schwz

Generated automatically from fix-gpu-transfers

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 -DSCHWARZ_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

- schwz::CommHelpers

  The CommHelper namespace .
- ProcessTopology

  The ProcessTopology namespace .

Classes

- class schwz::Communicate< ValueType, IndexType >

  The communication class that provides the methods for the communication between the subdomains.
- struct schwz::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

- schwz::PartitionTools

  The PartitionTools namespace .
- ProcessTopology

  The ProcessTopology namespace .

Classes

- class schwz::device_guard

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

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

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

  An implementation of the solver interface using the RAS solver.
- class schwz::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

- schwz::conv_tools

  The conv_tools namespace .
- schwz::SolverTools

  The SolverTools namespace .

Classes

- struct schwz::Metadata< ValueType, IndexType >

  The solver metadata struct.
- class schwz::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 schwz::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 schwz Namespace Reference

The Schwarz wrappers namespace.

Namespaces

- CommHelpers

  The CommHelper namespace .
- conv_tools

  The conv_tools 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 schwz::CommHelpers Namespace Reference

The CommHelper namespace .

6.3.1 Detailed Description

The CommHelper namespace .

comm_helpers

6.4 schwz::conv_tools Namespace Reference

The conv_tools namespace .

6.4.1 Detailed Description

The conv_tools namespace .

conv_tools

6.5 schwz::PartitionTools Namespace Reference

The PartitionTools namespace .

6.5.1 Detailed Description

The PartitionTools namespace .

part_tools

6.6 schwz::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

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

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

- exception.hpp (e0c2959)

7.2 schwz::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 (e0c2959)

7.3 schwz::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::matrix::Dense< ValueType > > cpu_send_buffer

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

  The RDMA window for the recv buffer.
- MPI_Win window_cpu_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 schwz::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 *> > schwz::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 schwz::SchwarzBase< ValueType, IndexType >::initialize(), schwz::SolverRAS< ValueType, IndexType >::setup_comm_buffers(), and schwz::SolverRAS< ValueType, IndexType >::setup_windows().

7.3.2.2 global_put

```
template<typename ValueType , typename IndexType >
std::shared_ptr<gko::Array<IndexType *> > schwz::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 schwz::SchwarzBase< ValueType, IndexType >::initialize(), schwz::SolverRAS< ValueType, IndexType >::setup_comm_buffers(), and schwz::SolverRAS< ValueType, IndexType >::setup_windows().

7.3.2.3 is_local_neighbor

```
template<typename ValueType , typename IndexType >
std::vector<bool> schwz::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 schwz::SchwarzBase< ValueType, IndexType >::initialize(), schwz::SolverRAS< ValueType, IndexType >::setup_comm_buffers(), and schwz::SolverRAS< ValueType, IndexType >::setup_windows().

7.3.2.4 local_get

```
template<typename ValueType , typename IndexType >
std::shared_ptr<gko::Array<IndexType *> > schwz::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 schwz::SolverRAS< ValueType, IndexType >::setup_comm_buffers(), and schwz::SolverRAS< ValueType, IndexType >::setup_windows().

7.3.2.5 local_put

```
template<typename ValueType , typename IndexType >
std::shared_ptr<gko::Array<IndexType *> > schwz::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 schwz::SolverRAS< ValueType, IndexType >::setup_comm_buffers(), and schwz::SolverRAS< ValueType, IndexType >::setup_windows().

7.3.2.6 remote_get

```
template<typename ValueType , typename IndexType >
std::shared_ptr<gko::Array<IndexType *> > schwz::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 schwz::SolverRAS< ValueType, IndexType >::setup_comm_buffers(), and schwz::SolverRAS< ValueType, IndexType >::setup_windows().

7.3.2.7 remote_put

```
template<typename ValueType , typename IndexType >
std::shared_ptr<gko::Array<IndexType *> > schwz::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 schwz::SolverRAS< ValueType, IndexType >::setup_comm_buffers(), and schwz::SolverRAS< ValueType, IndexType >::setup_windows().

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

- communicate.hpp (e0c2959)

7.4 schwz::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 >> &global_solution)=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 >> &global_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 schwz::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 schwz::Communicate< ValueType, IndexType >::exchange_boundary (
            const Settings & settings,
            const Metadata< ValueType, IndexType > & metadata,
            std::shared_ptr< gko::matrix::Dense< ValueType >> & global_solution )  [pure
virtual]
```

Exchanges the elements of the solution vector.

Parameters

settings	The settings struct.
metadata	The metadata struct.
global_solution	The solution vector being exchanged between the subdomains.

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

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

7.4.2.2 local_to_global_vector()

```
template<typename ValueType , typename IndexType >
void schwz::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.

```
{
    using vec = gko::matrix::Dense<ValueType>;
    auto alpha = gko::initialize<gko::matrix::Dense<ValueType>>(
        {1.0}, settings.executor);
    auto temp_vector = vec::create(
        settings.executor, gko::dim<2>(metadata.local_size, 1),
        gko::Array<ValueType>::view(
            settings.executor, metadata.local_size,
            &global_vector->get_values()[metadata.first_row
                                             ->get_data()[metadata.my_rank]]),
        1);

    auto temp_vector2 = vec::create(
        settings.executor, gko::dim<2>(metadata.local_size, 1),
        gko::Array<ValueType>::view(settings.executor, metadata.local_size,
                                    local_vector->get_values()),
        1);
    if (settings.convergence_settings.convergence_crit ==
        Settings::convergence_settings::local_convergence_crit::
            residual_based) {
        local_vector->add_scaled(alpha.get(), temp_vector.get());
        temp_vector->add_scaled(alpha.get(), local_vector.get());
    } else {
        temp_vector->copy_from(temp_vector2.get());
    }
}
```

7.4.2.3 setup_windows()

```
template<typename ValueType , typename IndexType >
void schwz::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 schwz::SolverRAS< ValueType, IndexType >.

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

7.4.2.4 update_boundary()

```
template<typename ValueType , typename IndexType >
void schwz::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 >> & global_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.
global_solution	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 schwz::SolverRAS< ValueType, IndexType >.

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

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

- communicate.hpp (e0c2959)
- /home/runner/work/schwarz-lib/schwarz-lib/source/communicate.cpp (e0c2959)

7.5 schwz::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 (e0c2959)

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

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

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

- exception.hpp (e0c2959)
- /home/runner/work/schwarz-lib/schwarz-lib/source/exception.cpp (e0c2959)

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

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

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

- exception.hpp (e0c2959)
- /home/runner/work/schwarz-lib/schwarz-lib/source/exception.cpp (e0c2959)

7.8 schwz::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 (e0c2959)

7.9 schwz::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 (const std::string &filename, 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)

  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 schwz::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 schwz::Initialize< ValueType, IndexType >::generate_rhs (
            std::vector< ValueType > & rhs )
```

Generates the right hand side vector.

Parameters

rhs	The rhs vector.

References schwz::Initialize< ValueType, IndexType >::setup_global_matrix().

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

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

7.9.2.2 partition()

```
template<typename ValueType , typename IndexType >
void schwz::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 schwz::Metadata< ValueType, IndexType >::global_size, schwz::Metadata< ValueType, IndexType >::my_rank, schwz::Metadata< ValueType, IndexType >::num_subdomains, and schwz::Settings::write_debug_out.

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

```
{
    partition_indices.resize(metadata.global_size);
    if (metadata.my_rank == 0) {
        auto partition_settings =
            (Settings::partition_settings::partition_zoltan |
             Settings::partition_settings::partition_metis |
             Settings::partition_settings::partition_regular |
             Settings::partition_settings::partition_regular2d |
             Settings::partition_settings::partition_custom) &
            settings.partition;

        if (partition_settings ==
            Settings::partition_settings::partition_zoltan) {
            SCHWARZ_NOT_IMPLEMENTED;
        } else if (partition_settings ==
                   Settings::partition_settings::partition_metis) {
            if (metadata.my_rank == 0) {
                std::cout << " METIS partition" << std::endl;
            }
            PartitionTools::PartitionMetis(
                settings, global_matrix, this->cell_weights,
                metadata.num_subdomains, partition_indices);
        } else if (partition_settings ==
                   Settings::partition_settings::partition_regular) {
            if (metadata.my_rank == 0) {
                std::cout << " Regular 1D partition" << std::endl;
            }
            PartitionTools::PartitionRegular(
                global_matrix, metadata.num_subdomains, partition_indices);
        } else if (partition_settings ==
                   Settings::partition_settings::partition_regular2d) {
            if (metadata.my_rank == 0) {
                std::cout << " Regular 2D partition" << std::endl;
            }
            PartitionTools::PartitionRegular2D(
                global_matrix, settings.write_debug_out,
                metadata.num_subdomains, partition_indices);
        } else if (partition_settings ==
                   Settings::partition_settings::partition_custom) {
            // User partitions mesh manually
            SCHWARZ_NOT_IMPLEMENTED;
        } else {
            SCHWARZ_NOT_IMPLEMENTED;
        }
    }
}
```

7.9.2.3 setup_global_matrix()

```
template<typename ValueType , typename IndexType >
void schwz::Initialize< ValueType, IndexType >::setup_global_matrix (
              const std::string & filename,
              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 schwz::Initialize< ValueType, IndexType >::generate_rhs(), and schwz::SchwarzBase< ValueType, IndexType >::initialize().

```
{
    using index_type = IndexType;
    using value_type = ValueType;
    using mtx = gko::matrix::Csr<value_type, index_type>;
    if (settings.matrix_filename != "null") {
        auto input_file = std::ifstream(filename, std::ios::in);
        if (!input_file) {
            std::cerr << "Could not find the file \"" << filename
                      << "\", which is required for this test.\n";
        }
        global_matrix =
            gko::read<mtx>(input_file, settings.executor->get_master());
        global_matrix->sort_by_column_index();
        std::cout << "Matrix from file " << filename << std::endl;
    } else if (settings.matrix_filename == "null" &&
               settings.explicit_laplacian) {
        std::cout << "Laplacian 2D Matrix (generated in house) " << std::endl;
        gko::size_type global_size = oned_laplacian_size *
  oned_laplacian_size;

        global_matrix = mtx::create(settings.executor->get_master(),
                                    gko::dim<2>(global_size), 5 * global_size);
        value_type *values = global_matrix->get_values();
        index_type *row_ptrs = global_matrix->get_row_ptrs();
        index_type *col_idxs = global_matrix->get_col_idxs();

        std::vector<gko::size_type> exclusion_set;

        std::map<IndexType, ValueType> stencil_map = {
            {-oned_laplacian_size, -1}, {-1, -1}, {0, 4}, {1, -1},
            {oned_laplacian_size, -1},
        };
        for (auto i = 2; i < global_size; ++i) {
            gko::size_type index = (i - 1) * oned_laplacian_size;
            if (index * index < global_size * global_size) {
                exclusion_set.push_back(
                    linearize_index(index, index - 1, global_size));
                exclusion_set.push_back(
                    linearize_index(index - 1, index, global_size));
            }
        }

        std::sort(exclusion_set.begin(),
                  exclusion_set.begin() + exclusion_set.size());

        IndexType pos = 0;
        IndexType col_idx = 0;
        row_ptrs[0] = pos;
        gko::size_type cur_idx = 0;
        for (IndexType i = 0; i < global_size; ++i) {
            for (auto ofs : stencil_map) {
                auto in_exclusion_flag =
                    (exclusion_set[cur_idx] ==
                     linearize_index(i, i + ofs.first, global_size));
                if (0 <= i + ofs.first && i + ofs.first < global_size &&
                    !in_exclusion_flag) {
                    values[pos] = ofs.second;
                    col_idxs[pos] = i + ofs.first;
                    ++pos;
                }
                if (in_exclusion_flag) {
                    cur_idx++;
                }
                col_idx = row_ptrs[i + 1] - pos;
            }
            row_ptrs[i + 1] = pos;
        }
    } else {
        std::cerr << " Need to provide a matrix or enable the default "
                     "laplacian matrix."
                  << std::endl;
        std::exit(-1);
    }
}
```

7.9.2.4 setup_local_matrices()

```
template<typename ValueType , typename IndexType >
void schwz::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 schwz::SolverRAS< ValueType, IndexType >.

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

7.9.2.5 setup_vectors()

```
template<typename ValueType , typename IndexType >
void schwz::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 )
```

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.

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

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

```
{
    using vec = gko::matrix::Dense<ValueType>;
    auto my_rank = metadata.my_rank;
    auto first_row = metadata.first_row->get_data()[my_rank];

    // Copy the global rhs vector to the required executor.
    gko::Array<ValueType> temp_rhs{settings.executor->get_master(), rhs.begin(),
                                   rhs.end()};
    global_rhs = vec::create(settings.executor,
                             gko::dim<2>{metadata.global_size, 1}, temp_rhs, 1);

    local_rhs =
        vec::create(settings.executor, gko::dim<2>(metadata.local_size_x, 1));
    // Extract the local rhs from the global rhs. Also takes into account the
    // overlap.
    SolverTools::extract_local_vector(settings, metadata, local_rhs.get(),
                                      global_rhs.get(), first_row);

    local_solution =
        vec::create(settings.executor, gko::dim<2>(metadata.local_size_x, 1));
}
```

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

- initialization.hpp (e0c2959)
- /home/runner/work/schwarz-lib/schwarz-lib/source/initialization.cpp (e0c2959)

7.10 schwz::Metadata< ValueType, IndexType > Struct Template Reference

The solver metadata struct.

`#include <settings.hpp>`

Classes

- struct post_process_data

  The struct used for storing data for post-processing.

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 Schwarz solver.
- IndexType local_max_iters

  The maximum iteration count of the local iterative solver.
- std::string local_precond

  Local preconditioner.
- 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 schwz::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 schwz::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 schwz::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 (e0c2959)

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

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

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

- exception.hpp (e0c2959)
- /home/runner/work/schwarz-lib/schwarz-lib/source/exception.cpp (e0c2959)

7.12 schwz::Metadata< ValueType, IndexType >::post_process_data Struct Reference

The struct used for storing data for post-processing.

```
#include <settings.hpp>
```

7.12.1 Detailed Description

template<typename ValueType, typename IndexType>
struct schwz::Metadata< ValueType, IndexType >::post_process_data

The struct used for storing data for post-processing.

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

- settings.hpp (e0c2959)

7.13 schwz::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_perm

  The local subdomain permutation matrix/array.
- std::shared_ptr< gko::matrix::Permutation< IndexType > > local_inv_perm

  The local subdomain inverse 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 > > work_vector

  A work vector on the device.
- std::shared_ptr< gko::matrix::Dense< ValueType > > cpu_work_vector

  The work vector on the CPU.
- 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.
- std::vector< ValueType > local_residual_vector_out

  The global residual vector.
- std::vector< std::vector< ValueType > > global_residual_vector_out

  The local residual vector.

Additional Inherited Members

7.13.1 Detailed Description

template<typename ValueType = gko::default_precision, typename IndexType = gko::int32>
class schwz::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.

7.13.2 Constructor & Destructor Documentation

7.13.2.1 SchwarzBase()

```
template<typename ValueType , typename IndexType >
schwz::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 schwz::Settings::cuda_device_guard, schwz::Settings::executor, schwz::Settings::executor_string, schwz::Metadata< ValueType, IndexType >::local_num_procs, schwz::Metadata< ValueType, IndexType >::mpi_communicator, schwz::Metadata< ValueType, IndexType >::my_local_rank, and schwz::Metadata< ValueType, IndexType >::my_rank.

```
    : Initialize<ValueType, IndexType>(settings, metadata),
      settings(settings),
      metadata(metadata)
{
    using vec_itype = gko::Array<IndexType>;
    using vec_vecshared = gko::Array<IndexType *>;
    metadata.my_local_rank =
        Utils<ValueType, IndexType>::get_local_rank(metadata.mpi_communicator);
    metadata.local_num_procs = Utils<ValueType, IndexType>::get_local_num_procs(
        metadata.mpi_communicator);
    auto my_local_rank = metadata.my_local_rank;
    if (settings.executor_string == "omp") {
        settings.executor = gko::OmpExecutor::create();
        auto exec_info =
            static_cast<gko::OmpExecutor *>(settings.executor.get())
                ->get_exec_info();
        exec_info->bind_to_core(metadata.my_local_rank);

    } else if (settings.executor_string == "cuda") {
        int num_devices = 0;
#if SCHW_HAVE_CUDA
        SCHWARZ_ASSERT_NO_CUDA_ERRORS(cudaGetDeviceCount(&num_devices));
#else
        SCHWARZ_NOT_IMPLEMENTED;
#endif
        Utils<ValueType, IndexType>::assert_correct_cuda_devices(
            num_devices, metadata.my_rank);
        settings.executor = gko::CudaExecutor::create(
            my_local_rank, gko::OmpExecutor::create());
        auto exec_info = static_cast<gko::OmpExecutor *>(
                             settings.executor->get_master().get())
                             ->get_exec_info();
        exec_info->bind_to_core(my_local_rank);
        settings.cuda_device_guard =
            std::make_shared<schwz::device_guard>(my_local_rank);

        std::cout << " Rank " << metadata.my_rank << " with local rank "
                  << my_local_rank << " has "
                  << (static_cast<gko::CudaExecutor *>(settings.executor.get()))
                         ->get_device_id()
                  << " id of gpu" << std::endl;
        MPI_Barrier(metadata.mpi_communicator);
    } else if (settings.executor_string == "reference") {
        settings.executor = gko::ReferenceExecutor::create();
        auto exec_info =
            static_cast<gko::ReferenceExecutor *>(settings.executor.get())
                ->get_exec_info();
        exec_info->bind_to_core(my_local_rank);
    }
}
```

7.13.3 Member Function Documentation

7.13.3.1 print_matrix()

```
template<typename ValueType = gko::default_precision, typename IndexType = gko::int32>
void schwz::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.

7.13.3.2 print_vector()

```
template<typename ValueType = gko::default_precision, typename IndexType = gko::int32>
void schwz::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.

7.13.3.3 run()

```
template<typename ValueType , typename IndexType >
void schwz::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 schwz::SchwarzBase< ValueType, IndexType >::cpu_work_vector, schwz::Communicate< ValueType, IndexType >::exchange_boundary(), schwz::Settings::executor, schwz::SchwarzBase< ValueType, IndexType >::global_matrix, schwz::SchwarzBase< ValueType, IndexType >::global_rhs, schwz::SchwarzBase< ValueType, IndexType >::global_solution, schwz::SchwarzBase< ValueType, IndexType >::interface_matrix, schwz::SchwarzBase< ValueType, IndexType >::local_inv_perm, schwz::SchwarzBase< ValueType, IndexType >::local_matrix, schwz::SchwarzBase< ValueType, IndexType >::local_perm, schwz::SchwarzBase< ValueType, IndexType >::local_rhs, schwz::SchwarzBase< ValueType, IndexType >::local_solution, schwz::Communicate< ValueType, IndexType >::setup_windows(), schwz::SchwarzBase< ValueType, IndexType >::triangular_factor_l, schwz::SchwarzBase< ValueType, IndexType >::triangular_factor_u, schwz::Communicate< ValueType, IndexType >::update_boundary(), schwz::SchwarzBase< ValueType, IndexType >::work_vector, and schwz::Settings::write_iters_and_residuals.

```
{
    using vec_vtype = gko::matrix::Dense<ValueType>;
    if (!solution.get()) {
        solution =
            vec_vtype::create(settings.executor->get_master(),
                              gko::dim<2>(this->metadata.global_size, 1));
    }
    // The main solution vector
    std::shared_ptr<vec_vtype> global_solution = vec_vtype::create(
        this->settings.executor, gko::dim<2>(this->metadata.global_size, 1));
    // Work vectors.
    this->work_vector = vec_vtype::create(
        settings.executor, gko::dim<2>(2 * this->metadata.local_size_x, 1));
    this->cpu_work_vector =
        vec_vtype::create(settings.executor->get_master(),
                          gko::dim<2>(2 * this->metadata.local_size_x, 1));
    // An initial guess.
    std::shared_ptr<vec_vtype> init_guess = vec_vtype::create(
        settings.executor, gko::dim<2>(this->metadata.local_size_x, 1));
    init_guess->copy_from(local_rhs.get());

    // std::vector<IndexType> local_converged_iter_count;

    // Setup the windows for the onesided communication.
    this->setup_windows(this->settings, this->metadata, global_solution);

    const auto solver_settings =
        (Settings::local_solver_settings::direct_solver_cholmod |
         Settings::local_solver_settings::direct_solver_umfpack |
         Settings::local_solver_settings::direct_solver_ginkgo |
         Settings::local_solver_settings::iterative_solver_dealii |
         Settings::local_solver_settings::iterative_solver_ginkgo) &
        settings.local_solver;

    ValueType local_residual_norm = -1.0, local_residual_norm0 = -1.0,
              global_residual_norm = 0.0, global_residual_norm0 = -1.0;
    metadata.iter_count = 0;
    auto start_time = std::chrono::steady_clock::now();
    int num_converged_procs = 0;
    for (; metadata.iter_count < metadata.max_iters; ++(metadata.iter_count)) {
        // Exchange the boundary values. The communication part.
        MEASURE_ELAPSED_FUNC_TIME(
            this->exchange_boundary(settings, metadata, global_solution), 0,
            metadata.my_rank, boundary_exchange, metadata.iter_count);

        // Update the boundary and interior values after the exchanging from
        // other processes.
        MEASURE_ELAPSED_FUNC_TIME(
            this->update_boundary(settings, metadata, this->
  local_solution,
                                  this->local_rhs, global_solution,
                                  this->interface_matrix),
            1, metadata.my_rank, boundary_update, metadata.iter_count);

        // Check for the convergence of the solver.
        // num_converged_procs = 0;
        MEASURE_ELAPSED_FUNC_TIME(
            (Solve<ValueType, IndexType>::check_convergence(
                settings, metadata, this->comm_struct, this->convergence_vector,
                global_solution, this->local_solution, this->
  local_matrix,
                work_vector, local_residual_norm, local_residual_norm0,
                global_residual_norm, global_residual_norm0,
                num_converged_procs)),
            2, metadata.my_rank, convergence_check, metadata.iter_count);

        // break if the solution diverges.
        if (std::isnan(global_residual_norm) || global_residual_norm > 1e12) {
            std::cout << " Rank " << metadata.my_rank << " diverged in "
                      << metadata.iter_count << " iters " << std::endl;
            std::exit(-1);
        }

        // break if all processes detect that all other processes have
        // converged otherwise continue iterations.
        if (num_converged_procs == metadata.num_subdomains) {
            break;
        } else {
            MEASURE_ELAPSED_FUNC_TIME(
                (Solve<ValueType, IndexType>::local_solve(
                    settings, metadata, this->local_matrix,
                    this->triangular_factor_l, this->
  triangular_factor_u,
                    this->local_perm, this->local_inv_perm,
  work_vector,
                    init_guess, this->local_solution)),
                3, metadata.my_rank, local_solve, metadata.iter_count);
            // Gather the local vector into the locally global vector for
            // communication.
            MEASURE_ELAPSED_FUNC_TIME(
                (Communicate<ValueType, IndexType>::local_to_global_vector
  (
                    settings, metadata, this->local_solution, global_solution)),
                4, metadata.my_rank, expand_local_vec, metadata.iter_count);
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
    auto elapsed_time = std::chrono::duration<ValueType>(
        std::chrono::steady_clock::now() - start_time);
    std::cout << " Rank " << metadata.my_rank << " converged in "
              << metadata.iter_count << " iters " << std::endl;
    ValueType mat_norm = -1.0, rhs_norm = -1.0, sol_norm = -1.0,
              residual_norm = -1.0;
    // Write the residuals and iterations to files
    if (settings.write_iters_and_residuals &&
        solver_settings ==
            Settings::local_solver_settings::iterative_solver_ginkgo) {
        std::string rank_string = std::to_string(metadata.my_rank);
        if (metadata.my_rank < 10) {
            rank_string = "0" + std::to_string(metadata.my_rank);
        }
        std::string filename = "iter_res_" + rank_string + ".csv";
        write_iters_and_residuals(
            metadata.num_subdomains, metadata.my_rank,
            metadata.post_process_data.local_residual_vector_out.size(),
            metadata.post_process_data.local_residual_vector_out,
            metadata.post_process_data.local_converged_iter_count,
            metadata.post_process_data.local_converged_resnorm, filename);
    }

    // Compute the final residual norm. Also gathers the solution from all
    // subdomains.
    Solve<ValueType, IndexType>::compute_residual_norm(
        settings, metadata, global_matrix, global_rhs, global_solution,
        mat_norm, rhs_norm, sol_norm, residual_norm);
    gather_comm_data<ValueType, IndexType>(
        metadata.num_subdomains, this->comm_struct, metadata.comm_data_struct);
    // clang-format off
    if (metadata.my_rank == 0)
      {
        std::cout
              << " residual norm " << residual_norm << "\n"
              << " relative residual norm of solution " << residual_norm/rhs_norm << "\n"
              << " Time taken for solve " << elapsed_time.count()
              << std::endl;
        if (num_converged_procs < metadata.num_subdomains)
          {
            std::cout << " Did not converge in " << metadata.iter_count
                      << " iterations."
                      << std::endl;
          }
      }
    // clang-format on
    if (metadata.my_rank == 0) {
        solution->copy_from(global_solution.get());
    }

    // Communicate<ValueType, IndexType>::clear(settings);
}
```

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

- schwarz_base.hpp (e0c2959)
- /home/runner/work/schwarz-lib/schwarz-lib/source/schwarz_base.cpp (e0c2959)

7.14 schwz::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.
- std::string matrix_filename = "null"

    The string that contains the matrix file name to read from .
- bool explicit_laplacian = true

    Flag if the laplacian 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 non_symmetric_matrix = false

    Is the matrix non-symmetric ? , Use GMRES for local solves.
- unsigned int restart_iter = 1u

    The restart iter for the GMRES solver.
- 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_iters_and_residuals = false

    Enable writing the iters and residuals to a 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.

7.14.1 Detailed Description

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

settings

7.14.2 Member Data Documentation

7.14.2.1 explicit_laplacian

```
bool schwz::Settings::explicit_laplacian = true
```

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

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

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

7.14.2.2 naturally_ordered_factor

```
bool schwz::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 (e0c2959)

7.15 schwz::Solve< ValueType, IndexType > Class Template Reference

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

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

Additional Inherited Members

7.15.1 Detailed Description

template<typename ValueType = gko::default_precision, typename IndexType = gko::int32>
class schwz::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 (e0c2959)
- /home/runner/work/schwarz-lib/schwarz-lib/source/solve.cpp (e0c2959)

7.16 schwz::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 >> &global_solution) 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 >> &global↩ _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

7.16.1 Detailed Description

template<typename ValueType = gko::default_precision, typename IndexType = gko::int32>
class schwz::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.

7.16.2 Constructor & Destructor Documentation

7.16.2.1 SolverRAS()

```
template<typename ValueType , typename IndexType >
schwz::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.

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

7.16.3 Member Function Documentation

7.16.3.1 exchange_boundary()

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

Exchanges the elements of the solution vector.

Parameters

settings	The settings struct.
metadata	The metadata struct.
global_solution	The solution vector being exchanged between the subdomains.

Implements schwz::Communicate< ValueType, IndexType >.

References schwz::SchwarzBase< ValueType, IndexType >::cpu_work_vector, schwz::Settings::comm_settings::enable_onesided, schwz::SchwarzBase< ValueType, IndexType >::global_solution, and schwz::SchwarzBase< ValueType, IndexType >::work_vector.

```
{
    if (settings.comm_settings.enable_onesided) {
        exchange_boundary_onesided<ValueType, IndexType>(
            settings, metadata, this->comm_struct, this->work_vector,
            this->cpu_work_vector, global_solution);
    } else {
        exchange_boundary_twosided<ValueType, IndexType>(
            settings, metadata, this->comm_struct, global_solution);
    }
}
```

7.16.3.2 setup_local_matrices()

```
template<typename ValueType , typename IndexType >
void schwz::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 schwz::Initialize< ValueType, IndexType >.

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

```
{
    using mtx = gko::matrix::Csr<ValueType, IndexType>;
    using vec_itype = gko::Array<IndexType>;
    using perm_type = gko::matrix::Permutation<IndexType>;
    using arr = gko::Array<IndexType>;
    auto my_rank = metadata.my_rank;
    auto comm_size = metadata.comm_size;
    auto num_subdomains = metadata.num_subdomains;
    auto global_size = metadata.global_size;
    auto mpi_itype = boost::mpi::get_mpi_datatype(*partition_indices.data());

    MPI_Bcast(partition_indices.data(), global_size, mpi_itype, 0,
              MPI_COMM_WORLD);

    std::vector<IndexType> local_p_size(num_subdomains);
    auto global_to_local = metadata.global_to_local->get_data();
    auto local_to_global = metadata.local_to_global->get_data();

    auto first_row = metadata.first_row->get_data();
    auto permutation = metadata.permutation->get_data();
    auto i_permutation = metadata.i_permutation->get_data();

    auto nb = (global_size + num_subdomains - 1) /
   num_subdomains;
    auto partition_settings =
        (Settings::partition_settings::partition_zoltan |
         Settings::partition_settings::partition_metis |
         Settings::partition_settings::partition_regular |
         Settings::partition_settings::partition_regular2d |
         Settings::partition_settings::partition_custom) &
        settings.partition;

    IndexType *gmat_row_ptrs = global_matrix->get_row_ptrs();
    IndexType *gmat_col_idxs = global_matrix->get_col_idxs();
    ValueType *gmat_values = global_matrix->get_values();

    // default local p size set for 1 subdomain.
    first_row[0] = 0;
    for (auto p = 0; p < num_subdomains; ++p) {
         local_p_size[p] = std::min(global_size - first_row[p], nb);
         first_row[p + 1] = first_row[p] + local_p_size[p];
     }


     if (partition_settings == Settings::partition_settings::partition_metis ||
         partition_settings ==
             Settings::partition_settings::partition_regular2d) {
         if (num_subdomains > 1) {
             for (auto p = 0; p < num_subdomains; p++) {
                 local_p_size[p] = 0;
             }
             for (auto i = 0; i < global_size; i++) {
                 local_p_size[partition_indices[i]]++;
             }
             first_row[0] = 0;
             for (auto p = 0; p < num_subdomains; ++p) {
                 first_row[p + 1] = first_row[p] + local_p_size[p];
             }
             // permutation
             for (auto i = 0; i < global_size; i++) {
                 permutation[first_row[partition_indices[i]]] = i;
                 first_row[partition_indices[i]]++;
             }
             for (auto p = num_subdomains; p > 0; p--) {
                 first_row[p] = first_row[p - 1];
             }
             first_row[0] = 0;

             // iperm
             for (auto i = 0; i < global_size; i++) {
                 i_permutation[permutation[i]] = i;
             }
         }

         auto gmat_temp = mtx::create(settings.executor->get_master(),
                                      global_matrix->get_size(),
                                      global_matrix->get_num_stored_elements());

         auto nnz = 0;
         gmat_temp->get_row_ptrs()[0] = 0;
         for (auto row = 0; row < metadata.global_size; ++row) {
             for (auto col = gmat_row_ptrs[permutation[row]];
                  col < gmat_row_ptrs[permutation[row] + 1]; ++col) {
                 gmat_temp->get_col_idxs()[nnz] =
                     i_permutation[gmat_col_idxs[col]];
                 gmat_temp->get_values()[nnz] = gmat_values[col];
                 nnz++;
```

```
            }
            gmat_temp->get_row_ptrs()[row + 1] = nnz;
        }
        global_matrix->copy_from(gmat_temp.get());
    }


    for (auto i = 0; i < global_size; i++) {
        global_to_local[i] = 0;
        local_to_global[i] = 0;
    }
    auto num = 0;
    for (auto i = first_row[my_rank]; i < first_row[
  my_rank + 1]; i++) {
        global_to_local[i] = 1 + num;
        local_to_global[num] = i;
        num++;
    }

    IndexType old = 0;
    for (auto k = 1; k < settings.overlap; k++) {
        auto now = num;
        for (auto i = old; i < now; i++) {
            for (auto j = gmat_row_ptrs[local_to_global[i]];
                 j < gmat_row_ptrs[local_to_global[i] + 1]; j++) {
                if (global_to_local[gmat_col_idxs[j]] == 0) {
                    local_to_global[num] = gmat_col_idxs[j];
                    global_to_local[gmat_col_idxs[j]] = 1 + num;
                    num++;
                }
            }
        }
        old = now;
    }
    metadata.local_size = local_p_size[my_rank];
    metadata.local_size_x = num;
    metadata.local_size_o = global_size;
    auto local_size = metadata.local_size;
    auto local_size_x = metadata.local_size_x;

    metadata.overlap_size = num - metadata.local_size;
    metadata.overlap_row = std::shared_ptr<vec_itype>(
        new vec_itype(gko::Array<IndexType>::view(
            settings.executor, metadata.overlap_size,
            &(metadata.local_to_global->get_data()[metadata.local_size]))),
        std::default_delete<vec_itype>());

    auto nnz_local = 0;
    auto nnz_interface = 0;

    for (auto i = first_row[my_rank]; i < first_row[my_rank + 1]; ++i) {
        for (auto j = gmat_row_ptrs[i]; j < gmat_row_ptrs[i + 1]; j++) {
            if (global_to_local[gmat_col_idxs[j]] != 0) {
                nnz_local++;
            } else {
                std::cout << " debug: invalid edge?" << std::endl;
            }
        }
    }
    auto temp = 0;
    for (auto k = 0; k < metadata.overlap_size; k++) {
        temp = metadata.overlap_row->get_data()[k];
        for (auto j = gmat_row_ptrs[temp]; j < gmat_row_ptrs[temp + 1]; j++) {
            if (global_to_local[gmat_col_idxs[j]] != 0) {
                nnz_local++;
            } else {
                nnz_interface++;
            }
        }
    }

    std::shared_ptr<mtx> local_matrix_compute;
    local_matrix_compute = mtx::create(settings.executor->get_master(),
                                       gko::dim<2>(local_size_x), nnz_local);
    IndexType *lmat_row_ptrs = local_matrix_compute->get_row_ptrs();
    IndexType *lmat_col_idxs = local_matrix_compute->get_col_idxs();
    ValueType *lmat_values = local_matrix_compute->get_values();

    std::shared_ptr<mtx> interface_matrix_compute;
    if (nnz_interface > 0) {
        interface_matrix_compute =
            mtx::create(settings.executor->get_master(),
                        gko::dim<2>(local_size_x), nnz_interface);
    } else {
        interface_matrix_compute = mtx::create(settings.executor->get_master());
    }

    IndexType *imat_row_ptrs = interface_matrix_compute->get_row_ptrs();
    IndexType *imat_col_idxs = interface_matrix_compute->get_col_idxs();
    ValueType *imat_values = interface_matrix_compute->get_values();

    num = 0;
    nnz_local = 0;
    auto nnz_interface_temp = 0;
    lmat_row_ptrs[0] = nnz_local;
    if (nnz_interface > 0) {
        imat_row_ptrs[0] = nnz_interface_temp;
    }
    // Local interior matrix
    for (auto i = first_row[my_rank]; i < first_row[my_rank + 1]; ++i) {
        for (auto j = gmat_row_ptrs[i]; j < gmat_row_ptrs[i + 1]; ++j) {
            if (global_to_local[gmat_col_idxs[j]] != 0) {
                lmat_col_idxs[nnz_local] =
                    global_to_local[gmat_col_idxs[j]] - 1;
                lmat_values[nnz_local] = gmat_values[j];
                nnz_local++;
            }
        }
        if (nnz_interface > 0) {
            imat_row_ptrs[num + 1] = nnz_interface_temp;
        }
        lmat_row_ptrs[num + 1] = nnz_local;
        num++;
    }

    // Interface matrix
    if (nnz_interface > 0) {
        nnz_interface = 0;
        for (auto k = 0; k < metadata.overlap_size; k++) {
            temp = metadata.overlap_row->get_data()[k];
            for (auto j = gmat_row_ptrs[temp]; j < gmat_row_ptrs[temp + 1];
                 j++) {
                if (global_to_local[gmat_col_idxs[j]] != 0) {
                    lmat_col_idxs[nnz_local] =
                        global_to_local[gmat_col_idxs[j]] - 1;
                    lmat_values[nnz_local] = gmat_values[j];
                    nnz_local++;
                } else {
                    imat_col_idxs[nnz_interface] = gmat_col_idxs[j];
                    imat_values[nnz_interface] = gmat_values[j];
                    nnz_interface++;
                }
            }
            lmat_row_ptrs[num + 1] = nnz_local;
            imat_row_ptrs[num + 1] = nnz_interface;
            num++;
        }
    }
    auto now = num;
    for (auto i = old; i < now; i++) {
        for (auto j = gmat_row_ptrs[local_to_global[i]];
             j < gmat_row_ptrs[local_to_global[i] + 1]; j++) {
            if (global_to_local[gmat_col_idxs[j]] == 0) {
                local_to_global[num] = gmat_col_idxs[j];
                global_to_local[gmat_col_idxs[j]] = 1 + num;
                num++;
            }
        }
    }

    local_matrix = mtx::create(settings.executor);
    local_matrix->copy_from(gko::lend(local_matrix_compute));
    interface_matrix = mtx::create(settings.executor);
    interface_matrix->copy_from(gko::lend(interface_matrix_compute));

    local_matrix->sort_by_column_index();
    interface_matrix->sort_by_column_index();
}
```

7.16.3.3 setup_windows()

```
template<typename ValueType , typename IndexType >
void schwz::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 schwz::Communicate< ValueType, IndexType >.

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

```
{
    using vec_itype = gko::Array<IndexType>;
    using vec_vtype = gko::matrix::Dense<ValueType>;
    auto num_subdomains = metadata.num_subdomains;
    auto local_size_o = metadata.local_size_o;
    auto neighbors_in = this->comm_struct.neighbors_in->get_data();
    auto global_get = this->comm_struct.global_get->get_data();
    auto neighbors_out = this->comm_struct.neighbors_out->get_data();
    auto global_put = this->comm_struct.global_put->get_data();

    // set displacement for the MPI buffer
    auto get_displacements = this->comm_struct.get_displacements->get_data();
    auto put_displacements = this->comm_struct.put_displacements->get_data();
    {
        std::vector<IndexType> tmp_num_comm_elems(num_subdomains + 1, 0);
        tmp_num_comm_elems[0] = 0;
        for (auto j = 0; j < this->comm_struct.num_neighbors_in; j++) {
            if ((global_get[j])[0] > 0) {
                int p = neighbors_in[j];
                tmp_num_comm_elems[p + 1] = (global_get[j])[0];
            }
        }
        for (auto j = 0; j < num_subdomains; j++) {
            tmp_num_comm_elems[j + 1] += tmp_num_comm_elems[j];
        }

        auto mpi_itype = boost::mpi::get_mpi_datatype(tmp_num_comm_elems[0]);
        MPI_Alltoall(tmp_num_comm_elems.data(), 1, mpi_itype, put_displacements,
                     1, mpi_itype, MPI_COMM_WORLD);
    }

    {
        std::vector<IndexType> tmp_num_comm_elems(num_subdomains + 1, 0);
        tmp_num_comm_elems[0] = 0;
        for (auto j = 0; j < this->comm_struct.num_neighbors_out; j++) {
            if ((global_put[j])[0] > 0) {
                int p = neighbors_out[j];
                tmp_num_comm_elems[p + 1] = (global_put[j])[0];
            }
        }
        for (auto j = 0; j < num_subdomains; j++) {
            tmp_num_comm_elems[j + 1] += tmp_num_comm_elems[j];
        }

        auto mpi_itype = boost::mpi::get_mpi_datatype(tmp_num_comm_elems[0]);
        MPI_Alltoall(tmp_num_comm_elems.data(), 1, mpi_itype, get_displacements,
                     1, mpi_itype, MPI_COMM_WORLD);
    }

    // setup windows
    if (settings.comm_settings.enable_onesided) {
        // Onesided
        MPI_Win_create(main_buffer->get_values(),
                       main_buffer->get_size()[0] * sizeof(ValueType),
                       sizeof(ValueType), MPI_INFO_NULL, MPI_COMM_WORLD,
                       &(this->comm_struct.window_x));
    }


    if (settings.comm_settings.enable_onesided) {
        // MPI_Alloc_mem ? Custom allocator ?  TODO
        MPI_Win_create(this->local_residual_vector->get_values(),
                       (num_subdomains) * sizeof(ValueType), sizeof(ValueType),
                       MPI_INFO_NULL, MPI_COMM_WORLD,
                       &(this->window_residual_vector));
        std::vector<IndexType> zero_vec(num_subdomains, 0);
        gko::Array<IndexType> temp_array{settings.executor->get_master(),
                                         zero_vec.begin(), zero_vec.end()};
        this->convergence_vector = std::shared_ptr<vec_itype>(
            new vec_itype(settings.executor->get_master(), temp_array),
            std::default_delete<vec_itype>());
        this->convergence_sent = std::shared_ptr<vec_itype>(
            new vec_itype(settings.executor->get_master(), num_subdomains),
            std::default_delete<vec_itype>());
        this->convergence_local = std::shared_ptr<vec_itype>(
            new vec_itype(settings.executor->get_master(), num_subdomains),
            std::default_delete<vec_itype>());
        MPI_Win_create(this->convergence_vector->get_data(),
                       (num_subdomains) * sizeof(IndexType), sizeof(IndexType),
                       MPI_INFO_NULL, MPI_COMM_WORLD,
                       &(this->window_convergence));
    }

    if (settings.comm_settings.enable_onesided && num_subdomains > 1) {
        // Lock all windows.
        if (settings.comm_settings.enable_get &&
            settings.comm_settings.enable_lock_all) {
            MPI_Win_lock_all(0, this->comm_struct.window_send_buffer);
            MPI_Win_lock_all(0, this->comm_struct.window_cpu_send_buffer);
        }
        if (settings.comm_settings.enable_put &&
            settings.comm_settings.enable_lock_all) {
            MPI_Win_lock_all(0, this->comm_struct.window_recv_buffer);
            MPI_Win_lock_all(0, this->comm_struct.window_cpu_recv_buffer);
        }
        if (settings.comm_settings.enable_one_by_one &&
            settings.comm_settings.enable_lock_all) {
            MPI_Win_lock_all(0, this->comm_struct.window_x);
        }
        MPI_Win_lock_all(0, this->window_residual_vector);
        MPI_Win_lock_all(0, this->window_convergence);
    }
}
```

7.16.3.4 update_boundary()

```
template<typename ValueType , typename IndexType >
void schwz::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 >> & global_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.
global_solution	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 schwz::Communicate< ValueType, IndexType >.

References schwz::Settings::executor, schwz::SchwarzBase< ValueType, IndexType >::global_solution, schwz::SchwarzBase< ValueType, IndexType >::interface_matrix, schwz::SchwarzBase< ValueType, IndexType >::local_rhs, schwz::Metadata< ValueType, IndexType >::local_size_x, schwz::SchwarzBase< ValueType, IndexType >::local_solution, schwz::Metadata< ValueType, IndexType >::num_subdomains, and schwz::Settings::overlap.

```
{
    using vec_vtype = gko::matrix::Dense<ValueType>;
    auto one = gko::initialize<gko::matrix::Dense<ValueType>>(
        {1.0}, settings.executor);
    auto neg_one = gko::initialize<gko::matrix::Dense<ValueType>>(
        {-1.0}, settings.executor);
    auto local_size_x = metadata.local_size_x;
    local_solution->copy_from(local_rhs.get());
    if (metadata.num_subdomains > 1 && settings.overlap > 0) {
        auto temp_solution = vec_vtype::create(
            settings.executor, local_solution->get_size(),
            gko::Array<ValueType>::view(settings.executor,
                                        local_solution->get_size()[0],
                                        global_solution->get_values()),
            1);
        interface_matrix->apply(neg_one.get(), temp_solution.get(), one.get(),
                                local_solution.get());
    }
}
```

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

- restricted_schwarz.hpp (e0c2959)
- /home/runner/work/schwarz-lib/schwarz-lib/source/restricted_schwarz.cpp (e0c2959)

7.17 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.

7.17.1 Detailed Description

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

7.17.2 Constructor & Destructor Documentation

7.17.2.1 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

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

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

- exception.hpp (e0c2959)
- /home/runner/work/schwarz-lib/schwarz-lib/source/exception.cpp (e0c2959)

7.18 schwz::Utils< ValueType, IndexType > Struct Template Reference

The utilities class which provides some checks and basic utilities.

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

7.18.1 Detailed Description

template<typename ValueType = gko::default_precision, typename IndexType = gko::int32>
struct schwz::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 (e0c2959)
- /home/runner/work/schwarz-lib/schwarz-lib/source/utils.cpp (e0c2959)

Index