the outermost cells that are not ghost cells (hereinafter *direct adjacent cells*) and then sent to the neighboring subdomain using MPI. This scheme can be seen on the right side of Figure 10. The gray cells indicate the *ghost cells*, the red arrows represent the data links and the green arrows show the initialization of the *ghost cells.* The steps just described are explained in more detail in the following.

2.4.1 Retrieving the index of direct adjacent cells

To ensure that the *ghost cells* can be initialized with data from the *direct adjacent cells* in each iteration step, the indices of these are calculated once in the initialization phase of *ARTSS* using the `calc_inner_index` function.
It should be noted that *ARTSS* uses a *multigrid method* to solve the pressure equation. The goal of the *multigrid method* is to accelerate the convergence of a basic iteration method by solving a computationally cheaper problem on a coarse grid [Küsters, 2018]. Changing the cell sizes automatically brings a change in the grid resolution and thus a change in the data structure. Accordingly, the indices must be calculated for each level, thus for each grid resolution. Figure 11 shows an example of a two-dimensional grid on the left and the first multigrid level on the right.

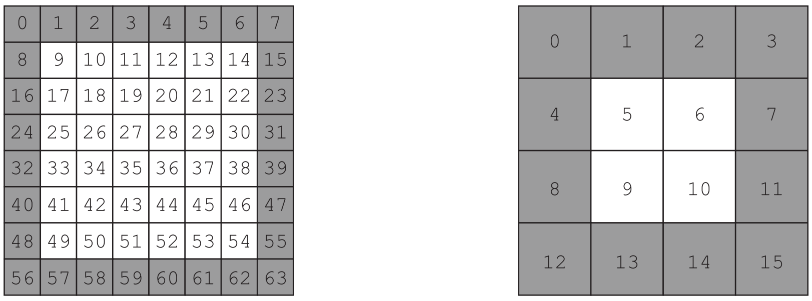


Figure 11: *Left: two-dimensional computational grid; Right: Multigrid level 1*

This example results in the following indices for the direct adjacent cells.

```
Level 0: 9,10,11,12,13,14,17,18,19,20,21,22,25,26,27,28,29,30,33,34,35,
         36,37,38,41,42,43,44,45,46,49,50,51,52,53,54

Level 1: 5,6,9,10
```

Depending on the grid resolution of the original grid, any maximum multigrid level can be defined by the end user. However, it is necessary that the coarsest grid resolution has at least four cells in each spatial direction.

2.4.2 Message Passing

The data exchange between the subdomains takes place in two steps. The first step is to update the boundary conditions for each subdomain, which can be either a Neuman, Dirichtlet, or periodic boundary condition. At this point it is not yet checked whether the subdomain has one of the three boundary conditions at all. In the second step the function `exchange_data` is called and the data exchange with the neighboring subdomains takes place.

```
void MPIHandler::exchange_data(real *data_field, size_t** index_fields,
                               const size_t *patch_starts, size_t level)
```

This function receives a total of four parameters. The variable `data_field` is an array which contains the actual physical values of the considered field (e.g. temperature, pressure, etc.). The indices for the respective patches of a grid cell (`TOP`, `BOTTOM`, `FRONT`, `BACK`, `LEFT`, `RIGHT`) are stored in the two-dimensional array `index_fields`. The start index of each patch is given in the list `patch_starts`. The variable `level` defines the multigrid level.
In the initialization phase of *ARTSS*, each subdomain checks whether a neighboring subdomain exists and stores the result and the direction of the subdomain in a list. This list is now looped through and if a neighbor is present, the following four steps are performed.

1. Depending on the direction of the neighboring subdomain, the indices belonging to the patch are determined in this step. The MPI process number of the neighboring subdomain is also determined, since this value is required for communication between the subdomains.

2. In this step the data of the *direct adjacent cells* are stored in a temporary send vector. At the same time a temporary receive vector of the same length is initialized. This ensures that send and receive data are separated from each other, otherwise they could be overwritten due to non-blocking communication.

3. Afterwards, the data in the send vector can be transmitted to the neighboring domain using the `MPI_Isend` function. After the data has been sent, the neighbor subdomain data is received by the `MPI_Irecv` function. Each communication is also tagged to uniquely associate a send/receive message pair. To complete the non-blocking communication the `MPI_Wait` function is called.

4. In the last step, the data of the receive vector is transferred to the respective *ghost cells* in the data field.

After the data has been exchanged between the subdomains, a check is made to see if periodic boundary conditions exist. For this the function `exchange_data` is called again. If periodic boundary conditions are present, the above steps are performed. For the communication this time not the process numbers of the directly neighboring subdomains are determined, but the process numbers of the periodically neighboring subdomains. It should be noted that each patch of a cell is processed only once either in the first or in the second function call.

Finally, all processes are synchronized by means of an `MPI_Barrier`. After this step the message passing for the respective field is completed.

Chapter 3

Analysis

In this analysis, the performance of the MPI implementation in ARTSS is tested. The MPI implementation can be used in three different ways.

1. **Multi CPU (Distributed Memory)**
   Here the parallelization is only done on CPUs using a *distributed memory system*. Each process gets its own memory area and data is only exchanged via the *Message Passing Interface*.

2. **Multi CPU (Hybrid Memory)**
   OpenACC-enabled source code can be compiled for parallel execution on either a multicore CPU or a GPU accelerator. In this approach MPI is used together with the multicore version of *ARTSS*. This combination is referred to as a *hybrid memory system*.

3. **Multi GPU** This combination allows *ARTSS* to use multiple GPU units. Each GPU is connected to a CPU core.

For a clear comparison, only the *Multi GPU* and *Multi CPU (Distributed Memory)* variants are compared in this analysis.

Scalability or *scaling* is often used to refer to the ability of software to perform calculations faster when the amount of resources is increased. Parallelization is implemented efficiently when the ratio between the actual speedup and the ideal speedup, using a certain number of processors, is minimized.

Speedup in the area of parallel computing is defined with

$$speedup = \frac{t_1}{t_N}. \tag{3.1}$$

t_1 represents the computing time of a calculation on one processor, where t_N refers to the computing time of the same calculation on N processors.
In the most optimal case, the speedup would increase linearly with the number of processors. In reality, however, the speedup is limited by the portion of the serial part of the software that cannot be parallelized. This statement can be described mathematically with *Amdahl's law* [Amdahl, 1967],

$$speedup = \frac{1}{s + \frac{p}{N}}, \tag{3.2}$$

where s is the serial fraction of the code, $p = 1 - s$ the paralellizable fraction and N the number of processes. *Amdahl's law* states that, for a fixed problem, the upper limit of speedup is determined by the serial fraction of the code. This is called *strong scaling*. Figure 12 shows the visual representation of *Amdahl's law*.

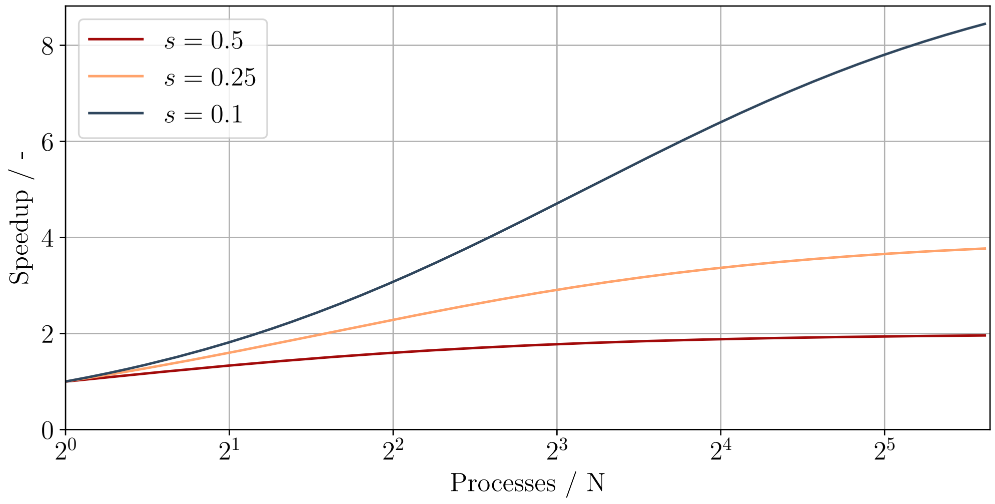


Figure 12: *Visual representation of the Amdahl's law (strong scaling)*

In contrast, *Gustafson's law*,

$$speedup = s + P \cdot N, \tag{3.3}$$

describes that the scaled speedup increases linearly with the number of processors and there is no upper bound on the scaled speedup [Gustafson, 1988]. This is called *weak scaling* and this statement is based on the assumption that the parallel part scales linearly with the amount of resources and that the serial part does not increase with the size of the problem. Figure 13 illustrates *Gustafson's law*.

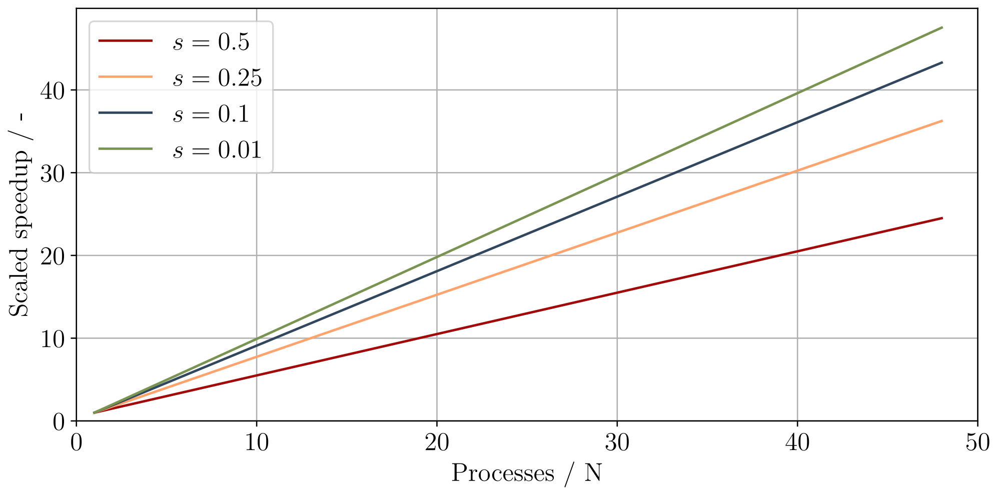


Figure 13: *Visual representation of the Gustafson's law (weak scaling)*

It can be seen that unlike *Amdahl's law*, *Gustafson's law* set has no upper limit. As a result, it is not advantageous to use a large amount of resources for computation when a problem requires only a small amount of resources. A more sensible choice is to use small amounts of resources for small problems and larger amounts of resources for large problems.

3.1 Verification

To determine the correctness of this implementation, verification is done by comparing the simulation data of the serial CPU version and the *Multi CPU*

(Distributed Memory) of *ARTSS*. The comparison is conducted on all test simulations available in *ARTSS* at the time of this work. For the compilation the *Open MPI* compiler wrapper (Version 4.0.4) is used, which in turn uses the *GCC* (Version 9.3) compiler. The simulation data of the MPI implementation is computed with six subdomains each (`x = 3`, `y = 2`, `z = 1` using the coordinates system of *ARTSS*). Subsequently, the results of the generated `csv` files are compared with each other. When comparing the data, no difference can be found between the serial version and the MPI implementation. Accordingly, the *Message Passing Interface* was successfully verified within *ARTSS*.

3.2 Scalability

The scalability analysis is performed on the supercomputer *JURECA* which is located at the Forschungszentrum Jülich. At the time of this analysis, the supercomputer is equipped with the following specifications [Forschungszentrum Jülich GmbH, 2020].

- 1872 compute nodes
  - Two Intel Xeon E5-2680 v3 Haswell CPUs per node
    - 2 x 12 cores, 2.5 GHz
    - Intel Hyperthreading Technology (Simultaneous Multithreading)
  - 75 compute nodes equipped with two NVIDIA K80 GPUs
    - 2 x 4992 CUDA cores
    - 2 x 24 GiB GDDR5 memory

The compilation is done with the PGI Compiler Suite (Version 19.1). This contains the *OpenACC* capable C++ compiler as well as an *Open MPI* compiler wrapper. All tests are performed in the benchmarking mode of *ARTSS*. That is, all outputs are deactivated and flags are set for compiler optimization.

3.2.1 Test setup

A simple tunnel setup is used as a test scenario for the performance analysis. Figure 14 shows the geometrical layout of the setup.

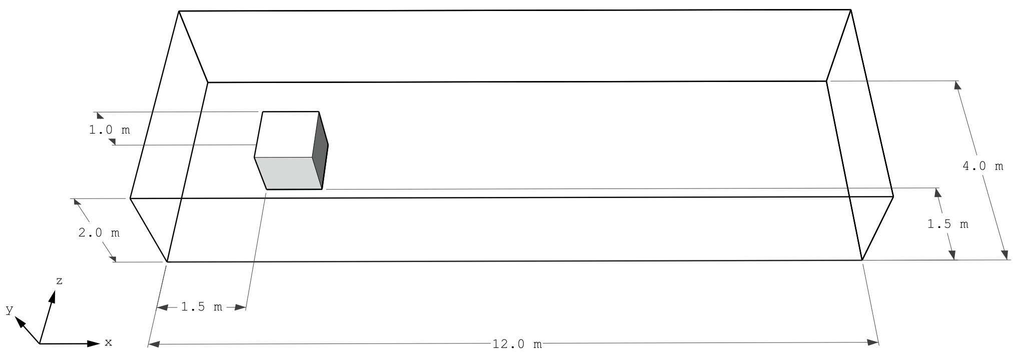


Figure 14: *Tunnel setup used in scalability analysis*

The grid resolution corresponds to 0.025 m in each spatial direction. This results in 480 cells in x-direction, 80 cells in y-direction and 160 cells in z-direction. The simulation time is 100 s with a fixed time step of 0.05 s. The background velocity is constant 0.8 $\frac{m}{s}$. Information about the used solver and boundary conditions can be found in Appendix A.

3.2.2 Strong Scaling

As mentioned above, the computational domain has a total of 6144000 cells. For the strong scaling analysis, this size is kept fixed. In total, five simulations for the *Multi GPU* and five simulations for the *Multi CPU* version are performed. The simulation runs differ in the number of processes respectively the number of subdomains.

- First run: 1 process (no domain decomposition)
- Second run: 2 processes (`-x 2, -y 1, -z 2` $\rightarrow$ 3072000 cells per domain)
- Third run: 4 processes (`-x 2, -y 1, -z 2` $\rightarrow$ 1536000 cells per domain)
- Fourth run: 8 processes (`-x 2, -y 2, -z 2` $\rightarrow$ 768000 cells per domain)
- Fifth run: 16 processes (`-x 4, -y 2, -z 2` $\rightarrow$ 384000 cells per domain)

For each simulation, the wall time, that is, the time taken for the calculation, and the time spent on the `exchange_data` function are measured. The results of the *Multi CPU* version are listed in Table 1. The results of the *Multi GPU* version are listed in Table 2. Figure 15 shows the results compared to *Amdahl's law.*

Multi CPU					
Processes	1	2	4	8	16
Wall time / s	6872.38	3765.52	1867.97	952.46	532.69
`exchange_data()`/ s	-	42.38	21.63	11.29	6.52
Speedup / -	-	1.82	3.67	7.22	12.09

Table 1: *Results of the strong scaling analysis for the Multi CPU Version*

Multi GPU					
Processes	1	2	4	8	16
Wall time / s	2428.39	1371.96	822.71	369.42	233.51
`exchange_data()`/ s	-	39.11	22.78	10.94	7.02
Speedup / -	-	1.77	2.95	6.5	10.40

Table 2: *Results of the strong scaling analysis for the Multi GPU Version*

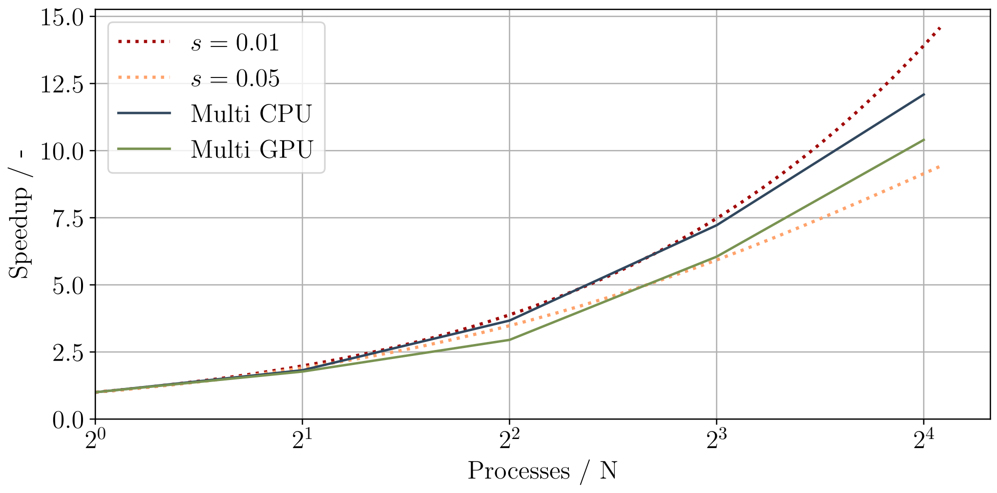


Figure 15: *Results of the strong scaling analysis compared to Amdahl's law*

The strong scaling analysis shows good results for the *Multi CPU* as well as the *Multi GPU* version. The acceleration of both versions is similar as predicted by *Amdahl's law*. However, it can be seen that the *Multi GPU* version deviates in the area of larger processor numbers. However, if one compares the times of the function `exchange_data` within the *Multi GPU*, it can be seen that these correspond to the values of the *Multi CPU* version. The reduction of the speedup with higher processors is therefore not due to the MPI implementation. The reason for the slowdown is due to the data exchange between CPU and GPU. With higher processor numbers, the problem to be calculated becomes smaller, which is why the data exchange becomes increasingly expensive compared to the actual calculation.

3.2.3 Weak scaling

In contrast to strong scaling, weak scaling does not change the number of cells per subdomain. As in the previous analysis, five simulations are performed, which differ in the number of processors.

- First run: 1 process (no domain decomposition)
- Second run: 2 processes (`-x 2, -y 1, -z 2` → 1536000 cells per domain)
- Third run: 4 processes (`-x 2, -y 1, -z 2` → 1536000 cells per domain)
- Fourth run: 8 processes (`-x 2, -y 2, -z 2` → 1536000 cells per domain)
- Fifth run: 16 processes (`-x 4, -y 2, -z 2` → 1536000 cells per domain)

Again, the wall time and the time spent on the `exchange_data` function are measured. The results of the *Multi CPU* version are listed in Table 3. The results of the *Multi GPU* version are listed in Table 4. Figure 16 shows the results compared to *Gustafson's law*.

Multi CPU					
Processes	1	2	4	8	16
Wall time / s	3624.53	3761.49	3837.31	3905.81	4021.41
`exchange_data()`/ s	-	40.98	65.31	83.28	113.88
Scaled speedup / -	-	1.93	3.77	7.42	14.42

Table 3: *Results of the weak scaling analysis for the Multi CPU Version*

Multi GPU					
Processes	1	2	4	8	16
Wall time / s	1251.93	1368.98	1390.30	1422.84	1469.52
`exchange_data()`/ s	-	38.55	63.01	84.12	112.72
Scaled speedup / -	-	1.82	3.45	7.03	13.62

Table 4: *Results of the weak scaling analysis for the Multi GPU Version*

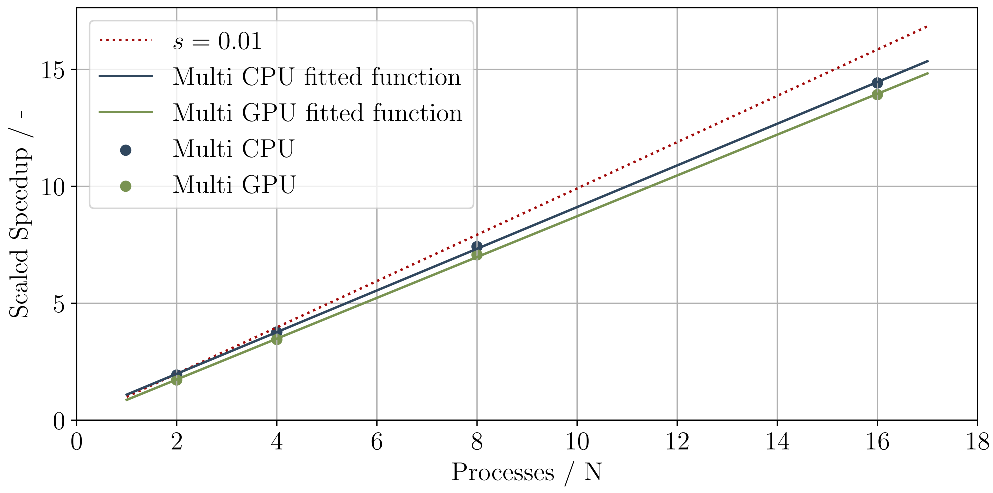


Figure 16: *Results of the weak scaling analysis compared to Gustafson's law*

For the weak scaling analysis it is important that both versions show a consistent progression to each other, which can be clearly seen in Figure 16. Comparing the results of the strong and weak scaling analysis when using two subdomains, a very good agreement in the wall time can be observed. This is expected, since in this case the domain sizes in weak and strong scaling correspond.

Chapter 4

Summary and Outlook

The goal of this work was to give ARTSS the ability to use multiple GPUs in a simulation. This not only allows for faster computation of a simulation, but also allows for theoretically unlimited use of GPU memory. Thus, for example, the previously existing maximum value for cells per simulation is no longer applicable, since the computational domain can now be divided among different GPUs by means of domain decomposition.
In order to achieve this goal, the *Message Passing Interface* was implemented. To avoid changing the structure of the existing code, a separate `MPIHandler` class was implemented within the `utility` group. To ensure that *ARTSS* can perform serial computations, all MPI functions are wrapped with conditional inclusion statements, which can be activated or deactivated during compilation. A decentralized approach was chosen as the parallelization strategy. That is, each process reads the configuration file for itself and performs the domain decomposition with the help of the `MPIHandler` class. Due to this decentralized approach, the data at the edges of the subdomains must be exchanged in each iteration step, since these are needed for the *7-Point-Stencil* of the *Finite-Difference-Method.* Data exchange is performed by using non-blocking communication. For this purpose, the *direct adjacent cells* of each subdomain are copied into temporary sender vectors. After successful communication with the neighboring subdomain, the received data is stored inside the *mul.*

The performance analysis showed that the MPI implementation scales well and provides the desired speedup. It should be noted, of course, that the raw data is only valid on the hardware used in the work. Nevertheless, it is expected that the scalability is transferable to other systems that support the *Message Passing Interface* standard.

The analysis in this thesis was performed in November 2020 on the supercomputer *JURECA* at Forschungszentrum Jülich. In December 2020, the complete hardware and software architecture of *JURECA* was renewed. A renewed analysis of the MPI implementation on the latest hardware would therefore be possible and advisable.

Furthermore, there is still room for improvement in the area of data output. Currently, each process generates its own output. Especially the visual evaluation of the `VTK` files is very tedious. Further tools should be written, which simplify the handling of the MPI implementation. There is also room for improvement in the area of data output. Currently, each process generates its own output. With a high number of processes, this can result in a very large number of files, which have a confusing effect on the project. Especially the visual evaluation of the `vtk` files is very tedious. More tools should be written to simplify the handling of the MPI implementation.

The *OpenACC* standard allows *ARTSS* to be independent from different hardware vendors. However, it is worth considering using the `GPUDirect` function offered by Nvidia for the *Multi Gpu* variant. This allows data to be exchanged between GPUs directly via the PCIe bus, without having to take a detour via main memory. An interesting approach can be found in [Sourouri et al., 2014].

Bibliography

Amdahl, G. M. [1967], Validity of the single processor approach to achieving large scale computing capabilities, *in* 'Proceedings of the April 18-20, 1967, Spring Joint Computer Conference', AFIPS '67 (Spring), Association for Computing Machinery, New York, NY, USA, p. 483–485.
URL: *https://doi.org/10.1145/1465482.1465560*

Forschungszentrum Jülich GmbH [2020], 'JURECA - Configuration', *Institute for Advanced Simulation (IAS)* .
URL: *https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/Configuration/Configuration_node.html*

George, T. and Sarin, V. [2011], *Domain Decomposition*, Springer US, Boston, MA, pp. 578–587.
URL: *https://doi.org/10.1007/978-0-387-09766-4_291*

Gropp, W., Hoefler, T., Thakur, R. and Lusk, E. [2014], *Using Advanced MPI: Modern Features of the Message-Passing Interface*, The MIT Press.

Gustafson, J. L. [1988], 'Reevaluating amdahl's law', *Commun. ACM* **31**(5), 532–533.
URL: *https://doi.org/10.1145/42411.42415*

Hwu, W., Keutzer, K. and Mattson, T. G. [2008], 'The concurrency challenge', *IEEE Design Test of Computers* **25**(4), 312–320.
URL: *https://ieeexplore.ieee.org/document/4584454*

Kirk, D. B. and Hwu, W.-m. W. [2016], *Programming Massively Parallel Processors, Third Edition: A Hands-on Approach*, 3rd edn, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Küsters, A. [2018], Real-Time Simulation and Prognosis of Smoke Propagation in Compartments Using a GPU, Dissertation, Bergische Universität Wuppertal, Jülich.
URL: *https://juser.fz-juelich.de/record/860655*

Levin, E. [1989], ‘Grand challenges to computation science’, *Commun. ACM* **32**(12), 1456–1457.
URL: *https://doi.org/10.1145/76380.76385*

MPI Forum [2015], *MPI: A Message-Passing Interface Standard, Version 3.1*, High Performance Computing Center Stuttgart (HLRS).
URL: *https://fs.hlrs.de/projects/par/mpi//mpi31/*

Nvidia [2020*a*], ‘NVIDIA TESLA P100’, *Performance specification* .
URL: *https://www.nvidia.com/en-us/data-center/tesla-p100/*

Nvidia [2020*b*], OpenACC getting started guide, techreport.
URL: *https://www.pgroup.com/resources/docs/20.4/pdf/openacc20_gs.pdf*

Sourouri, M., Gillberg, T., Baden, S. B. and Cai, X. [2014], Effective multi-gpu communication using multiple cuda streams and threads, *in* ‘2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)’, pp. 981–986.

Sterling, T., Anderson, M. and Brodowicz, M. [2018], Chapter 1 - introduction, *in* T. Sterling, M. Anderson and M. Brodowicz, eds, ‘High Performance Computing’, Morgan Kaufmann, Boston, pp. 1 – 42.
URL: *http://www.sciencedirect.com/science/article/pii/B9780124201583000010*

University of Washington [2020], ‘CPU performance’.
URL: *https://boinc.bakerlab.org/rosetta/cpu_list.php*

Würzburger, M. L. [2019], Dynamic Domain Adaption for Smoke Simulation in JuROr, Masterarbeit, FH Aachen. Masterarbeit, FH Aachen, 2019.
URL: *https://juser.fz-juelich.de/record/868016*

Appendix A

Inputfile Tunnel-Setup

```
<?xml version="1.0" encoding="UTF-8" ?>
<ARTSS>

  <physical_parameters>
    <t_end> 40. </t_end>
    <dt>   0.01 </dt>
    <nu> 0.00001 </nu>
  </physical_parameters>

  <domain_parameters>
    <X1> 0. </X1>
    <X2> 12. </X2>
    <Y1> 0. </Y1>
    <Y2> 2. </Y2>
    <Z1> -2.0 </Z1>
    <Z2> 2.0 </Z2>
    <x1> 0. </x1>
    <x2> 12. </x2>
    <y1> 0. </y1>
    <y2> 2. </y2>
    <z1> -2.0 </z1>
    <z2> 2.0 </z2>
    <nx> 480 </nx>
    <ny> 80 </ny>
    <nz> 160 </nz>
  </domain_parameters>

  <adaption dynamic="No" data_extraction="No"> </adaption>
```

(Continued on the next page)

```xml
<solver description="NSTurbSolver" >

  <advection type="SemiLagrangian" field="u,v,w">
  </advection>

  <diffusion type="Jacobi" field="u,v,w">
    <max_iter> 100 </max_iter>
    <tol_res> 1e-07 </tol_res>
    <w> 1 </w>
  </diffusion>

  <turbulence type="ConstSmagorinsky">
    <Cs> 0.2 </Cs>
  </turbulence>

  <source type="ExplicitEuler" force_fct="Zero" dir="xyz">
  </source>

  <pressure type="VCycleMG" field="p">
    <n_level> 4 </n_level>
    <n_cycle> 2 </n_cycle>
    <max_cycle> 4 </max_cycle>
    <tol_res> 1e-07 </tol_res>
    <diffusion type="Jacobi" field="p">
          <n_relax> 4 </n_relax>
          <max_solve> 100 </max_solve>
          <tol_res> 1e-07 </tol_res>
          <w> 0.6666666667 </w>
    </diffusion>
  </pressure>

  <solution available="No"></solution>

</solver>
```

(Continued on the next page)

```xml
<boundaries>
  <boundary field="u"
      patch="left,right" type="dirichlet" value="0.4" />
  <boundary field="v,w"
      patch="left,right" type="dirichlet" value="0.0" />
  <boundary field="u,v,w"
      patch="front,back,bottom,top" type="neumann" value="0.0" />
  <boundary field="p"
      patch="front,back,bottom,top,left,right" type="neumann"
          value="0.0" />
</boundaries>

<obstacles enabled="Yes">
  <obstacle>
    <geometry ox1="1.5" ox2="2.5"
      oy1="0.5" oy2="1.5" oz1="-0.5" oz2="0.5"/>
    <boundary field="u,v,w"
      patch="front,back,left,right,bottom,top" type="dirichlet"
          value="0.0" />
    <boundary field="p"
      patch="front,back,left,right,bottom,top" type="neumann"
          value="0.0" />
  </obstacle>
</obstacles>

<surfaces enabled="No"/>
```

(Continued on the next page)

```
  <initial_conditions usr_fct="Drift" random="No">
    <u_lin> 0.8 </u_lin>
    <v_lin> 0.0 </v_lin>
    <w_lin> 0.0 </w_lin>
    <pa> 0. </pa>
  </initial_conditions>

  <visualisation save_vtk="Yes" save_csv="No">
    <vtk_nth_plot> 100 </vtk_nth_plot>
  </visualisation>

  <logging file="info.log" level="info">
  </logging>
</ARTSS>
```