# schwarz-lib

Generated automatically from umfpack-fact

Generated by Doxygen 1.8.13

# Contents

# Chapter 1

# Main Page

This is the main page for the Schwarz library pdf documentation. The repository is hosted on `github`. Documentation on aspects such as the build system, can be found at the # Installation Instructions page.

#### Modules

The structure of the Schwarz Library code is divided into different `modules` :

- Initialization : Handles the initialization of the problem and the solver.
- Communicate : Handles the communication.
- Solve : Handles the local solution and the convergence detection.
- Schwarz Class : The Classes related to the Schwarz solvers.
- Utils : Provides some basic utilities.

# Chapter 2

# # Installation Instructions

### Building

Use the standard cmake build procedure:

```
mkdir build; cd build
cmake -G "Unix Makefiles" [OPTIONS] .. && make
```

Replace `[OPTIONS]` with desired cmake options for your build. The library adds the following additional switches to control what is being built:

- `-DSCHWARZ_BUILD_BENCHMARKING={ON, OFF}` Builds some example benchmarks. Default is `ON`
- `-DSCHWARZ_BUILD_METIS={ON, OFF}` Builds with support for the `METIS` partitioner. User needs to provide the path to the installation of the `METIS` library in `METIS_DIR`, preferably as an environment variable. Default is `OFF`
- `-DSCHWARZ_BUILD_CHOLMOD={ON, OFF}` Builds with support for the `CHOLMOD` module from the Suitesparse library. User needs to set an environment variable `CHOLMOD_DIR` to the path containing the `CHOLMOD` installation. Default is `OFF`
- `-DSCHWARZ_BUILD_CUDA={ON, OFF}` Builds with CUDA support. Though Ginkgo provides most of the required CUDA support, we do need to link to CUDA for explicit setting of GPU affinities, some custom gather and scatter operations. Default is `OFF`.
- `-DSCHWARZ_BUILD_CLANG_TIDY={ON, OFF}` Builds with support for `clang-tidy` Default is `OFF`
- `-DSCHWARZ_BUILD_DEALII={ON, OFF}` Builds with support for the finite element library `deal.ii` Default is `OFF`
- `-DSCHWARZ_WITH_HWLOC={ON, OFF}` Builds with support for the hardware locality library used for binding hardware. `hwloc` is distributed as a part of the Open-MPI project. Default is `ON`
- `-DSCHWARZ_DEVEL_TOOLS={ON, OFF}` Builds with some developer tools support. Default is `ON`. In particular uses `git-cmake-format` to automatically format the source files with `clang-format`.

### Tips

- If you are having CUDA problems and you are not using CUDA, then feel free to switch the CUDA module off with `-DSCHWARZ_BUILD_CUDA=off`.
- Installing CHOLMOD can be a bit annoying. TODO add some details on fixing Suitesparse compilation.
- When doing merge commits it is possible that make format does not work. You can run `cmake -DSCH↩ WARZ_DEVEL_TOOLS=OFF ..` to temporarily switch off the formatting. Please switch it on again when committing normally.

Chapter 3

# Testing Instructions

# Chapter 4

# Benchmarking.

### # Benchmark example 1.

#### ## Poisson solver using Restricted Additive Schwarz with overlap.

The flag `-DSCHWARZ_BUILD_BENCHMARKING` (default `ON`) enables the example and benchmarking snippets. The following command line options are available for this example. This is setup using `gflags`.

The executable is run in the following fashion:

“‘sh [MPI_COMMAND] [MPI_OPTIONS]

# Chapter 5

# Module Documentation

## 5.1 Communicate

A module dedicated to the Communication interface in schwarz-lib.

### Namespaces

- SchwarzWrappers::CommHelpers

  *The CommHelper namespace .*
- ProcessTopology

  *The ProcessTopology namespace .*

### Classes

- class SchwarzWrappers::Communicate< ValueType, IndexType >

  *The communication class that provides the methods for the communication between the subdomains.*
- struct SchwarzWrappers::Metadata< ValueType, IndexType >

  *The solver metadata struct.*

### 5.1.1 Detailed Description

A module dedicated to the Communication interface in schwarz-lib.

## 5.2 Initialization

A module dedicated to the initialization and setup and usage of the solvers in schwarz-lib.

### Namespaces

- SchwarzWrappers::PartitionTools

  *The PartitionTools namespace .*
- ProcessTopology

  *The ProcessTopology namespace .*

### Classes

- class SchwarzWrappers::device_guard

  *This class defines a device guard for the cuda functions and the cuda module.*
- class SchwarzWrappers::Initialize< ValueType, IndexType >

  *The initialization class that provides methods for initialization of the solver.*
- struct SchwarzWrappers::Settings

  *The struct that contains the solver settings and the parameters to be set by the user.*
- struct SchwarzWrappers::Metadata< ValueType, IndexType >

  *The solver metadata struct.*

### 5.2.1 Detailed Description

A module dedicated to the initialization and setup and usage of the solvers in schwarz-lib.

## 5.3 Schwarz Class

A module dedicated to the Schwarz solver classes in schwarz-lib.

### Classes

- class SchwarzWrappers::SolverRAS< ValueType, IndexType >

  *An implementation of the solver interface using the RAS solver.*
- class SchwarzWrappers::SchwarzBase< ValueType, IndexType >

  *The Base solver class is meant to be the class implementing the common implementations for all the schwarz methods.*

### 5.3.1 Detailed Description

A module dedicated to the Schwarz solver classes in schwarz-lib.

## 5.4 Solve

A module dedicated to the solvers including local solution and convergence detection in schwarz-lib.

### Namespaces

- SchwarzWrappers::ConvergenceTools

  *The ConvergenceTools namespace .*
- SchwarzWrappers::SolverTools

  *The SolverTools namespace .*

### Classes

- struct SchwarzWrappers::Metadata< ValueType, IndexType >

  *The solver metadata struct.*
- class SchwarzWrappers::Solve< ValueType, IndexType >

  *The Solver class the provides the solver and the convergence checking methods.*

### 5.4.1 Detailed Description

A module dedicated to the solvers including local solution and convergence detection in schwarz-lib.

## 5.5 Utils

A module dedicated to the utilities in schwarz-lib.

### Classes

- struct SchwarzWrappers::Utils< ValueType, IndexType >

  *The utilities class which provides some checks and basic utilities.*

### 5.5.1 Detailed Description

A module dedicated to the utilities in schwarz-lib.

# Chapter 6

# Namespace Documentation

## 6.1 ProcessTopology Namespace Reference

The ProcessTopology namespace .

### 6.1.1 Detailed Description

The ProcessTopology namespace .

proc_topo

## 6.2 SchwarzWrappers Namespace Reference

The Schwarz wrappers namespace.

### Namespaces

- CommHelpers

  *The CommHelper namespace .*
- ConvergenceTools

  *The ConvergenceTools namespace .*
- PartitionTools

  *The PartitionTools namespace .*
- SolverTools

  *The SolverTools namespace .*

### Classes

- class Communicate

  *The communication class that provides the methods for the communication between the subdomains.*
- class device_guard

  *This class defines a device guard for the cuda functions and the cuda module.*
- class Initialize

  *The initialization class that provides methods for initialization of the solver.*
- struct Metadata

  *The solver metadata struct.*
- class SchwarzBase

  *The Base solver class is meant to be the class implementing the common implementations for all the schwarz methods.*
- struct Settings

  *The struct that contains the solver settings and the parameters to be set by the user.*
- class Solve

  *The Solver class the provides the solver and the convergence checking methods.*
- class SolverRAS

  *An implementation of the solver interface using the RAS solver.*
- struct Utils

  *The utilities class which provides some checks and basic utilities.*

#### 6.2.1 Detailed Description

The Schwarz wrappers namespace.

## 6.3 SchwarzWrappers::CommHelpers Namespace Reference

The CommHelper namespace .

#### 6.3.1 Detailed Description

The CommHelper namespace .

comm_helpers

## 6.4 SchwarzWrappers::ConvergenceTools Namespace Reference

The ConvergenceTools namespace .

#### 6.4.1 Detailed Description

The ConvergenceTools namespace .

conv_tools

## 6.5 SchwarzWrappers::PartitionTools Namespace Reference

The PartitionTools namespace .

### 6.5.1 Detailed Description

The PartitionTools namespace .

part_tools

## 6.6 SchwarzWrappers::SolverTools Namespace Reference

The SolverTools namespace .

### 6.6.1 Detailed Description

The SolverTools namespace .

solver_tools

# Chapter 7

# Class Documentation

## 7.1 BadDimension Class Reference

BadDimension is thrown if an operation is being applied to a LinOp with bad dimensions.

`#include <exception.hpp>`

### Public Member Functions

- BadDimension (const std::string &file, int line, const std::string &func, const std::string &op_name, std::size_t op_num_rows, std::size_t op_num_cols, const std::string &clarification)

  *Initializes a bad dimension error.*

### 7.1.1 Detailed Description

BadDimension is thrown if an operation is being applied to a LinOp with bad dimensions.

### 7.1.2 Constructor & Destructor Documentation

#### 7.1.2.1 BadDimension()

```
BadDimension::BadDimension (
            const std::string & file,
            int line,
            const std::string & func,
            const std::string & op_name,
            std::size_t op_num_rows,
            std::size_t op_num_cols,
            const std::string & clarification )  [inline]
```

Initializes a bad dimension error.

**Parameters**

| | |
|---|---|
| *file* | The name of the offending source file |
| *line* | The source code line number where the error occurred |
| *func* | The function name where the error occurred |
| *op_name* | The name of the operator |
| *op_num_rows* | The row dimension of the operator |
| *op_num_cols* | The column dimension of the operator |
| *clarification* | An additional message further describing the error |

```
115         : Error(file, line,
116                 func + ": Object " + op_name + " has dimensions [" +
117                     std::to_string(op_num_rows) + " x " +
118                     std::to_string(op_num_cols) + "]: " + clarification)
119     {}
```

The documentation for this class was generated from the following file:

- exception.hpp (4da3021)

## 7.2 SchwarzWrappers::Settings::comm_settings Struct Reference

The settings for the various available communication paradigms.

`#include <settings.hpp>`

### Public Attributes

- bool enable_onesided = false

  *Enable one-sided communication.*
- bool enable_overlap = false

  *Enable explicit overlap between communication and computation.*
- bool enable_put = false

  *Put the data to the window using MPI_Put rather than get.*
- bool enable_get = true

  *Get the data to the window using MPI_Get rather than put.*
- bool enable_one_by_one = false

  *Push each element separately directly into the buffer.*
- bool enable_flush_local = false

  *Use local flush.*
- bool enable_flush_all = true

  *Use flush all.*
- bool enable_lock_local = false

  *Use local locks.*
- bool enable_lock_all = true

  *Use lock all.*

### 7.2.1 Detailed Description

The settings for the various available communication paradigms.

The documentation for this struct was generated from the following file:

- settings.hpp (4da3021)

## 7.3 SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct Struct Reference

The communication struct used to store the communication data.

`#include <communicate.hpp>`

### Public Attributes

- int num_neighbors_in
  *The number of neighbors this subdomain has to receive data from.*
- int num_neighbors_out
  *The number of neighbors this subdomain has to send data to.*
- std::shared_ptr< gko::Array< IndexType > > neighbors_in
  *The neighbors this subdomain has to receive data from.*
- std::shared_ptr< gko::Array< IndexType > > neighbors_out
  *The neighbors this subdomain has to send data to.*
- std::vector< bool > is_local_neighbor
  *The bool vector which is true if the neighbors of a subdomain are in one node.*
- int local_num_neighbors_in
  *The number of neighbors this subdomain has to receive data from.*
- int local_num_neighbors_out
  *The number of neighbors this subdomain has to send data to.*
- std::shared_ptr< gko::Array< IndexType > > local_neighbors_in
  *The neighbors this subdomain has to receive data from.*
- std::shared_ptr< gko::Array< IndexType > > local_neighbors_out
  *The neighbors this subdomain has to send data to.*
- std::shared_ptr< gko::Array< IndexType * > > global_put
  *The array containing the number of elements that each subdomain sends from the other.*
- std::shared_ptr< gko::Array< IndexType * > > local_put
  *The array containing the number of elements that each subdomain sends from the other.*
- std::shared_ptr< gko::Array< IndexType * > > remote_put
  *The array containing the number of elements that each subdomain sends from the other.*
- std::shared_ptr< gko::Array< IndexType * > > global_get
  *The array containing the number of elements that each subdomain gets from the other.*
- std::shared_ptr< gko::Array< IndexType * > > local_get
  *The array containing the number of elements that each subdomain gets from the other.*
- std::shared_ptr< gko::Array< IndexType * > > remote_get
  *The array containing the number of elements that each subdomain gets from the other.*
- std::shared_ptr< gko::Array< IndexType > > window_ids

*The RDMA window ids.*

- std::shared_ptr< gko::Array< IndexType > > windows_from

  *The RDMA window ids to receive data from.*
- std::shared_ptr< gko::Array< IndexType > > windows_to

  *The RDMA window ids to send data to.*
- std::shared_ptr< gko::Array< MPI_Request > > put_request

  *The put request array.*
- std::shared_ptr< gko::Array< MPI_Request > > get_request

  *The get request array.*
- std::shared_ptr< gko::matrix::Dense< ValueType > > send_buffer

  *The send buffer used for the actual communication for both one-sided and two-sided.*
- std::shared_ptr< gko::matrix::Dense< ValueType > > recv_buffer

  *The recv buffer used for the actual communication for both one-sided and two-sided.*
- std::shared_ptr< gko::Array< IndexType > > get_displacements

  *The displacements for the receiving of the buffer.*
- std::shared_ptr< gko::Array< IndexType > > put_displacements

  *The displacements for the sending of the buffer.*
- MPI_Win window_recv_buffer

  *The RDMA window for the recv buffer.*
- MPI_Win window_send_buffer

  *The RDMA window for the send buffer.*
- MPI_Win window_x

  *The RDMA window for the solution vector.*

### 7.3.1 Detailed Description

**template<typename ValueType, typename IndexType>**
**struct SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct**

The communication struct used to store the communication data.

### 7.3.2 Member Data Documentation

#### 7.3.2.1 global_get

```
template<typename ValueType , typename IndexType >
std::shared_ptr<gko::Array<IndexType *> > SchwarzWrappers::Communicate< ValueType, IndexType
>::comm_struct::global_get
```

The array containing the number of elements that each subdomain gets from the other.

For example. global_get[p][0] contains the overall number of elements to be received to subdomain p and global↩ _put[p][i] contains the index of the solution vector to be received from subdomain p.

Referenced by SchwarzWrappers::SchwarzBase< ValueType, IndexType >::initialize(), SchwarzWrappers::↩ SchwarzBase< ValueType, IndexType >::SchwarzBase(), SchwarzWrappers::SolverRAS< ValueType, IndexType >::setup_comm_buffers(), and SchwarzWrappers::SolverRAS< ValueType, IndexType >::setup_windows().

#### 7.3.2.2 global_put

```
template<typename ValueType , typename IndexType >
std::shared_ptr<gko::Array<IndexType *> > SchwarzWrappers::Communicate< ValueType, IndexType
>::comm_struct::global_put
```

The array containing the number of elements that each subdomain sends from the other.

For example. global_put[p][0] contains the overall number of elements to be sent to subdomain p and global_put[p][i] contains the index of the solution vector to be sent to subdomain p.

Referenced by SchwarzWrappers::SchwarzBase< ValueType, IndexType >::initialize(), SchwarzWrappers::↩
SchwarzBase< ValueType, IndexType >::SchwarzBase(), SchwarzWrappers::SolverRAS< ValueType, IndexType
>::setup_comm_buffers(), and SchwarzWrappers::SolverRAS< ValueType, IndexType >::setup_windows().

#### 7.3.2.3 is_local_neighbor

```
template<typename ValueType , typename IndexType >
std::vector<bool> SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::is_↩
local_neighbor
```

The bool vector which is true if the neighbors of a subdomain are in one node.

Referenced by SchwarzWrappers::SchwarzBase< ValueType, IndexType >::SchwarzBase(), SchwarzWrappers↩
::SolverRAS< ValueType, IndexType >::setup_comm_buffers(), and SchwarzWrappers::SolverRAS< ValueType,
IndexType >::setup_windows().

#### 7.3.2.4 local_get

```
template<typename ValueType , typename IndexType >
std::shared_ptr<gko::Array<IndexType *> > SchwarzWrappers::Communicate< ValueType, IndexType
>::comm_struct::local_get
```

The array containing the number of elements that each subdomain gets from the other.

For example. global_get[p][0] contains the overall number of elements to be received to subdomain p and global↩
_put[p][i] contains the index of the solution vector to be received from subdomain p.

Referenced by SchwarzWrappers::SolverRAS< ValueType, IndexType >::setup_comm_buffers(), and Schwarz↩
Wrappers::SolverRAS< ValueType, IndexType >::setup_windows().

#### 7.3.2.5 local_put

```
template<typename ValueType , typename IndexType >
std::shared_ptr<gko::Array<IndexType *> > SchwarzWrappers::Communicate< ValueType, IndexType
>::comm_struct::local_put
```

The array containing the number of elements that each subdomain sends from the other.

For example. global_put[p][0] contains the overall number of elements to be sent to subdomain p and global_put[p][i] contains the index of the solution vector to be sent to subdomain p.

Referenced by SchwarzWrappers::SolverRAS< ValueType, IndexType >::setup_comm_buffers(), and SchwarzWrappers::SolverRAS< ValueType, IndexType >::setup_windows().

#### 7.3.2.6 remote_get

```
template<typename ValueType , typename IndexType >
std::shared_ptr<gko::Array<IndexType *> > SchwarzWrappers::Communicate< ValueType, IndexType
>::comm_struct::remote_get
```

The array containing the number of elements that each subdomain gets from the other.

For example. global_get[p][0] contains the overall number of elements to be received to subdomain p and global_put[p][i] contains the index of the solution vector to be received from subdomain p.

Referenced by SchwarzWrappers::SolverRAS< ValueType, IndexType >::setup_comm_buffers(), and SchwarzWrappers::SolverRAS< ValueType, IndexType >::setup_windows().

#### 7.3.2.7 remote_put

```
template<typename ValueType , typename IndexType >
std::shared_ptr<gko::Array<IndexType *> > SchwarzWrappers::Communicate< ValueType, IndexType
>::comm_struct::remote_put
```

The array containing the number of elements that each subdomain sends from the other.

For example. global_put[p][0] contains the overall number of elements to be sent to subdomain p and global_put[p][i] contains the index of the solution vector to be sent to subdomain p.

Referenced by SchwarzWrappers::SolverRAS< ValueType, IndexType >::setup_comm_buffers(), and SchwarzWrappers::SolverRAS< ValueType, IndexType >::setup_windows().

The documentation for this struct was generated from the following file:

- communicate.hpp (4da3021)

## 7.4 SchwarzWrappers::Communicate< ValueType, IndexType > Class Template Reference

The communication class that provides the methods for the communication between the subdomains.

`#include <communicate.hpp>`

### Classes

- struct comm_struct

  *The communication struct used to store the communication data.*

### Public Member Functions

- virtual void setup_comm_buffers ()=0

  *Sets up the communication buffers needed for the boundary exchange.*
- virtual void setup_windows (const Settings &settings, const Metadata< ValueType, IndexType > &metadata, std::shared_ptr< gko::matrix::Dense< ValueType >> &main_buffer)=0

  *Sets up the windows needed for the asynchronous communication.*
- virtual void exchange_boundary (const Settings &settings, const Metadata< ValueType, IndexType > &metadata, std::shared_ptr< gko::matrix::Dense< ValueType >> &solution_vector)=0

  *Exchanges the elements of the solution vector.*
- void local_to_global_vector (const Settings &settings, const Metadata< ValueType, IndexType > &metadata, const std::shared_ptr< gko::matrix::Dense< ValueType >> &local_vector, std::shared_ptr< gko::matrix::↩ Dense< ValueType >> &global_vector)

  *Transforms data from a local vector to a global vector.*
- virtual void update_boundary (const Settings &settings, const Metadata< ValueType, IndexType > &metadata, std::shared_ptr< gko::matrix::Dense< ValueType >> &local_solution, const std::shared_ptr< gko↩ ::matrix::Dense< ValueType >> &local_rhs, const std::shared_ptr< gko::matrix::Dense< ValueType >> &solution_vector, std::shared_ptr< gko::matrix::Dense< ValueType >> &global_old_solution, const std↩ ::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> &interface_matrix)=0

  *Update the values into local vector from obtained from the neighboring sub-domains using the interface matrix.*
- void clear (Settings &settings)

  *Clears the data.*

### 7.4.1 Detailed Description

**template<typename ValueType, typename IndexType>**
**class SchwarzWrappers::Communicate< ValueType, IndexType >**

The communication class that provides the methods for the communication between the subdomains.

**Template Parameters**

| | |
|---|---|
| *ValueType* | The type of the floating point values. |
| *IndexType* | The type of the index type values. |

Communicate

### 7.4.2 Member Function Documentation

#### 7.4.2.1 exchange_boundary()

```
template<typename ValueType , typename IndexType >
void SchwarzWrappers::Communicate< ValueType, IndexType >::exchange_boundary (
            const Settings & settings,
            const Metadata< ValueType, IndexType > & metadata,
            std::shared_ptr< gko::matrix::Dense< ValueType >> & solution_vector )  [pure
virtual]
```

Exchanges the elements of the solution vector.

**Parameters**

| | |
|---|---|
| *settings* | The settings struct. |
| *metadata* | The metadata struct. |
| *solution_vector* | The solution vector being exchanged between the subdomains. |

Implemented in SchwarzWrappers::SolverRAS< ValueType, IndexType >.

Referenced by SchwarzWrappers::SchwarzBase< ValueType, IndexType >::run().

#### 7.4.2.2 local_to_global_vector()

```
template<typename ValueType , typename IndexType >
void SchwarzWrappers::Communicate< ValueType, IndexType >::local_to_global_vector (
            const Settings & settings,
            const Metadata< ValueType, IndexType > & metadata,
            const std::shared_ptr< gko::matrix::Dense< ValueType >> & local_vector,
            std::shared_ptr< gko::matrix::Dense< ValueType >> & global_vector )
```

Transforms data from a local vector to a global vector.

**Parameters**

| | |
|---|---|
| *settings* | The settings struct. |
| *metadata* | The metadata struct. |
| *local_vector* | The local vector in question. |
| *global_vector* | The global vector in question. |

```
70 {
71     using vec = gko::matrix::Dense<ValueType>;
72     auto alpha = gko::initialize<gko::matrix::Dense<ValueType>>(
73         {1.0}, settings.executor);
74     auto temp_vector = vec::create(
75         settings.executor, gko::dim<2>(metadata.local_size, 1),
```

```
76          (gko::Array<ValueType>::view(
77              settings.executor, metadata.local_size,
78              &global_vector->get_values()[metadata.first_row
79                                           ->get_data()[metadata.my_rank]])),
80          1);
81
82      auto temp_vector2 = vec::create(
83          settings.executor, gko::dim<2>(metadata.local_size, 1),
84          (gko::Array<ValueType>::view(settings.executor, metadata.local_size,
85                                       &local_vector->get_values()[0])),
86          1);
87      if (settings.convergence_settings.convergence_crit ==
88          Settings::convergence_settings::local_convergence_crit::
89              residual_based) {
90          local_vector->add_scaled(alpha.get(), temp_vector.get());
91          temp_vector->add_scaled(alpha.get(), local_vector.get());
92      } else {
93          // TODO GPU: DONE
94          temp_vector->copy_from(temp_vector2.get());
95      }
96  }
```

#### 7.4.2.3 setup_windows()

```
template<typename ValueType , typename IndexType >
void SchwarzWrappers::Communicate< ValueType, IndexType >::setup_windows (
             const Settings & settings,
             const Metadata< ValueType, IndexType > & metadata,
             std::shared_ptr< gko::matrix::Dense< ValueType >> & main_buffer )  [pure virtual]
```

Sets up the windows needed for the asynchronous communication.

**Parameters**

| | |
|---|---|
| *settings* | The settings struct. |
| *metadata* | The metadata struct. |
| *main_buffer* | The main buffer being exchanged between the subdomains. |

Implemented in SchwarzWrappers::SolverRAS< ValueType, IndexType >.

Referenced by SchwarzWrappers::SchwarzBase< ValueType, IndexType >::run().

#### 7.4.2.4 update_boundary()

```
template<typename ValueType , typename IndexType >
void SchwarzWrappers::Communicate< ValueType, IndexType >::update_boundary (
             const Settings & settings,
             const Metadata< ValueType, IndexType > & metadata,
             std::shared_ptr< gko::matrix::Dense< ValueType >> & local_solution,
             const std::shared_ptr< gko::matrix::Dense< ValueType >> & local_rhs,
             const std::shared_ptr< gko::matrix::Dense< ValueType >> & solution_vector,
             std::shared_ptr< gko::matrix::Dense< ValueType >> & global_old_solution,
             const std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> & interface_↩
matrix )  [pure virtual]
```

Update the values into local vector from obtained from the neighboring sub-domains using the interface matrix.

**Parameters**

| | |
|---|---|
| *settings* | The settings struct. |
| *metadata* | The metadata struct. |
| *local_solution* | The local solution vector in the subdomain. |
| *local_rhs* | The local right hand side vector in the subdomain. |
| *solution_vector* | The workspace solution vector. |
| *global_old_solution* | The global solution vector of the previous iteration. |
| *interface_matrix* | The interface matrix containing the interface and the overlap data mainly used for exchanging values between different sub-domains. |

Implemented in SchwarzWrappers::SolverRAS< ValueType, IndexType >.

Referenced by SchwarzWrappers::SchwarzBase< ValueType, IndexType >::run().

The documentation for this class was generated from the following files:

- communicate.hpp (4da3021)
- /home/runner/work/schwarz-lib/schwarz-lib/source/communicate.cpp (4da3021)

## 7.5 SchwarzWrappers::Settings::convergence_settings Struct Reference

The various convergence settings available.

`#include <settings.hpp>`

### 7.5.1 Detailed Description

The various convergence settings available.

The documentation for this struct was generated from the following file:

- settings.hpp (4da3021)

## 7.6 CudaError Class Reference

CudaError is thrown when a CUDA routine throws a non-zero error code.

`#include <exception.hpp>`

### Public Member Functions

- CudaError (const std::string &file, int line, const std::string &func, int error_code)

  *Initializes a CUDA error.*

### 7.6.1 Detailed Description

CudaError is thrown when a CUDA routine throws a non-zero error code.

### 7.6.2 Constructor & Destructor Documentation

#### 7.6.2.1 CudaError()

```
CudaError::CudaError (
            const std::string & file,
            int line,
            const std::string & func,
            int error_code )  [inline]
```

Initializes a CUDA error.

**Parameters**

| | |
|---|---|
| *file* | The name of the offending source file |
| *line* | The source code line number where the error occurred |
| *func* | The name of the CUDA routine that failed |
| *error_code* | The resulting CUDA error code |

```
137         : Error(file, line, func + ": " + get_error(error_code))
138     {}
```

The documentation for this class was generated from the following files:

- exception.hpp (4da3021)
- /home/runner/work/schwarz-lib/schwarz-lib/source/exception.cpp (4da3021)

## 7.7 CusparseError Class Reference

CusparseError is thrown when a cuSPARSE routine throws a non-zero error code.

```
#include <exception.hpp>
```

### Public Member Functions

- CusparseError (const std::string &file, int line, const std::string &func, int error_code)

  *Initializes a cuSPARSE error.*

### 7.7.1 Detailed Description

CusparseError is thrown when a cuSPARSE routine throws a non-zero error code.

### 7.7.2 Constructor & Destructor Documentation

#### 7.7.2.1 CusparseError()

```
CusparseError::CusparseError (
            const std::string & file,
            int line,
            const std::string & func,
            int error_code )  [inline]
```

Initializes a cuSPARSE error.

**Parameters**

| | |
|---|---|
| *file* | The name of the offending source file |
| *line* | The source code line number where the error occurred |
| *func* | The name of the cuSPARSE routine that failed |
| *error_code* | The resulting cuSPARSE error code |

```
159         : Error(file, line, func + ": " + get_error(error_code))
160     {}
```

The documentation for this class was generated from the following files:

- exception.hpp (4da3021)
- /home/runner/work/schwarz-lib/schwarz-lib/source/exception.cpp (4da3021)

## 7.8 SchwarzWrappers::device_guard Class Reference

This class defines a device guard for the cuda functions and the cuda module.

`#include <device_guard.hpp>`

### 7.8.1 Detailed Description

This class defines a device guard for the cuda functions and the cuda module.

The guard is used to make sure that the device code is run on the correct cuda device, when run with multiple devices. The class records the current device id and uses `cudaSetDevice` to set the device id to the one being passed in. After the scope has been exited, the destructor sets the device_id back to the one before entering the scope.

The documentation for this class was generated from the following file:

- device_guard.hpp (4da3021)

## 7.9 SchwarzWrappers::Initialize< ValueType, IndexType > Class Template Reference

The initialization class that provides methods for initialization of the solver.

`#include <initialization.hpp>`

### Public Member Functions

- void generate_rhs (std::vector< ValueType > &rhs)

  *Generates the right hand side vector.*
- void setup_global_matrix_laplacian (const gko::size_type &oned_laplacian_size, std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> &global_matrix)

  *Generates the 2D global laplacian matrix.*
- void partition (const Settings &settings, const Metadata< ValueType, IndexType > &metadata, const std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> &global_matrix, std::vector< unsigned int > &partition_indices)

  *The partitioning function.*
- void setup_vectors (const Settings &settings, const Metadata< ValueType, IndexType > &metadata, std::vector< ValueType > &rhs, std::shared_ptr< gko::matrix::Dense< ValueType >> &local_rhs, std::shared_ptr< gko::matrix::Dense< ValueType >> &global_rhs, std::shared_ptr< gko::matrix::Dense< ValueType >> &local_solution, std::shared_ptr< gko::matrix::Dense< ValueType >> &global_solution)

  *Setup the vectors with default values and allocate mameory if not allocated.*
- virtual void setup_local_matrices (Settings &settings, Metadata< ValueType, IndexType > &metadata, std::vector< unsigned int > &partition_indices, std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> &global_matrix, std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> &local_matrix, std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> &interface_matrix)=0

  *Sets up the local and the interface matrices from the global matrix and the partition indices.*

### Public Attributes

- std::vector< unsigned int > partition_indices

  *The partition indices containing the subdomains to which each row(vertex) of the matrix(graph) belongs to.*
- std::vector< unsigned int > cell_weights

  *The cell weights for the partition algorithm.*

### Additional Inherited Members

### 7.9.1 Detailed Description

**template<typename ValueType = gko::default_precision, typename IndexType = gko::int32>**
**class SchwarzWrappers::Initialize< ValueType, IndexType >**

The initialization class that provides methods for initialization of the solver.

**Template Parameters**

| | |
|---|---|
| *ValueType* | The type of the floating point values. |
| *IndexType* | The type of the index type values. |

Initialization

### 7.9.2 Member Function Documentation

#### 7.9.2.1 generate_rhs()

```
template<typename ValueType , typename IndexType >
void SchwarzWrappers::Initialize< ValueType, IndexType >::generate_rhs (
            std::vector< ValueType > & rhs )
```

Generates the right hand side vector.

**Parameters**

| | |
|---|---|
| *rhs* | The rhs vector. |

Referenced by SchwarzWrappers::SchwarzBase< ValueType, IndexType >::initialize().

```
81 {
82     std::uniform_real_distribution<double> unif(0.0, 1.0);
83     std::default_random_engine engine;
84     for (gko::size_type i = 0; i < rhs.size(); ++i) {
85         rhs[i] = unif(engine);
86     }
87 }
```

#### 7.9.2.2 partition()

```
template<typename ValueType , typename IndexType >
void SchwarzWrappers::Initialize< ValueType, IndexType >::partition (
            const Settings & settings,
            const Metadata< ValueType, IndexType > & metadata,
            const std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> & global_↩
matrix,
            std::vector< unsigned int > & partition_indices )
```

The partitioning function.

Allows the partition of the global matrix depending with METIS and a regular 1D decomposition.

**Parameters**

| | |
|---|---|
| *settings* | The settings struct. |
| *metadata* | The metadata struct. |
| *global_matrix* | The global matrix. |
| *partition_indices* | The partition indices [OUTPUT]. |

References SchwarzWrappers::Metadata< ValueType, IndexType >::global_size, SchwarzWrappers::Metadata< ValueType, IndexType >::my_rank, SchwarzWrappers::Metadata< ValueType, IndexType >::num_subdomains, and SchwarzWrappers::Settings::write_debug_out.

Referenced by SchwarzWrappers::SchwarzBase< ValueType, IndexType >::initialize().

```
265 {
266     partition_indices.resize(metadata.global_size);
267     if (metadata.my_rank == 0) {
268         auto partition_settings =
269             (Settings::partition_settings::partition_zoltan |
270              Settings::partition_settings::partition_metis |
271              Settings::partition_settings::partition_regular |
272              Settings::partition_settings::partition_regular2d |
273              Settings::partition_settings::partition_custom) &
274             settings.partition;
275
276         if (partition_settings ==
277             Settings::partition_settings::partition_zoltan) {
278             SCHWARZ_NOT_IMPLEMENTED;
279         } else if (partition_settings ==
280                    Settings::partition_settings::partition_metis) {
281             if (metadata.my_rank == 0) {
282                 std::cout << " METIS partition" << std::endl;
283             }
284             PartitionTools::PartitionMetis(
285                 settings, global_matrix, this->cell_weights,
286                 metadata.num_subdomains, partition_indices);
287         } else if (partition_settings ==
288                    Settings::partition_settings::partition_regular) {
289             if (metadata.my_rank == 0) {
290                 std::cout << " Regular 1D partition" << std::endl;
291             }
292             PartitionTools::PartitionRegular(
293                 global_matrix, metadata.num_subdomains, partition_indices);
294         } else if (partition_settings ==
295                    Settings::partition_settings::partition_regular2d) {
296             if (metadata.my_rank == 0) {
297                 std::cout << " Regular 2D partition" << std::endl;
298             }
299             PartitionTools::PartitionRegular2D(
300                 global_matrix, settings.write_debug_out,
301                 metadata.num_subdomains, partition_indices);
302         } else if (partition_settings ==
303                    Settings::partition_settings::partition_custom) {
304             // User partitions mesh manually
305             SCHWARZ_NOT_IMPLEMENTED;
306         } else {
307             SCHWARZ_NOT_IMPLEMENTED;
308         }
309     }
310 }
```

#### 7.9.2.3 setup_global_matrix_laplacian()

```
template<typename ValueType , typename IndexType >
void SchwarzWrappers::Initialize< ValueType, IndexType >::setup_global_matrix_laplacian (
             const gko::size_type & oned_laplacian_size,
             std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> & global_matrix )
```

Generates the 2D global laplacian matrix.

**Parameters**

| | |
|---|---|
| *oned_laplacian_size* | The size of the one d laplacian grid. |
| *global_matrix* | The global matrix. |

Referenced by SchwarzWrappers::SchwarzBase< ValueType, IndexType >::initialize().

```
203 {
204     using index_type = IndexType;
205     using value_type = ValueType;
206     using mtx = gko::matrix::Csr<value_type, index_type>;
207     gko::size_type global_size = oned_laplacian_size *
    oned_laplacian_size;
208
209     global_matrix = mtx::create(settings.executor->get_master(),
210                                 gko::dim<2>(global_size), 5 * global_size);
211     value_type *values = global_matrix->get_values();
212     index_type *row_ptrs = global_matrix->get_row_ptrs();
213     index_type *col_idxs = global_matrix->get_col_idxs();
214
215     std::vector<gko::size_type> exclusion_set;
216
217     std::map<IndexType, ValueType> stencil_map = {
218         {-oned_laplacian_size, -1}, {-1, -1}, {0, 4}, {1, -1},
219         {oned_laplacian_size, -1},
220     };
221     for (auto i = 2; i < global_size; ++i) {
222         gko::size_type index = (i - 1) * oned_laplacian_size;
223         if (index * index < global_size * global_size) {
224             exclusion_set.push_back(
225                 linearize_index(index, index - 1, global_size));
226             exclusion_set.push_back(
227                 linearize_index(index - 1, index, global_size));
228         }
229     }
230
231     std::sort(exclusion_set.begin(),
232               exclusion_set.begin() + exclusion_set.size());
233
234     IndexType pos = 0;
235     IndexType col_idx = 0;
236     row_ptrs[0] = pos;
237     gko::size_type cur_idx = 0;
238     for (IndexType i = 0; i < global_size; ++i) {
239         for (auto ofs : stencil_map) {
240             auto in_exclusion_flag =
241                 (exclusion_set[cur_idx] ==
242                  linearize_index(i, i + ofs.first, global_size));
243             if (0 <= i + ofs.first && i + ofs.first < global_size &&
244                 !in_exclusion_flag) {
245                 values[pos] = ofs.second;
246                 col_idxs[pos] = i + ofs.first;
247                 ++pos;
248             }
249             if (in_exclusion_flag) {
250                 cur_idx++;
251             }
252             col_idx = row_ptrs[i + 1] - pos;
253         }
254         row_ptrs[i + 1] = pos;
255     }
256 }
```

#### 7.9.2.4 setup_local_matrices()

```
template<typename ValueType , typename IndexType >
void SchwarzWrappers::Initialize< ValueType, IndexType >::setup_local_matrices (
             Settings & settings,
             Metadata< ValueType, IndexType > & metadata,
             std::vector< unsigned int > & partition_indices,
             std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> & global_matrix,
             std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> & local_matrix,
             std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> & interface_matrix )
[pure virtual]
```

Sets up the local and the interface matrices from the global matrix and the partition indices.

**Parameters**

| | |
|---|---|
| *settings* | The settings struct. |
| *metadata* | The metadata struct. |
| *partition_indices* | The array containing the partition indices. |
| *global_matrix* | The global system matrix. |
| *local_matrix* | The local system matrix. |
| *interface_matrix* | The interface matrix containing the interface and the overlap data mainly used for exchanging values between different sub-domains. |
| *local_perm* | The local permutation, obtained through RCM or METIS. |

Implemented in SchwarzWrappers::SolverRAS< ValueType, IndexType >.

Referenced by SchwarzWrappers::SchwarzBase< ValueType, IndexType >::initialize().

#### 7.9.2.5 setup_vectors()

```
template<typename ValueType , typename IndexType >
void SchwarzWrappers::Initialize< ValueType, IndexType >::setup_vectors (
             const Settings & settings,
             const Metadata< ValueType, IndexType > & metadata,
             std::vector< ValueType > & rhs,
             std::shared_ptr< gko::matrix::Dense< ValueType >> & local_rhs,
             std::shared_ptr< gko::matrix::Dense< ValueType >> & global_rhs,
             std::shared_ptr< gko::matrix::Dense< ValueType >> & local_solution,
             std::shared_ptr< gko::matrix::Dense< ValueType >> & global_solution )
```

Setup the vectors with default values and allocate mameory if not allocated.

**Parameters**

| | |
|---|---|
| *settings* | The settings struct. |
| *metadata* | The metadata struct. |
| *local_rhs* | The local right hand side vector in the subdomain. |
| *global_rhs* | The global right hand side vector. |
| *local_solution* | The local solution vector in the subdomain. |
| *global_solution* | The global solution vector. |

References SchwarzWrappers::Settings::executor, SchwarzWrappers::Metadata< ValueType, IndexType >::first_row, SchwarzWrappers::Metadata< ValueType, IndexType >::global_size, SchwarzWrappers::Metadata< ValueType, IndexType >::local_size_x, and SchwarzWrappers::Metadata< ValueType, IndexType >::my_rank.

Referenced by SchwarzWrappers::SchwarzBase< ValueType, IndexType >::initialize().

```
321 {
322     using vec = gko::matrix::Dense<ValueType>;
323     auto my_rank = metadata.my_rank;
324     auto first_row = metadata.first_row->get_data()[my_rank];
325
326     // Copy the global rhs vector to the required executor.
327     gko::Array<ValueType> temp_rhs{settings.executor->get_master(), rhs.begin(),
```

```
328                                     rhs.end()};
329     global_rhs = vec::create(settings.executor,
330                              gko::dim<2>{metadata.global_size, 1}, temp_rhs, 1);
331     global_solution = vec::create(settings.executor->get_master(),
332                                   gko::dim<2>(metadata.global_size, 1));
333
334     local_rhs =
335         vec::create(settings.executor, gko::dim<2>(metadata.local_size_x, 1));
336     // Extract the local rhs from the global rhs. Also takes into account the
337     // overlap.
338     SolverTools::extract_local_vector(settings, metadata, local_rhs, global_rhs,
339                                       first_row);
340
341     local_solution =
342         vec::create(settings.executor, gko::dim<2>(metadata.local_size_x, 1));
343 }
```

The documentation for this class was generated from the following files:

- initialization.hpp (4da3021)
- /home/runner/work/schwarz-lib/schwarz-lib/source/initialization.cpp (4da3021)

## 7.10 SchwarzWrappers::Metadata< ValueType, IndexType > Struct Template Reference

The solver metadata struct.

`#include <settings.hpp>`

### Public Attributes

- MPI_Comm mpi_communicator

  *The MPI communicator.*
- gko::size_type global_size = 0

  *The size of the global matrix.*
- gko::size_type oned_laplacian_size = 0

  *The size of the 1 dimensional laplacian grid.*
- gko::size_type local_size = 0

  *The size of the local subdomain matrix.*
- gko::size_type local_size_x = 0

  *The size of the local subdomain matrix + the overlap.*
- gko::size_type local_size_o = 0

  *The size of the local subdomain matrix + the overlap.*
- gko::size_type overlap_size = 0

  *The size of the overlap between the subdomains.*
- gko::size_type num_subdomains = 1

  *The number of subdomains used within the solver.*
- int my_rank

  *The rank of the subdomain.*
- int my_local_rank

  *The local rank of the subdomain.*
- int local_num_procs

  *The local number of procs in the subdomain.*
- int comm_size

  *The number of subdomains used within the solver, size of the communicator.*

- int num_threads

  *The number of threads used within the solver for each subdomain.*
- IndexType iter_count

  *The iteration count of the solver.*
- ValueType tolerance

  *The tolerance of the complete solver.*
- ValueType local_solver_tolerance

  *The tolerance of the local solver in case of an iterative solve.*
- IndexType max_iters

  *The maximum iteration count of the solver.*
- unsigned int precond_max_block_size

  *The maximum block size for the preconditioner.*
- ValueType current_residual_norm = -1.0

  *The current residual norm of the subdomain.*
- ValueType min_residual_norm = -1.0

  *The minimum residual norm of the subdomain.*
- std::vector< std::tuple< int, int, int, std::string, std::vector< ValueType > > > time_struct

  *The struct used to measure the timings of each function within the solver loop.*
- std::vector< std::tuple< int, std::vector< std::tuple< int, int > >, std::vector< std::tuple< int, int > >, int, int > > comm_data_struct

  *The struct used to measure the timings of each function within the solver loop.*
- std::shared_ptr< gko::Array< IndexType > > global_to_local

  *The mapping containing the global to local indices.*
- std::shared_ptr< gko::Array< IndexType > > local_to_global

  *The mapping containing the local to global indices.*
- std::shared_ptr< gko::Array< IndexType > > overlap_row

  *The overlap row indices.*
- std::shared_ptr< gko::Array< IndexType > > first_row

  *The starting row of each subdomain in the matrix.*
- std::shared_ptr< gko::Array< IndexType > > permutation

  *The permutation used for the re-ordering.*
- std::shared_ptr< gko::Array< IndexType > > i_permutation

  *The inverse permutation used for the re-ordering.*

### 7.10.1 Detailed Description

**template<typename ValueType, typename IndexType>**
**struct SchwarzWrappers::Metadata< ValueType, IndexType >**

The solver metadata struct.

**Template Parameters**

| | |
|---|---|
| *ValueType* | The type of the floating point values. |
| *IndexType* | The type of the index type values. |

### 7.10.2 Member Data Documentation

#### 7.10.2.1 local_solver_tolerance

```
template<typename ValueType, typename IndexType>
ValueType SchwarzWrappers::Metadata< ValueType, IndexType >::local_solver_tolerance
```

The tolerance of the local solver in case of an iterative solve.

The residual norm reduction required.

#### 7.10.2.2 tolerance

```
template<typename ValueType, typename IndexType>
ValueType SchwarzWrappers::Metadata< ValueType, IndexType >::tolerance
```

The tolerance of the complete solver.

The residual norm reduction required.

The documentation for this struct was generated from the following file:

- settings.hpp (4da3021)

## 7.11 MetisError Class Reference

MetisError is thrown when a METIS routine throws a non-zero error code.

```
#include <exception.hpp>
```

### Public Member Functions

- MetisError (const std::string &file, int line, const std::string &func, int error_code)
  *Initializes a METIS error.*

### 7.11.1 Detailed Description

MetisError is thrown when a METIS routine throws a non-zero error code.

### 7.11.2 Constructor & Destructor Documentation

#### 7.11.2.1 MetisError()

```
MetisError::MetisError (
            const std::string & file,
            int line,
            const std::string & func,
            int error_code )  [inline]
```

Initializes a METIS error.

**Parameters**

| | |
|---|---|
| *file* | The name of the offending source file |
| *line* | The source code line number where the error occurred |
| *func* | The name of the METIS routine that failed |
| *error_code* | The resulting METIS error code |

```
182         : Error(file, line, func + ": " + get_error(error_code))
183     {}
```

The documentation for this class was generated from the following files:

- exception.hpp (4da3021)
- /home/runner/work/schwarz-lib/schwarz-lib/source/exception.cpp (4da3021)

## 7.12 SchwarzWrappers::SchwarzBase< ValueType, IndexType > Class Template Reference

The Base solver class is meant to be the class implementing the common implementations for all the schwarz methods.

```
#include <schwarz_base.hpp>
```

### Public Member Functions

- SchwarzBase (Settings &settings, Metadata< ValueType, IndexType > &metadata)

  *The constructor that takes in the user settings and a metadata struct containing the solver metadata.*
- void initialize ()

  *Initialize the matrix and vectors.*
- void run (std::shared_ptr< gko::matrix::Dense< ValueType >> &solution)

  *The function that runs the actual solver and obtains the final solution.*
- void print_vector (const std::shared_ptr< gko::matrix::Dense< ValueType >> &vector, int subd, std::string name)

  *The auxiliary function that prints a passed in vector.*
- void print_matrix (const std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> &matrix, int rank, std::string name)

  *The auxiliary function that prints a passed in CSR matrix.*

### Public Attributes

- std::shared_ptr< gko::matrix::Csr< ValueType, IndexType > > local_matrix
  *The local subdomain matrix.*
- std::shared_ptr< gko::matrix::Permutation< IndexType > > local_row_perm
  *The local subdomain row permutation matrix/array.*
- std::shared_ptr< gko::matrix::Permutation< IndexType > > local_inv_row_perm
  *The local subdomain inverse row permutation matrix/array.*
- std::shared_ptr< gko::matrix::Permutation< IndexType > > local_col_perm
  *The local subdomain column permutation matrix/array.*
- std::shared_ptr< gko::matrix::Permutation< IndexType > > local_inv_col_perm
  *The local subdomain inverse column permutation matrix/array.*
- std::shared_ptr< gko::matrix::Csr< ValueType, IndexType > > triangular_factor_l
  *The local lower triangular factor used for the triangular solves.*
- std::shared_ptr< gko::matrix::Csr< ValueType, IndexType > > triangular_factor_u
  *The local upper triangular factor used for the triangular solves.*
- std::shared_ptr< gko::matrix::Csr< ValueType, IndexType > > interface_matrix
  *The local interface matrix.*
- std::shared_ptr< gko::matrix::Csr< ValueType, IndexType > > global_matrix
  *The global matrix.*
- std::shared_ptr< gko::matrix::Dense< ValueType > > local_rhs
  *The local right hand side.*
- std::shared_ptr< gko::matrix::Dense< ValueType > > global_rhs
  *The global right hand side.*
- std::shared_ptr< gko::matrix::Dense< ValueType > > local_solution
  *The local solution vector.*
- std::shared_ptr< gko::matrix::Dense< ValueType > > global_solution
  *The global solution vector.*

### Additional Inherited Members

#### 7.12.1 Detailed Description

**template<typename ValueType = gko::default_precision, typename IndexType = gko::int32>**
**class SchwarzWrappers::SchwarzBase< ValueType, IndexType >**

The Base solver class is meant to be the class implementing the common implementations for all the schwarz methods.

It derives from the Initialization class, the Communication class and the Solve class all of which are templated.

**Template Parameters**

| | |
|---|---|
| *ValueType* | The type of the floating point values. |
| *IndexType* | The type of the index type values. |

### 7.12.2 Constructor & Destructor Documentation

#### 7.12.2.1 SchwarzBase()

```
template<typename ValueType , typename IndexType >
SchwarzWrappers::SchwarzBase< ValueType, IndexType >::SchwarzBase (
            Settings & settings,
            Metadata< ValueType, IndexType > & metadata )
```

The constructor that takes in the user settings and a metadata struct containing the solver metadata.

**Parameters**

| | |
|---|---|
| *settings* | The settings struct. |
| *metadata* | The metadata struct. |

References SchwarzWrappers::Metadata< ValueType, IndexType >::comm_size, SchwarzWrappers::Settings::cuda_device_guard, SchwarzWrappers::Settings::executor, SchwarzWrappers::Settings::executor_string, SchwarzWrappers::Metadata< ValueType, IndexType >::first_row, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::get_displacements, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::global_get, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::global_put, SchwarzWrappers::Metadata< ValueType, IndexType >::global_size, SchwarzWrappers::Metadata< ValueType, IndexType >::global_to_local, SchwarzWrappers::Metadata< ValueType, IndexType >::i_permutation, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::is_local_neighbor, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::local_neighbors_in, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::local_neighbors_out, SchwarzWrappers::Metadata< ValueType, IndexType >::local_num_procs, SchwarzWrappers::Metadata< ValueType, IndexType >::local_to_global, SchwarzWrappers::Metadata< ValueType, IndexType >::mpi_communicator, SchwarzWrappers::Metadata< ValueType, IndexType >::my_local_rank, SchwarzWrappers::Metadata< ValueType, IndexType >::my_rank, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::neighbors_in, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::neighbors_out, SchwarzWrappers::Metadata< ValueType, IndexType >::num_subdomains, SchwarzWrappers::Metadata< ValueType, IndexType >::permutation, and SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::put_displacements.

```
50     : Initialize<ValueType, IndexType>(settings, metadata),
51       settings(settings),
52       metadata(metadata)
53 {
54     using vec_itype = gko::Array<IndexType>;
55     using vec_vecshared = gko::Array<IndexType *>;
56     metadata.my_local_rank =
57         Utils<ValueType, IndexType>::get_local_rank(metadata.mpi_communicator);
58     metadata.local_num_procs = Utils<ValueType, IndexType>::get_local_num_procs(
59         metadata.mpi_communicator);
60     auto my_local_rank = metadata.my_local_rank;
61     if (settings.executor_string == "omp") {
62         settings.executor = gko::OmpExecutor::create();
63         auto exec_info =
64             static_cast<gko::OmpExecutor *>(settings.executor.get())
65                 ->get_exec_info();
66         exec_info->bind_to_core(metadata.my_local_rank);
67
68     } else if (settings.executor_string == "cuda") {
69         int num_devices = 0;
70 #if SCHW_HAVE_CUDA
71         SCHWARZ_ASSERT_NO_CUDA_ERRORS(cudaGetDeviceCount(&num_devices));
72 #else
73         SCHWARZ_NOT_IMPLEMENTED;
74 #endif
75         if (num_devices > 0) {
```

```
76              if (metadata.my_rank == 0) {
77                  std::cout << " Number of available devices: " << num_devices
78                            << std::endl;
79              }
80          } else {
81              std::cout << " No CUDA devices available for rank "
82                        << metadata.my_rank << std::endl;
83              std::exit(-1);
84          }
85          settings.executor = gko::CudaExecutor::create(
86              my_local_rank, gko::OmpExecutor::create());
87          auto exec_info = static_cast<gko::OmpExecutor *>(
88                               settings.executor->get_master().get())
89                               ->get_exec_info();
90          exec_info->bind_to_core(my_local_rank);
91          settings.cuda_device_guard =
92              std::make_shared<SchwarzWrappers::device_guard>(my_local_rank);
93
94          std::cout << " Rank " << metadata.my_rank << " with local rank "
95                    << my_local_rank << " has "
96                    << (static_cast<gko::CudaExecutor *>(settings.executor.get()))
97                           ->get_device_id()
98                    << " id of gpu" << std::endl;
99          MPI_Barrier(metadata.mpi_communicator);
100     } else if (settings.executor_string == "reference") {
101         settings.executor = gko::ReferenceExecutor::create();
102         auto exec_info =
103             static_cast<gko::ReferenceExecutor *>(settings.executor.get())
104                 ->get_exec_info();
105         exec_info->bind_to_core(my_local_rank);
106     }
107
108     auto my_rank = this->metadata.my_rank;
109     auto comm_size = this->metadata.comm_size;
110     auto num_subdomains = this->metadata.num_subdomains;
111     auto global_size = this->metadata.global_size;
112
113     // Some arrays for partitioning and local matrix creation.
114     metadata.first_row = std::shared_ptr<vec_itype>(
115         new vec_itype(settings.executor->get_master(), num_subdomains + 1),
116         std::default_delete<vec_itype>());
117     metadata.permutation = std::shared_ptr<vec_itype>(
118         new vec_itype(settings.executor->get_master(), global_size),
119         std::default_delete<vec_itype>());
120     metadata.i_permutation = std::shared_ptr<vec_itype>(
121         new vec_itype(settings.executor->get_master(), global_size),
122         std::default_delete<vec_itype>());
123     metadata.global_to_local = std::shared_ptr<vec_itype>(
124         new vec_itype(settings.executor->get_master(), global_size),
125         std::default_delete<vec_itype>());
126     metadata.local_to_global = std::shared_ptr<vec_itype>(
127         new vec_itype(settings.executor->get_master(), global_size),
128         std::default_delete<vec_itype>());
129
130     // Some arrays for communication.
131     comm_struct.local_neighbors_in = std::shared_ptr<vec_itype>(
132         new vec_itype(settings.executor->get_master(), num_subdomains + 1),
133         std::default_delete<vec_itype>());
134     comm_struct.local_neighbors_out = std::shared_ptr<vec_itype>(
135         new vec_itype(settings.executor->get_master(), num_subdomains + 1),
136         std::default_delete<vec_itype>());
137     comm_struct.neighbors_in = std::shared_ptr<vec_itype>(
138         new vec_itype(settings.executor->get_master(), num_subdomains + 1),
139         std::default_delete<vec_itype>());
140     comm_struct.neighbors_out = std::shared_ptr<vec_itype>(
141         new vec_itype(settings.executor->get_master(), num_subdomains + 1),
142         std::default_delete<vec_itype>());
143     comm_struct.is_local_neighbor = std::vector<bool>(
    num_subdomains + 1, 0);
144     comm_struct.global_get = std::shared_ptr<vec_vecshared>(
145         new vec_vecshared(settings.executor->get_master(), num_subdomains + 1),
146         std::default_delete<vec_vecshared>());
147     comm_struct.global_put = std::shared_ptr<vec_vecshared>(
148         new vec_vecshared(settings.executor->get_master(), num_subdomains + 1),
149         std::default_delete<vec_vecshared>());
150     // Need this to initialize the arrays with zeros.
151     std::vector<IndexType> temp(num_subdomains + 1, 0);
152     comm_struct.get_displacements = std::shared_ptr<vec_itype>(
153         new vec_itype(settings.executor->get_master(), temp.begin(),
154                       temp.end()),
155         std::default_delete<vec_itype>());
156     comm_struct.put_displacements = std::shared_ptr<vec_itype>(
157         new vec_itype(settings.executor->get_master(), temp.begin(),
158                       temp.end()),
159         std::default_delete<vec_itype>());
160 }
```

### 7.12.3 Member Function Documentation

#### 7.12.3.1 print_matrix()

```
template<typename ValueType = gko::default_precision, typename IndexType = gko::int32>
void SchwarzWrappers::SchwarzBase< ValueType, IndexType >::print_matrix (
            const std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> & matrix,
            int rank,
            std::string name )
```

The auxiliary function that prints a passed in CSR matrix.

**Parameters**

| | |
|---|---|
| *matrix* | The matrix to be printed. |
| *subd* | The subdomain on which the vector exists. |
| *name* | The name of the matrix as a string. |

#### 7.12.3.2 print_vector()

```
template<typename ValueType = gko::default_precision, typename IndexType = gko::int32>
void SchwarzWrappers::SchwarzBase< ValueType, IndexType >::print_vector (
            const std::shared_ptr< gko::matrix::Dense< ValueType >> & vector,
            int subd,
            std::string name )
```

The auxiliary function that prints a passed in vector.

**Parameters**

| | |
|---|---|
| *vector* | The vector to be printed. |
| *subd* | The subdomain on which the vector exists. |
| *name* | The name of the vector as a string. |

#### 7.12.3.3 run()

```
template<typename ValueType , typename IndexType >
void SchwarzWrappers::SchwarzBase< ValueType, IndexType >::run (
            std::shared_ptr< gko::matrix::Dense< ValueType >> & solution )
```

The function that runs the actual solver and obtains the final solution.

**Parameters**

| | |
|---|---|
| *solution* | The solution vector. |

References SchwarzWrappers::Communicate< ValueType, IndexType >::exchange_boundary(), SchwarzWrappers::Settings::executor, SchwarzWrappers::SchwarzBase< ValueType, IndexType >::global_matrix, SchwarzWrappers::SchwarzBase< ValueType, IndexType >::global_rhs, SchwarzWrappers::SchwarzBase< ValueType, IndexType >::interface_matrix, SchwarzWrappers::SchwarzBase< ValueType, IndexType >::local_col_perm, SchwarzWrappers::SchwarzBase< ValueType, IndexType >::local_inv_col_perm, SchwarzWrappers::SchwarzBase< ValueType, IndexType >::local_inv_row_perm, SchwarzWrappers::SchwarzBase< ValueType, IndexType >::local_matrix, SchwarzWrappers::SchwarzBase< ValueType, IndexType >::local_rhs, SchwarzWrappers::SchwarzBase< ValueType, IndexType >::local_row_perm, SchwarzWrappers::SchwarzBase< ValueType, IndexType >::local_solution, SchwarzWrappers::Communicate< ValueType, IndexType >::setup_windows(), SchwarzWrappers::SchwarzBase< ValueType, IndexType >::triangular_factor_l, SchwarzWrappers::SchwarzBase< ValueType, IndexType >::triangular_factor_u, and SchwarzWrappers::Communicate< ValueType, IndexType >::update_boundary().

```
337 {
338     using vec_vtype = gko::matrix::Dense<ValueType>;
339     // The main solution vector
340     std::shared_ptr<vec_vtype> solution_vector = vec_vtype::create(
341         settings.executor, gko::dim<2>(metadata.global_size, 1));
342     // A temp local solution
343     std::shared_ptr<vec_vtype> init_guess =
344         vec_vtype::create(settings.executor, this->local_solution->get_size());
345     // A global gathered solution of the previous iteration.
346     std::shared_ptr<vec_vtype> global_old_solution = vec_vtype::create(
347         settings.executor, gko::dim<2>(metadata.global_size, 1));
348     // Setup the windows for the onesided communication.
349     this->setup_windows(settings, metadata, solution_vector);
350
351     const auto solver_settings =
352         (Settings::local_solver_settings::direct_solver_cholmod |
353          Settings::local_solver_settings::direct_solver_umfpack |
354          Settings::local_solver_settings::direct_solver_ginkgo |
355          Settings::local_solver_settings::iterative_solver_dealii |
356          Settings::local_solver_settings::iterative_solver_ginkgo) &
357         settings.local_solver;
358
359     ValueType local_residual_norm = -1.0, local_residual_norm0 = -1.0,
360               global_residual_norm = 0.0, global_residual_norm0 = -1.0;
361     metadata.iter_count = 0;
362     auto start_time = std::chrono::steady_clock::now();
363     int num_converged_procs = 0;
364
365     for (; metadata.iter_count < metadata.max_iters; ++(metadata.iter_count)) {
366         // Exchange the boundary values. The communication part.
367         MEASURE_ELAPSED_FUNC_TIME(
368             this->exchange_boundary(settings, metadata, solution_vector), 0,
369             metadata.my_rank, boundary_exchange, metadata.iter_count);
370
371         // Update the boundary and interior values after the exchanging from
372         // other processes.
373         MEASURE_ELAPSED_FUNC_TIME(
374             this->update_boundary(settings, metadata, this->
    local_solution,
375                                   this->local_rhs, solution_vector,
376                                   global_old_solution, this->interface_matrix),
377             1, metadata.my_rank, boundary_update, metadata.iter_count);
378
379         // Check for the convergence of the solver.
380         num_converged_procs = 0;
381         MEASURE_ELAPSED_FUNC_TIME(
382             (Solve<ValueType, IndexType>::check_convergence(
383                 settings, metadata, this->comm_struct, this->convergence_vector,
384                 global_old_solution, this->local_solution, this->
    local_matrix,
385                 local_residual_norm, local_residual_norm0, global_residual_norm,
386                 global_residual_norm0, num_converged_procs)),
387             2, metadata.my_rank, convergence_check, metadata.iter_count);
388
389         // break if the solution diverges.
390         if (std::isnan(global_residual_norm) || global_residual_norm > 1e12) {
391             std::cout << " Rank " << metadata.my_rank << " diverged in "
392                       << metadata.iter_count << " iters " << std::endl;
393             std::exit(-1);
```

```
394         }
395
396         // break if all processes detect that all other processes have
397         // converged otherwise continue iterations.
398         if (num_converged_procs == metadata.num_subdomains) {
399             break;
400         } else {
401             MEASURE_ELAPSED_FUNC_TIME(
402                 (Solve<ValueType, IndexType>::local_solve(
403                     settings, metadata, this->local_matrix,
404                     this->triangular_factor_l, this->
    triangular_factor_u,
405                     this->local_row_perm, this->local_inv_row_perm,
406                     this->local_col_perm, this->local_inv_col_perm,
    init_guess,
407                     this->local_solution)),
408                 3, metadata.my_rank, local_solve, metadata.iter_count);
409             // init_guess->copy_from(this->local_solution.get());
410             // Gather the local vector into the locally global vector for
411             // communication.
412             MEASURE_ELAPSED_FUNC_TIME(
413                 (Communicate<ValueType, IndexType>::local_to_global_vector
    (
414                     settings, metadata, this->local_solution, solution_vector)),
415                 4, metadata.my_rank, expand_local_vec, metadata.iter_count);
416         }
417     }
418     MPI_Barrier(MPI_COMM_WORLD);
419     auto elapsed_time = std::chrono::duration<ValueType>(
420         std::chrono::steady_clock::now() - start_time);
421     std::cout << " Rank " << metadata.my_rank << " converged in "
422               << metadata.iter_count << " iters " << std::endl;
423     ValueType mat_norm = -1.0, rhs_norm = -1.0, sol_norm = -1.0,
424               residual_norm = -1.0;
425     // Compute the final residual norm. Also gathers the solution from all
426     // subdomains.
427     Solve<ValueType, IndexType>::compute_residual_norm(
428         settings, metadata, global_matrix, global_rhs, solution_vector,
429         mat_norm, rhs_norm, sol_norm, residual_norm);
430     gather_comm_data<ValueType, IndexType>(
431         metadata.num_subdomains, this->comm_struct, metadata.comm_data_struct);
432     // clang-format off
433     if (metadata.my_rank == 0)
434       {
435         std::cout
436               << " residual norm " << residual_norm << "\n"
437               << " relative residual norm of solution " << residual_norm/rhs_norm << "\n"
438               << " Time taken for solve " << elapsed_time.count()
439               << std::endl;
440         if (num_converged_procs < metadata.num_subdomains)
441           {
442             std::cout << " Did not converge in " << metadata.iter_count
443                       << " iterations."
444                       << std::endl;
445           }
446       }
447     // clang-format on
448     if (metadata.my_rank == 0) {
449         solution->copy_from(solution_vector.get());
450     }
451
452     // Communicate<ValueType, IndexType>::clear(settings);
453 }
```

The documentation for this class was generated from the following files:

- schwarz_base.hpp (4da3021)
- /home/runner/work/schwarz-lib/schwarz-lib/source/schwarz_base.cpp (4da3021)

## 7.13 SchwarzWrappers::Settings Struct Reference

The struct that contains the solver settings and the parameters to be set by the user.

`#include <settings.hpp>`

## Classes

- struct comm_settings

  *The settings for the various available communication paradigms.*
- struct convergence_settings

  *The various convergence settings available.*

## Public Types

- enum partition_settings

  *The partition algorithm to be used for partitioning the matrix.*
- enum local_solver_settings

  *The local solver algorithm for the local subdomain solves.*

## Public Attributes

- std::string executor_string

  *The string that contains the ginkgo executor paradigm.*
- std::shared_ptr< gko::Executor > executor = gko::ReferenceExecutor::create()

  *The ginkgo executor the code is to be executed on.*
- std::shared_ptr< device_guard > cuda_device_guard

  *The ginkgo executor the code is to be executed on.*
- gko::int32 overlap = 2

  *The overlap between the subdomains.*
- bool explicit_laplacian = true

  *Flag if the laplcian matrix should be generated within the library.*
- bool enable_random_rhs = false

  *Flag to enable a random rhs.*
- bool print_matrices = false

  *Flag to enable printing of matrices.*
- bool debug_print = false

  *Flag to enable some debug printing.*
- bool naturally_ordered_factor = false

  *Disables the re-ordering of the matrix before computing the triangular factors during the CHOLMOD factorization.*
- std::string metis_objtype

  *This setting defines the objective type for the metis partitioning.*
- bool use_precond = false

  *Enable the block jacobi local preconditioner for the local solver.*
- bool write_debug_out = false

  *Enable the writing of debug out to file.*
- bool write_perm_data = false

  *Enable the local permutations from CHOLMOD to a file.*
- int shifted_iter = 1

  *Iteration shift for node local communication.*
- std::string factorization = "cholmod"

  *The factorization for the local direct solver.*
- std::string reorder

  *The reordering for the local solve.*

### 7.13.1 Detailed Description

The struct that contains the solver settings and the parameters to be set by the user.

settings

### 7.13.2 Member Data Documentation

#### 7.13.2.1 explicit_laplacian

```
bool SchwarzWrappers::Settings::explicit_laplacian = true
```

Flag if the laplcian matrix should be generated within the library.

If false, an external matrix and rhs needs to be provided

Referenced by SchwarzWrappers::SchwarzBase< ValueType, IndexType >::initialize().

#### 7.13.2.2 naturally_ordered_factor

```
bool SchwarzWrappers::Settings::naturally_ordered_factor = false
```

Disables the re-ordering of the matrix before computing the triangular factors during the CHOLMOD factorization.

**Note**

This is mainly to allow compatibility with GPU solution.

The documentation for this struct was generated from the following file:

- settings.hpp (4da3021)

## 7.14 SchwarzWrappers::Solve< ValueType, IndexType > Class Template Reference

The Solver class the provides the solver and the convergence checking methods.

```
#include <solve.hpp>
```

### Additional Inherited Members

### 7.14.1 Detailed Description

**template<typename ValueType = gko::default_precision, typename IndexType = gko::int32>**
**class SchwarzWrappers::Solve< ValueType, IndexType >**

The Solver class the provides the solver and the convergence checking methods.

**Template Parameters**

| | |
|---|---|
| *ValueType* | The type of the floating point values. |
| *IndexType* | The type of the index type values. |

Solve

The documentation for this class was generated from the following files:

- solve.hpp (4da3021)
- /home/runner/work/schwarz-lib/schwarz-lib/source/solve.cpp (4da3021)

## 7.15 SchwarzWrappers::SolverRAS< ValueType, IndexType > Class Template Reference

An implementation of the solver interface using the RAS solver.

`#include <restricted_schwarz.hpp>`

### Public Member Functions

- SolverRAS (Settings &settings, Metadata< ValueType, IndexType > &metadata)

  *The constructor that takes in the user settings and a metadata struct containing the solver metadata.*
- void setup_local_matrices (Settings &settings, Metadata< ValueType, IndexType > &metadata, std::vector< unsigned int > &partition_indices, std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> &global_matrix, std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> &local_matrix, std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> &interface_matrix) override

  *Sets up the local and the interface matrices from the global matrix and the partition indices.*
- void setup_comm_buffers () override

  *Sets up the communication buffers needed for the boundary exchange.*
- void setup_windows (const Settings &settings, const Metadata< ValueType, IndexType > &metadata, std::shared_ptr< gko::matrix::Dense< ValueType >> &main_buffer) override

  *Sets up the windows needed for the asynchronous communication.*
- void exchange_boundary (const Settings &settings, const Metadata< ValueType, IndexType > &metadata, std::shared_ptr< gko::matrix::Dense< ValueType >> &solution_vector) override

  *Exchanges the elements of the solution vector.*
- void update_boundary (const Settings &settings, const Metadata< ValueType, IndexType > &metadata, std::shared_ptr< gko::matrix::Dense< ValueType >> &local_solution, const std::shared_ptr< gko::matrix::Dense< ValueType >> &local_rhs, const std::shared_ptr< gko::matrix::Dense< ValueType >> &solution_vector, std::shared_ptr< gko::matrix::Dense< ValueType >> &global_old_solution, const std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> &interface_matrix) override

  *Update the values into local vector from obtained from the neighboring sub-domains using the interface matrix.*

### Additional Inherited Members

### 7.15.1 Detailed Description

**template<typename ValueType = gko::default_precision, typename IndexType = gko::int32>**
**class SchwarzWrappers::SolverRAS< ValueType, IndexType >**

An implementation of the solver interface using the RAS solver.

**Template Parameters**

| | |
|---|---|
| *ValueType* | The type of the floating point values. |
| *IndexType* | The type of the index type values. |

### 7.15.2 Constructor & Destructor Documentation

#### 7.15.2.1 SolverRAS()

```
template<typename ValueType , typename IndexType >
SchwarzWrappers::SolverRAS< ValueType, IndexType >::SolverRAS (
            Settings & settings,
            Metadata< ValueType, IndexType > & metadata )
```

The constructor that takes in the user settings and a metadata struct containing the solver metadata.

**Parameters**

| | |
|---|---|
| *settings* | The settings struct. |
| *metadata* | The metadata struct. |
| *data* | The additional data struct. |

```
50     : SchwarzBase<ValueType, IndexType>(settings, metadata)
51 {}
```

### 7.15.3 Member Function Documentation

#### 7.15.3.1 exchange_boundary()

```
template<typename ValueType , typename IndexType >
void SchwarzWrappers::SolverRAS< ValueType, IndexType >::exchange_boundary (
            const Settings & settings,
            const Metadata< ValueType, IndexType > & metadata,
            std::shared_ptr< gko::matrix::Dense< ValueType >> & solution_vector )  [override],
[virtual]
```

Exchanges the elements of the solution vector.

**Parameters**

| | |
|---|---|
| *settings* | The settings struct. |
| *metadata* | The metadata struct. |
| *solution_vector* | The solution vector being exchanged between the subdomains. |

Implements SchwarzWrappers::Communicate< ValueType, IndexType >.

References SchwarzWrappers::Settings::comm_settings::enable_onesided.

```
795 {
796     if (settings.comm_settings.enable_onesided) {
797         exchange_boundary_onesided<ValueType, IndexType>(
798             settings, metadata, this->comm_struct, solution_vector);
799     } else {
800         exchange_boundary_twosided<ValueType, IndexType>(
801             settings, metadata, this->comm_struct, solution_vector);
802     }
803 }
```

#### 7.15.3.2 setup_local_matrices()

```
template<typename ValueType , typename IndexType >
void SchwarzWrappers::SolverRAS< ValueType, IndexType >::setup_local_matrices (
             Settings & settings,
             Metadata< ValueType, IndexType > & metadata,
             std::vector< unsigned int > & partition_indices,
             std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> & global_matrix,
             std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> & local_matrix,
             std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> & interface_matrix )
[override], [virtual]
```

Sets up the local and the interface matrices from the global matrix and the partition indices.

**Parameters**

| | |
|---|---|
| *settings* | The settings struct. |
| *metadata* | The metadata struct. |
| *partition_indices* | The array containing the partition indices. |
| *global_matrix* | The global system matrix. |
| *local_matrix* | The local system matrix. |
| *interface_matrix* | The interface matrix containing the interface and the overlap data mainly used for exchanging values between different sub-domains. |
| *local_perm* | The local permutation, obtained through RCM or METIS. |

Implements SchwarzWrappers::Initialize< ValueType, IndexType >.

References SchwarzWrappers::Metadata< ValueType, IndexType >::comm_size, SchwarzWrappers::Settings::executor, SchwarzWrappers::Metadata< ValueType, IndexType >::first_row, SchwarzWrappers::SchwarzBase< ValueType, IndexType >::global_matrix, SchwarzWrappers::Metadata< ValueType, IndexType >::global_size, SchwarzWrappers::Metadata< ValueType, IndexType >::global_to_local, SchwarzWrappers::Metadata< ValueType, IndexType >::i_permutation, SchwarzWrappers::SchwarzBase< ValueType, IndexType >::interface_matrix, SchwarzWrappers::SchwarzBase< ValueType, IndexType >::local_matrix, SchwarzWrappers::Metadata< ValueType, IndexType >::local_size, SchwarzWrappers::Metadata< ValueType, IndexType >::local_size_o, SchwarzWrappers::Metadata< ValueType, IndexType >::local_size_x, SchwarzWrappers::Metadata< ValueType, IndexType >::local_to_global, SchwarzWrappers::Metadata< ValueType, IndexType >::my_rank, SchwarzWrappers::Metadata< ValueType, IndexType >::num_subdomains, SchwarzWrappers::Settings::overlap, SchwarzWrappers::Metadata< ValueType, IndexType >::overlap_row, SchwarzWrappers::Metadata< ValueType, IndexType >::overlap_size, and SchwarzWrappers::Metadata< ValueType, IndexType >::permutation.

```
61 {
62     using mtx = gko::matrix::Csr<ValueType, IndexType>;
63     using vec_itype = gko::Array<IndexType>;
64     using perm_type = gko::matrix::Permutation<IndexType>;
65     using arr = gko::Array<IndexType>;
66     auto my_rank = metadata.my_rank;
67     auto comm_size = metadata.comm_size;
68     auto num_subdomains = metadata.num_subdomains;
69     auto global_size = metadata.global_size;
70     auto mpi_itype = boost::mpi::get_mpi_datatype(*partition_indices.data());
71
72     MPI_Bcast(partition_indices.data(), global_size, mpi_itype, 0,
73               MPI_COMM_WORLD);
74
75     std::vector<IndexType> local_p_size(num_subdomains);
76     auto global_to_local = metadata.global_to_local->get_data();
77     auto local_to_global = metadata.local_to_global->get_data();
78
79     auto first_row = metadata.first_row->get_data();
80     auto permutation = metadata.permutation->get_data();
81     auto i_permutation = metadata.i_permutation->get_data();
82
83     auto nb = (global_size + num_subdomains - 1) /
    num_subdomains;
84     auto partition_settings =
85         (Settings::partition_settings::partition_zoltan |
86          Settings::partition_settings::partition_metis |
87          Settings::partition_settings::partition_regular |
88          Settings::partition_settings::partition_regular2d |
89          Settings::partition_settings::partition_custom) &
90         settings.partition;
91
92     IndexType *gmat_row_ptrs = global_matrix->get_row_ptrs();
93     IndexType *gmat_col_idxs = global_matrix->get_col_idxs();
94     ValueType *gmat_values = global_matrix->get_values();
95
96     // default local p size set for 1 subdomain.
97     first_row[0] = 0;
98     for (auto p = 0; p < num_subdomains; ++p) {
99         local_p_size[p] = std::min(global_size - first_row[p], nb);
100         first_row[p + 1] = first_row[p] + local_p_size[p];
101     }
102
103     if (partition_settings == Settings::partition_settings::partition_metis ||
104         partition_settings ==
105             Settings::partition_settings::partition_regular2d) {
106         if (num_subdomains > 1) {
107             for (auto p = 0; p < num_subdomains; p++) {
108                 local_p_size[p] = 0;
109             }
110             for (auto i = 0; i < global_size; i++) {
111                 local_p_size[partition_indices[i]]++;
112             }
113             first_row[0] = 0;
114             for (auto p = 0; p < num_subdomains; ++p) {
115                 first_row[p + 1] = first_row[p] + local_p_size[p];
116             }
117             // permutation
118             for (auto i = 0; i < global_size; i++) {
119                 permutation[first_row[partition_indices[i]]] = i;
120                 first_row[partition_indices[i]]++;
121             }
122             for (auto p = num_subdomains; p > 0; p--) {
123                 first_row[p] = first_row[p - 1];
124             }
125             first_row[0] = 0;
126
127             // iperm
128             for (auto i = 0; i < global_size; i++) {
129                 i_permutation[permutation[i]] = i;
130             }
131         }
132
133         auto gmat_temp = mtx::create(settings.executor->get_master(),
134                                      global_matrix->get_size(),
135                                      global_matrix->get_num_stored_elements());
136         auto nnz = 0;
137         gmat_temp->get_row_ptrs()[0] = 0;
138         for (auto row = 0; row < metadata.global_size; ++row) {
139             for (auto col = gmat_row_ptrs[permutation[row]];
140                  col < gmat_row_ptrs[permutation[row] + 1]; ++col) {
141                 gmat_temp->get_col_idxs()[nnz] =
142                     i_permutation[gmat_col_idxs[col]];
143                 gmat_temp->get_values()[nnz] = gmat_values[col];
144                 nnz++;
145             }
146             gmat_temp->get_row_ptrs()[row + 1] = nnz;
```

```
147         }
148         global_matrix->copy_from(gmat_temp.get());
149     }
150     for (auto i = 0; i < global_size; i++) {
151         global_to_local[i] = 0;
152         local_to_global[i] = 0;
153     }
154     auto num = 0;
155     for (auto i = first_row[my_rank]; i < first_row[
    my_rank + 1]; i++) {
156         global_to_local[i] = 1 + num;
157         local_to_global[num] = i;
158         num++;
159     }
160
161     IndexType old = 0;
162     for (auto k = 1; k < settings.overlap; k++) {
163         auto now = num;
164         for (auto i = old; i < now; i++) {
165             for (auto j = gmat_row_ptrs[local_to_global[i]];
166                  j < gmat_row_ptrs[local_to_global[i] + 1]; j++) {
167                 if (global_to_local[gmat_col_idxs[j]] == 0) {
168                     local_to_global[num] = gmat_col_idxs[j];
169                     global_to_local[gmat_col_idxs[j]] = 1 + num;
170                     num++;
171                 }
172             }
173         }
174         old = now;
175     }
176     metadata.local_size = local_p_size[my_rank];
177     metadata.local_size_x = num;
178     metadata.local_size_o = global_size;
179     auto local_size = metadata.local_size;
180     auto local_size_x = metadata.local_size_x;
181
182     metadata.overlap_size = num - metadata.local_size;
183     metadata.overlap_row = std::shared_ptr<vec_itype>(
184         new vec_itype(gko::Array<IndexType>::view(
185             settings.executor, metadata.overlap_size,
186             &(metadata.local_to_global->get_data()[metadata.local_size]))),
187         std::default_delete<vec_itype>());
188
189     auto nnz_local = 0;
190     auto nnz_interface = 0;
191
192     for (auto i = first_row[my_rank]; i < first_row[my_rank + 1]; ++i) {
193         for (auto j = gmat_row_ptrs[i]; j < gmat_row_ptrs[i + 1]; j++) {
194             if (global_to_local[gmat_col_idxs[j]] != 0) {
195                 nnz_local++;
196             } else {
197                 std::cout << " debug: invalid edge?" << std::endl;
198             }
199         }
200     }
201     auto temp = 0;
202     for (auto k = 0; k < metadata.overlap_size; k++) {
203         temp = metadata.overlap_row->get_data()[k];
204         for (auto j = gmat_row_ptrs[temp]; j < gmat_row_ptrs[temp + 1]; j++) {
205             if (global_to_local[gmat_col_idxs[j]] != 0) {
206                 nnz_local++;
207             } else {
208                 nnz_interface++;
209             }
210         }
211     }
212
213     std::shared_ptr<mtx> local_matrix_compute;
214     local_matrix_compute = mtx::create(settings.executor->get_master(),
215                                        gko::dim<2>(local_size_x), nnz_local);
216     IndexType *lmat_row_ptrs = local_matrix_compute->get_row_ptrs();
217     IndexType *lmat_col_idxs = local_matrix_compute->get_col_idxs();
218     ValueType *lmat_values = local_matrix_compute->get_values();
219
220     std::shared_ptr<mtx> interface_matrix_compute;
221     if (nnz_interface > 0) {
222         interface_matrix_compute =
223             mtx::create(settings.executor->get_master(),
224                         gko::dim<2>(local_size_x), nnz_interface);
225     } else {
226         interface_matrix_compute = mtx::create(settings.executor->get_master());
227     }
228
229     IndexType *imat_row_ptrs = interface_matrix_compute->get_row_ptrs();
230     IndexType *imat_col_idxs = interface_matrix_compute->get_col_idxs();
231     ValueType *imat_values = interface_matrix_compute->get_values();
232
```

```
233     num = 0;
234     nnz_local = 0;
235     auto nnz_interface_temp = 0;
236     lmat_row_ptrs[0] = nnz_local;
237     if (nnz_interface > 0) {
238         imat_row_ptrs[0] = nnz_interface_temp;
239     }
240     // Local interior matrix
241     for (auto i = first_row[my_rank]; i < first_row[my_rank + 1]; ++i) {
242         for (auto j = gmat_row_ptrs[i]; j < gmat_row_ptrs[i + 1]; ++j) {
243             if (global_to_local[gmat_col_idxs[j]] != 0) {
244                 lmat_col_idxs[nnz_local] =
245                     global_to_local[gmat_col_idxs[j]] - 1;
246                 lmat_values[nnz_local] = gmat_values[j];
247                 nnz_local++;
248             }
249         }
250         if (nnz_interface > 0) {
251             imat_row_ptrs[num + 1] = nnz_interface_temp;
252         }
253         lmat_row_ptrs[num + 1] = nnz_local;
254         num++;
255     }
256
257     // Interface matrix
258     if (nnz_interface > 0) {
259         nnz_interface = 0;
260         for (auto k = 0; k < metadata.overlap_size; k++) {
261             temp = metadata.overlap_row->get_data()[k];
262             for (auto j = gmat_row_ptrs[temp]; j < gmat_row_ptrs[temp + 1];
263                  j++) {
264                 if (global_to_local[gmat_col_idxs[j]] != 0) {
265                     lmat_col_idxs[nnz_local] =
266                         global_to_local[gmat_col_idxs[j]] - 1;
267                     lmat_values[nnz_local] = gmat_values[j];
268                     nnz_local++;
269                 } else {
270                     imat_col_idxs[nnz_interface] = gmat_col_idxs[j];
271                     imat_values[nnz_interface] = gmat_values[j];
272                     nnz_interface++;
273                 }
274             }
275             lmat_row_ptrs[num + 1] = nnz_local;
276             imat_row_ptrs[num + 1] = nnz_interface;
277             num++;
278         }
279     }
280     auto now = num;
281     for (auto i = old; i < now; i++) {
282         for (auto j = gmat_row_ptrs[local_to_global[i]];
283              j < gmat_row_ptrs[local_to_global[i] + 1]; j++) {
284             if (global_to_local[gmat_col_idxs[j]] == 0) {
285                 local_to_global[num] = gmat_col_idxs[j];
286                 global_to_local[gmat_col_idxs[j]] = 1 + num;
287                 num++;
288             }
289         }
290     }
291
292     local_matrix = mtx::create(settings.executor);
293     local_matrix->copy_from(gko::lend(local_matrix_compute));
294     interface_matrix = mtx::create(settings.executor);
295     interface_matrix->copy_from(gko::lend(interface_matrix_compute));
296     local_matrix->sort_by_column_index();
297     interface_matrix->sort_by_column_index();
298 }
```

#### 7.15.3.3 setup_windows()

```
template<typename ValueType , typename IndexType >
void SchwarzWrappers::SolverRAS< ValueType, IndexType >::setup_windows (
            const Settings & settings,
            const Metadata< ValueType, IndexType > & metadata,
            std::shared_ptr< gko::matrix::Dense< ValueType >> & main_buffer )  [override],
[virtual]
```

Sets up the windows needed for the asynchronous communication.

**Parameters**

| | |
|---|---|
| *settings* | The settings struct. |
| *metadata* | The metadata struct. |
| *main_buffer* | The main buffer being exchanged between the subdomains. |

Implements SchwarzWrappers::Communicate< ValueType, IndexType >.

References SchwarzWrappers::Settings::comm_settings::enable_get, SchwarzWrappers::Settings::comm_settings::enable_lock_all, SchwarzWrappers::Settings::comm_settings::enable_one_by_one, SchwarzWrappers::Settings::comm_settings::enable_onesided, SchwarzWrappers::Settings::comm_settings::enable_overlap, SchwarzWrappers::Settings::comm_settings::enable_put, SchwarzWrappers::Settings::executor, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::get_displacements, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::get_request, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::global_get, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::global_put, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::is_local_neighbor, SchwarzWrappers::Metadata< ValueType, IndexType >::iter_count, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::local_get, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::local_neighbors_in, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::local_neighbors_out, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::local_num_neighbors_in, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::local_num_neighbors_out, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::local_put, SchwarzWrappers::Metadata< ValueType, IndexType >::local_size_o, SchwarzWrappers::SchwarzBase< ValueType, IndexType >::local_solution, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::neighbors_in, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::neighbors_out, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::num_neighbors_in, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::num_neighbors_out, SchwarzWrappers::Metadata< ValueType, IndexType >::num_subdomains, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::put_displacements, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::put_request, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::recv_buffer, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::remote_get, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::remote_put, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::send_buffer, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::window_recv_buffer, SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::window_send_buffer, and SchwarzWrappers::Communicate< ValueType, IndexType >::comm_struct::window_x.

```
502 {
503     using vec_itype = gko::Array<IndexType>;
504     using vec_vtype = gko::matrix::Dense<ValueType>;
505     auto num_subdomains = metadata.num_subdomains;
506     auto local_size_o = metadata.local_size_o;
507     auto neighbors_in = this->comm_struct.neighbors_in->get_data();
508     auto global_get = this->comm_struct.global_get->get_data();
509     auto neighbors_out = this->comm_struct.neighbors_out->get_data();
510     auto global_put = this->comm_struct.global_put->get_data();
511
512     // set displacement for the MPI buffer
513     auto get_displacements = this->comm_struct.get_displacements->get_data();
514     auto put_displacements = this->comm_struct.put_displacements->get_data();
515     {
516         std::vector<IndexType> tmp_num_comm_elems(num_subdomains + 1, 0);
517         tmp_num_comm_elems[0] = 0;
518         for (auto j = 0; j < this->comm_struct.num_neighbors_in; j++) {
519             if ((global_get[j])[0] > 0) {
520                 int p = neighbors_in[j];
521                 tmp_num_comm_elems[p + 1] = (global_get[j])[0];
522             }
523         }
524         for (auto j = 0; j < num_subdomains; j++) {
525             tmp_num_comm_elems[j + 1] += tmp_num_comm_elems[j];
526         }
527
528         auto mpi_itype = boost::mpi::get_mpi_datatype(tmp_num_comm_elems[0]);
529         MPI_Alltoall(tmp_num_comm_elems.data(), 1, mpi_itype, put_displacements,
530                      1, mpi_itype, MPI_COMM_WORLD);
531     }
```

```
532
533     {
534         std::vector<IndexType> tmp_num_comm_elems(num_subdomains + 1, 0);
535         tmp_num_comm_elems[0] = 0;
536         for (auto j = 0; j < this->comm_struct.num_neighbors_out; j++) {
537             if ((global_put[j])[0] > 0) {
538                 int p = neighbors_out[j];
539                 tmp_num_comm_elems[p + 1] = (global_put[j])[0];
540             }
541         }
542         for (auto j = 0; j < num_subdomains; j++) {
543             tmp_num_comm_elems[j + 1] += tmp_num_comm_elems[j];
544         }
545
546         auto mpi_itype = boost::mpi::get_mpi_datatype(tmp_num_comm_elems[0]);
547         MPI_Alltoall(tmp_num_comm_elems.data(), 1, mpi_itype, get_displacements,
548                      1, mpi_itype, MPI_COMM_WORLD);
549     }
550
551     // setup windows
552     if (settings.comm_settings.enable_onesided) {
553         // Onesided
554         MPI_Win_create(main_buffer->get_values(),
555                        main_buffer->get_size()[0] * sizeof(ValueType),
556                        sizeof(ValueType), MPI_INFO_NULL, MPI_COMM_WORLD,
557                        &(this->comm_struct.window_x));
558     }
559
560
561     if (settings.comm_settings.enable_onesided) {
562         // MPI_Alloc_mem ? Custom allocator ?  TODO
563         MPI_Win_create(this->local_residual_vector->get_values(),
564                        (num_subdomains) * sizeof(ValueType), sizeof(ValueType),
565                        MPI_INFO_NULL, MPI_COMM_WORLD,
566                        &(this->window_residual_vector));
567         std::vector<IndexType> zero_vec(num_subdomains, 0);
568         gko::Array<IndexType> temp_array{settings.executor->get_master(),
569                                          zero_vec.begin(), zero_vec.end()};
570         this->convergence_vector = std::shared_ptr<vec_itype>(
571             new vec_itype(settings.executor->get_master(), temp_array),
572             std::default_delete<vec_itype>());
573         this->convergence_sent = std::shared_ptr<vec_itype>(
574             new vec_itype(settings.executor->get_master(), num_subdomains),
575             std::default_delete<vec_itype>());
576         this->convergence_local = std::shared_ptr<vec_itype>(
577             new vec_itype(settings.executor->get_master(), num_subdomains),
578             std::default_delete<vec_itype>());
579         MPI_Win_create(this->convergence_vector->get_data(),
580                        (num_subdomains) * sizeof(IndexType), sizeof(IndexType),
581                        MPI_INFO_NULL, MPI_COMM_WORLD,
582                        &(this->window_convergence));
583     }
584
585     if (settings.comm_settings.enable_onesided && num_subdomains > 1) {
586         // Lock all windows.
587         if (settings.comm_settings.enable_get &&
588             settings.comm_settings.enable_lock_all) {
589             MPI_Win_lock_all(0, this->comm_struct.window_send_buffer);
590         }
591         if (settings.comm_settings.enable_put &&
592             settings.comm_settings.enable_lock_all) {
593             MPI_Win_lock_all(0, this->comm_struct.window_recv_buffer);
594         }
595         if (settings.comm_settings.enable_one_by_one &&
596             settings.comm_settings.enable_lock_all) {
597             MPI_Win_lock_all(0, this->comm_struct.window_x);
598         }
599         MPI_Win_lock_all(0, this->window_residual_vector);
600         MPI_Win_lock_all(0, this->window_convergence);
601     }
602 }
```

#### 7.15.3.4 update_boundary()

```
template<typename ValueType , typename IndexType >
void SchwarzWrappers::SolverRAS< ValueType, IndexType >::update_boundary (
             const Settings & settings,
```

```
            const Metadata< ValueType, IndexType > & metadata,
            std::shared_ptr< gko::matrix::Dense< ValueType >> & local_solution,
            const std::shared_ptr< gko::matrix::Dense< ValueType >> & local_rhs,
            const std::shared_ptr< gko::matrix::Dense< ValueType >> & solution_vector,
            std::shared_ptr< gko::matrix::Dense< ValueType >> & global_old_solution,
            const std::shared_ptr< gko::matrix::Csr< ValueType, IndexType >> & interface_↩
matrix )  [override], [virtual]
```

Update the values into local vector from obtained from the neighboring sub-domains using the interface matrix.

**Parameters**

| | |
|---|---|
| *settings* | The settings struct. |
| *metadata* | The metadata struct. |
| *local_solution* | The local solution vector in the subdomain. |
| *local_rhs* | The local right hand side vector in the subdomain. |
| *solution_vector* | The workspace solution vector. |
| *global_old_solution* | The global solution vector of the previous iteration. |
| *interface_matrix* | The interface matrix containing the interface and the overlap data mainly used for exchanging values between different sub-domains. |

Implements SchwarzWrappers::Communicate< ValueType, IndexType >.

References SchwarzWrappers::Settings::executor, SchwarzWrappers::SchwarzBase< ValueType, IndexType >↩ ::interface_matrix, SchwarzWrappers::SchwarzBase< ValueType, IndexType >::local_rhs, SchwarzWrappers↩ ::Metadata< ValueType, IndexType >::local_size_x, SchwarzWrappers::SchwarzBase< ValueType, IndexType >::local_solution, SchwarzWrappers::Metadata< ValueType, IndexType >::num_subdomains, and Schwarz↩ Wrappers::Settings::overlap.

```
815 {
816     using vec_vtype = gko::matrix::Dense<ValueType>;
817     auto one = gko::initialize<gko::matrix::Dense<ValueType>>(
818         {1.0}, settings.executor);
819     auto neg_one = gko::initialize<gko::matrix::Dense<ValueType>>(
820         {-1.0}, settings.executor);
821     auto local_size_x = metadata.local_size_x;
822     local_solution->copy_from(local_rhs.get());
823     global_old_solution->copy_from(solution_vector.get());
824     if (metadata.num_subdomains > 1 && settings.overlap > 0) {
825         auto temp_solution = vec_vtype::create(
826             settings.executor, local_solution->get_size(),
827             gko::Array<ValueType>::view(
828                 settings.executor, local_solution->get_size()[0],
829                 &(global_old_solution->get_values()[0])),
830             1);
831         interface_matrix->apply(neg_one.get(), temp_solution.get(), one.get(),
832                                 (local_solution).get());
833     }
834 }
```

The documentation for this class was generated from the following files:

- restricted_schwarz.hpp (4da3021)
- /home/runner/work/schwarz-lib/schwarz-lib/source/restricted_schwarz.cpp (4da3021)

## 7.16 UmfpackError Class Reference

UmfpackError is thrown when a METIS routine throws a non-zero error code.

```
#include <exception.hpp>
```

### Public Member Functions

- UmfpackError (const std::string &file, int line, const std::string &func, int error_code)
  *Initializes a METIS error.*

## 7.16.1 Detailed Description

UmfpackError is thrown when a METIS routine throws a non-zero error code.

## 7.16.2 Constructor & Destructor Documentation

### 7.16.2.1 UmfpackError()

```
UmfpackError::UmfpackError (
            const std::string & file,
            int line,
            const std::string & func,
            int error_code )  [inline]
```

Initializes a METIS error.

**Parameters**

| | |
|---|---|
| *file* | The name of the offending source file |
| *line* | The source code line number where the error occurred |
| *func* | The name of the METIS routine that failed |
| *error_code* | The resulting METIS error code |

```
205         : Error(file, line, func + ": " + get_error(error_code))
206     {}
```

The documentation for this class was generated from the following files:

- exception.hpp (4da3021)
- /home/runner/work/schwarz-lib/schwarz-lib/source/exception.cpp (4da3021)

# 7.17 SchwarzWrappers::Utils< ValueType, IndexType > Struct Template Reference

The utilities class which provides some checks and basic utilities.

```
#include <utils.hpp>
```

### 7.17.1 Detailed Description

**template<typename ValueType = gko::default_precision, typename IndexType = gko::int32>**
**struct SchwarzWrappers::Utils< ValueType, IndexType >**

The utilities class which provides some checks and basic utilities.

**Template Parameters**

| | |
|---|---|
| *ValueType* | The type of the floating point values. |
| *IndexType* | The type of the index type values. |

Utils

The documentation for this struct was generated from the following files:

- utils.hpp (4da3021)
- /home/runner/work/schwarz-lib/schwarz-lib/source/utils.cpp (4da3021)

# Index