Department of Physics and Astronomy

University of Heidelberg

Master thesis

in Computer Engineering

submitted by

Benjamin Zach

born in Erlangen

2017

Domain Decomposition with coarsened and fine halos

This Master thesis has been carried out by Benjamin Zach

at the

Institute of Computer Engineering

under the supervision of

Prof. Dr. Robert Strzodka

Abstract

Abstract goes here

For Dada

Acknowledgements

I want to thank...

Contents

Chapter 1

Introduction

Chapter 2

Applications

Chapter 3

Approach

Chapter 4

Implementation

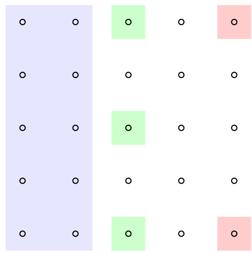

Figure 4.1: Halo structure for `_halo_t halo_property = {2, 1, 1}`: two lines of coarseness 0 (entirely occupied), one line of coarsenesses 1 and 2, respectively. The resulting halo size is `halo_size = 5`.

4.1 Shape

4.1.1 Type definitions

The following type definitions are made:

```
typedef int _int_t;
typedef int _coord_t;
typedef std::vector< int > _halo_t;
typedef std::array<int, 3> _vector_t;
```

4.1.2 Halos

The form of the halos is referred to as `halo_property` and can be defined using the aforesaid typedef. The `int` at position 0 hereby indicates the number of lines to be sent entirely, the `int` at index 1 represents the number of lines coarsened by the factor 2^1 et cetera. Thus at index i is given the number of lines for the coarseness 2^i. In the following we identify i as the order of coarseness. See figure 4.1.2 for an example. The term `halo_size` indicates the overall number of halo layers and is calculated iteratively as

$$\begin{aligned}\texttt{halo\_size}_0 &= \texttt{halo\_property[0]} \\ \texttt{halo\_size}_n &= \left(\frac{\texttt{halo\_size}_{n-1}+1}{2^n} + \texttt{halo\_property[n]}\right)\cdot 2^n + 1.\end{aligned} \tag{4.1}$$

4.1.3 Domain dimensions

We name disaggregated all the classes that are split up among the MPI processes, like `Process_domain` and `Operator`. In this context, `global_size` denotes the size of the objects' overall conjunction. We define `begin` of a dissaggregated object as the tuple containing the object's smallest index in each dimension and `end` the tuple of smallest indeces outside the object. With this it is possible to calculate an object's `domain_size` as the difference between its end and begin in each dimension. Eventually each object has its `total_size` which adds the surrounding halo layers in each dimension. Consequently it consists of the `domain_size` + 2 * `halo_size` in each dimension.

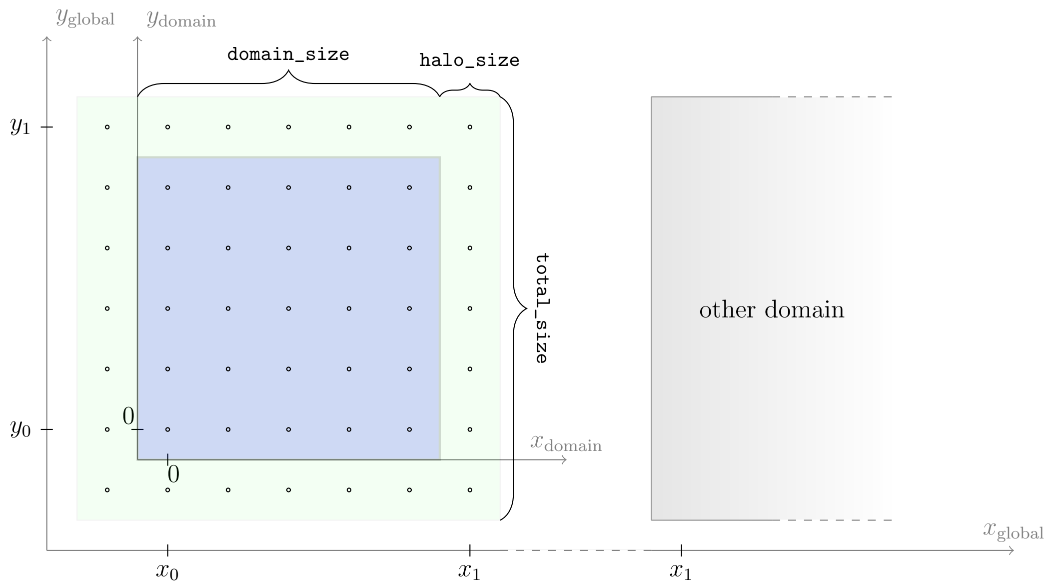

Figure 4.2: Two dimensional sketch of a `Process_domain` object with `begin=`$\{x_0, y_0\}$ and `end=`$\{x_1, y_1\}$ defined in the global coordinate system over all process domains) and the lengths `domain_size`, `halo_size` and `total_size`.

4.1.4 Data structure

Here has to be: - scheme how data is indexed - scheme of aggergated data - how fct Domain_shape::index() works

4.1.5 The class 'Shape'

Eventually the class `Shape` combines the concepts presented in 4.1.2, 4.1.3 and 4.1.4. It indeed provides attributes to store the domain's `begin` and `end` - in the global coordinate system - and the extents `domain_size`, `total_size` and `global_size`. Its function `index` converts a point's three dimensional coordinates into the associated index in the data storage by

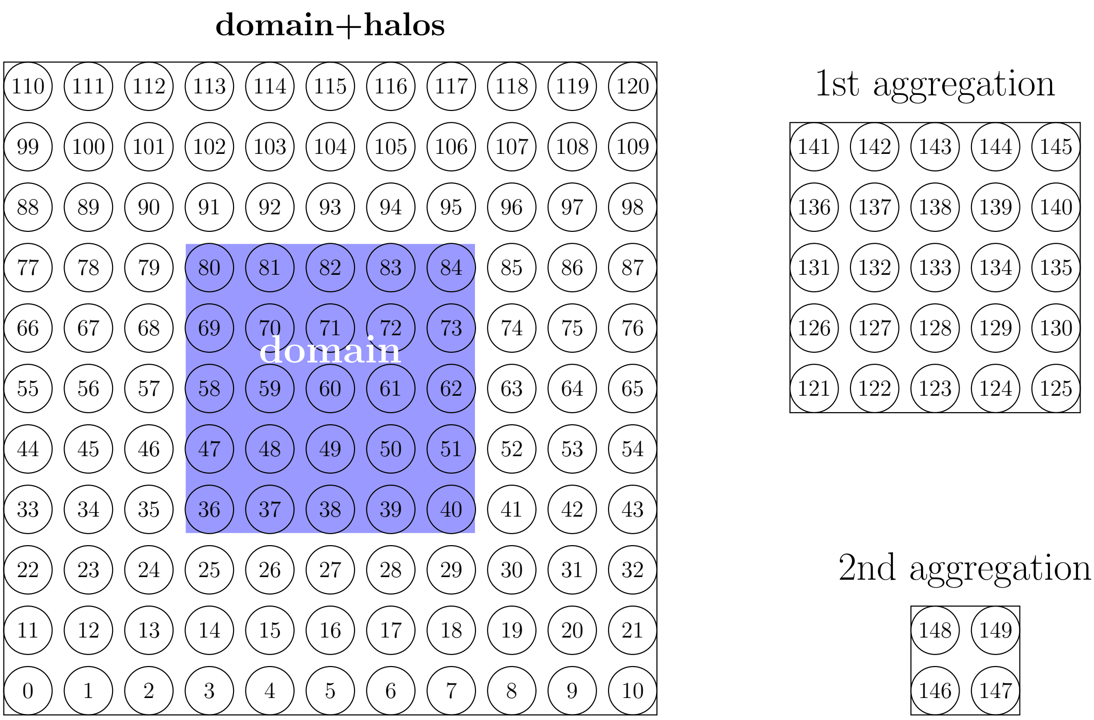

Figure 4.3: Scheme for the indices of a process domain's data points in the underlying storage structure: the first section locates the domain itself with the surrounding halos. Directly hereafter follow aggregated values, always the complete coarsened domain at a time.

```
    return (x+halo_size_) + (y+halo_size_) * total_size[X] +
                            (z+halo_size_) * total_size[X] * total_size[Y];
```

The remaining two function serves for accessing the `halo_size`.

Shape
begin, end : _vector_t # domain_size, total_size, global_size : _vector_t # halo_size_ : _int_t - class_tag : std::string
+ index(_coord_t, _coord_t, _coord_t) constant : _int_t + halo_size() constant : constant _int_t&

4.1.6 Logical Position

The logical position `log_pos` of a process P describes whether it has neighboring processes in a specified dimension. It is of interest for deciding whether a halo zone is among two processes or on the very outside as different behaviours may be desired. For its definition the declarations

$$N_c^+ := P \text{ has a successive neighbour in dimension } c$$

$$N_c^- := P \text{ has a preceding neighbour in dimension } c$$

are made. According to the combination of both the definition is made as

$$\texttt{log\_pos[c]} = \begin{cases} 1 & if \quad N_c^+ \wedge \neg N_c^- \\ 0 & if \quad N_c^+ \wedge N_c^- \\ -1 & if \quad \neg N_c^+ \wedge N_c^- \\ -4 & if \quad \neg N_c^+ \wedge \neg N_c^- \end{cases} \tag{4.2}$$

4.2 Policies

4.2.1 Data storage

The abstract class `Storage` works as a uniform interface to access the data. The base type is defined with template argument `typename type__`. Its provides functions for constant and non-constant array access:

```
virtual type__& operator[](unsigned int) = 0;
virtual const type__& operator[](unsigned int) const = 0;
```

Two concrete implementations are subclasses of the above-named. `Storage_various` is the design to actually hold a value for every data point the domain. Therefore it keeps the memmber `std::vector<type__> data` to store these values. The instanciation works via the constructor `Storage_various(unsigned int size, type__ val=0)`, featuring a parameter for the size and an optional parameter to define a standard value. On the other `Storage_const` facilitates the storage of a unique value as `type__ data` and can be instanciated the analog way `Storage_various(unsigned int, type__` though the size parameter does not have to be used. In distinguishing these two it is possible to save memory for cases where a data containing stores one constant value.

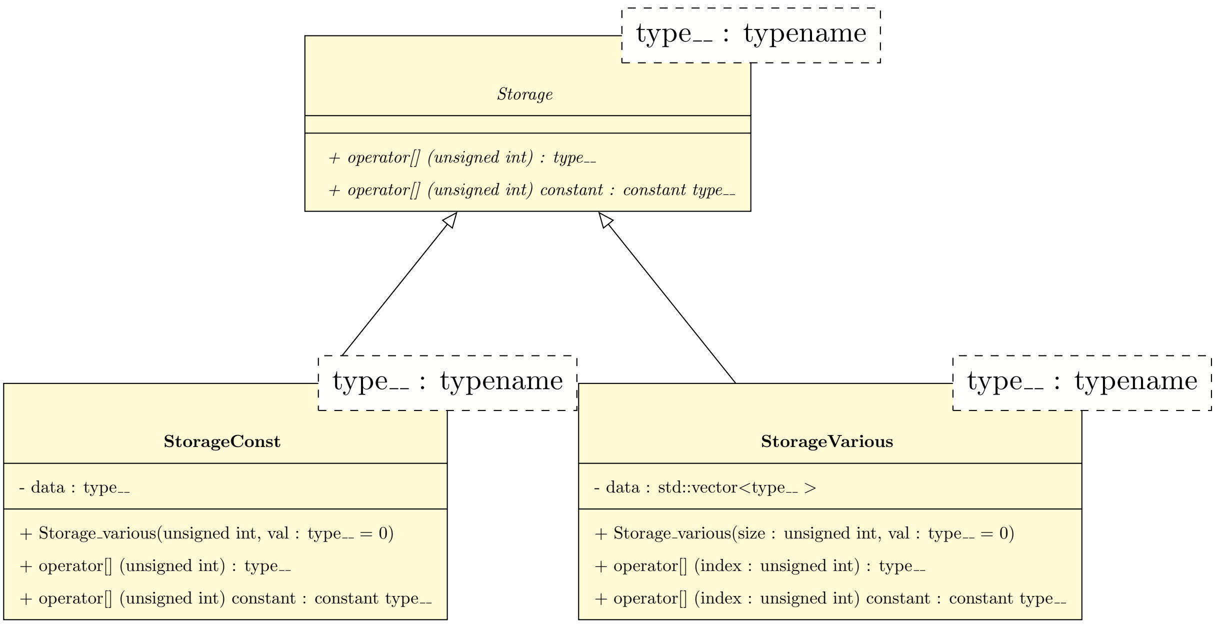

Figure 4.4: Attributes and relations among the storage classes `Storage`, `Storage_const` and `Storage_various`.

4.2.2 Data access

The abstract class `Accessible` provides the data access interface. Its template parameters define the base type of the data `typename type__` and the number of underlying storages `unsigned int env__`. The general interface for constant and non-constant element access is then defined as

```
virtual type__& get (_coord_t, _coord_t, _coord_t, _coord_t) = 0;
virtual const type__& get (_coord_t, _coord_t, _coord_t, _coord_t) const = 0;
```

Its four parameters split in one to address a certain storage and three actual coordinates. For the case of one single underlying storage a partial template specialization `Accessible<1, type__>` exists: The interface then specifies the same functions, yet only the three coordinate parameters are needed.

4.3 Processor_grid

The class `Processor_grid` reproduces the topology of the MPI processes. It provides functions to obtain the id of a process with relative coordinates to the calling process or the calling process' own:

```
int operator() ( int rel_x, int rel_y, int rel_z ) const;
const int& operator() () const;
```

Furthermore it facilitates access to the process' absolute and the logical position in the grid, where the latter for a process P is defined as

```
const int& abs_pos(Coords c) const;
const int& log_pos(Coords c) const;
const std::array<int,3>& get_log_pos() const
```

```
const int& size ( Coords c ) const;
const int& size () const;
MPI_Comm& getCommunicator();
```

4.4 Type_container

4.4.1 Basic functionality

The `Type_container` can be instanciated by using its public constructor

```
Type_container(const Domain_shape& shape)
```

and provides references to the send and receive types. The associated functions are called

```
const MPI_Datatype& get_send_data_type(int x, int y, int z) const,
const MPI_Datatype& get_recv_data_type(int x, int y, int z) const.
```

. These types simplify the communication process as we do not any more need to copy the relevant data to or from buffers, respectively, in order to have it in

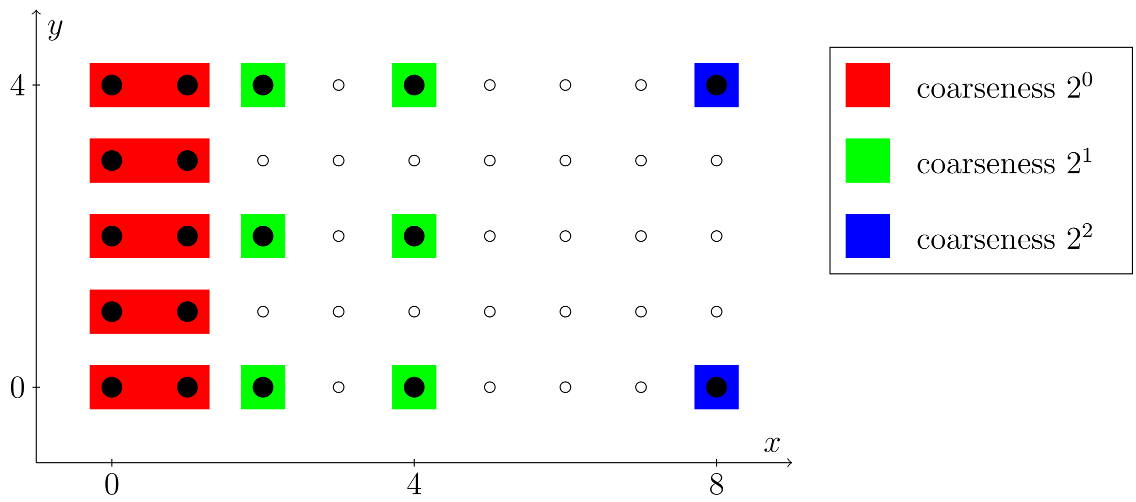

Figure 4.5: Scheme for the block structure of the receive types: for the uncoarsened values (2^0) each line is one block; for all coarsened areas each value forms its own block.

a continuous piece of memory. This task is rolled out to other routines provided by MPI. It is only necessary to define and assemble the positions of all relevant memory locations. Moreover it is possible to interpret messages differently and therefore reconstruct the data to unequal positions on sending than on receiving side. The base MPI type of the values is handed via a template argument. Valid examples hereof are `MPI_INT` or `MPI_DOUBLE`; the total list can be found at [For15], tables 3.2 to 3.4.

4.4.2 MPI datatype construction routines

MPI provides a variety of routines to construct customizes data types. In general one can define the parameters:

- *count*: the number of blocks or elements the new type consists of
- *length*: the number of consecutive elements in one block
- *displacements* or *stride*: the displacements of the blocks or elements or the constant stride among them
- *oldtype*: the base type of a single element

The different constructors vary in whether these parameters are constant within the new type.

In the implementation at hand two new types are interleaved within one exchange type: the inner type bundles up all values of a certain coarseness; for example one inner type for all uncoarsened, one for all coarsened by 2^1 and so on. Hence the parameters have the following characteristic: the blocklength remains constant (either the number of planes to be sent, or 1), we give an array of displacements with respect to the first point in the domain as in general there is no constant stride among the data points and the oldtype is the constant base type. For this case MPI provides the routine

```
int MPI_Type_create_indexed_block(int count, int blocklength, const int displacements[],
      MPI_Datatype oldtype, MPI_Datatype *newtype).
```

In order to pack all types in one outer type we later use

```
int MPI_Type_create_struct(int count, const int blocklengths[], const MPI_Aint displacements[],
      const MPI_Datatype types[], MPI_Datatype *newtype).
```

which is the simplest possibility to package the different inner types. In this routine allthough they all use the point 0 for reference, it is necessary to specify a whole array of displacements as 0's. The same holds for the blocklength which is actually always 1 throughout the type.

4.4.3 Receive types

4.4.4 Initialization process

The constructor requires a reference to an object of `Domain_shape`, which contains all necessary information about the size and halo properties of the process' domains. It is used to perform init routine `void init_data_types(const Domain_shape&)`, which splits up in a phase for the send and the receive types, respecively.

4.5 Logging

For debugging purposes the logging facility offered by [Mar07] is used.

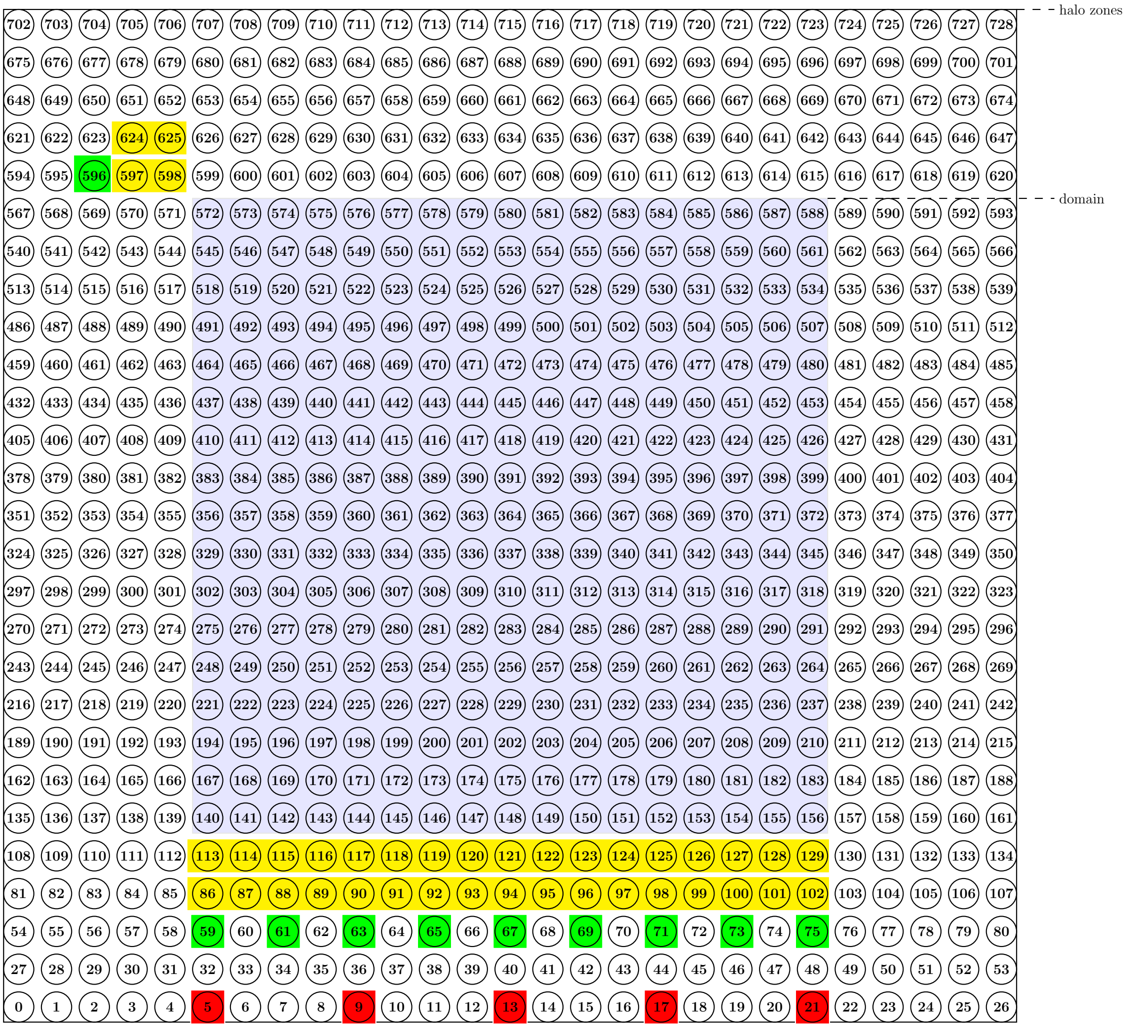

Figure 4.6: ;kj;

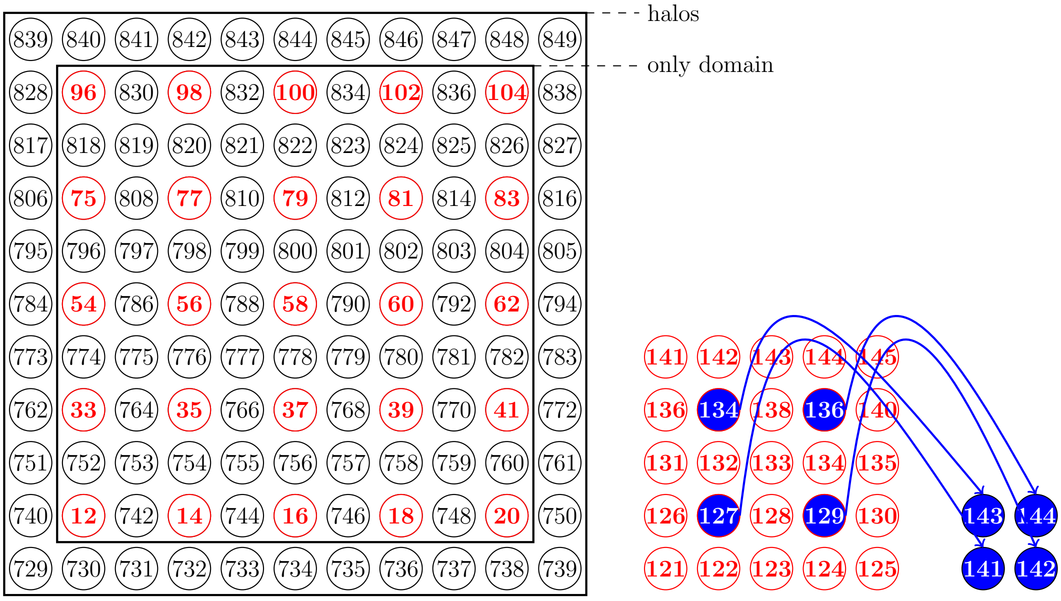

Figure 4.7: ;kj;

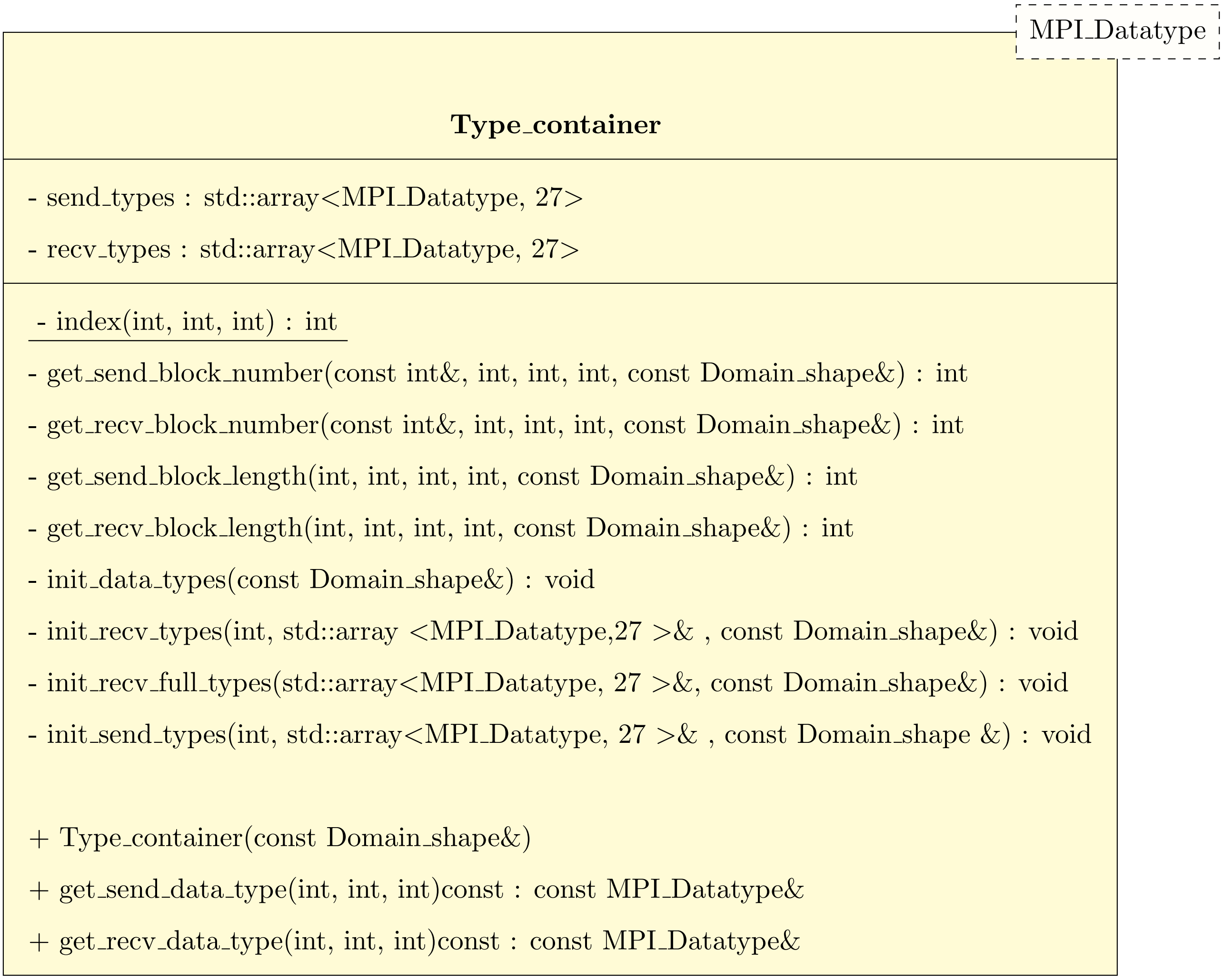

Figure 4.8: Attributes of the class `Type_container`.

Chapter 5

Benchmark Tests

Chapter 6

Conclusion

Bibliography

[For15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard. Version 3.1.* 2015.

[Mar07] Petru Marginean. *Logging In C++.* 2007. URL: `http://www.drdobbs.com/cpp/logging-in-c/201804215?pgno=1`.

Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 15.06.2017 Unterschrift: . . . .