rocALUTION Documentation

Release 2.0.2

Advanced Micro Devices

Feb 23, 2023

CONTENTS:

rocALUTION is a sparse linear algebra library with focus on exploring fine-grained parallelism on top of AMD's Radeon Open Compute ROCm runtime and toolchains, targeting modern CPU and GPU platforms. Based on C++ and HIP, it provides a portable, generic and flexible design that allows seamless integration with other scientific software packages.

In the following, three separate chapters are available:

- *User Manual*: This is the manual of rocALUTION. It can be seen as a starting guide for new users but also a reference book for more experienced users.
- *Design Documentation*: The Design Document is targeted to advanced users / developers that want to understand, modify or extend the functionality of the rocALUTION library. To embed rocALUTION into your project, it is not required to read the Design Document.
- *API*: This is a list of API functions provided by rocALUTION.

CHAPTER

ONE

USER MANUAL

1.1 Introduction

1.1.1 Overview

rocALUTION is a sparse linear algebra library with focus on exploring fine-grained parallelism, targeting modern processors and accelerators including multi/many-core CPU and GPU platforms. The main goal of this package is to provide a portable library for iterative sparse methods on state of the art hardware. rocALUTION can be seen as middle-ware between different parallel backends and application specific packages.

The major features and characteristics of the library are

- **Various backends**
  - Host - fallback backend, designed for CPUs
  - GPU/HIP - accelerator backend, designed for HIP capable AMD GPUs
  - OpenMP - designed for multi-core CPUs
  - MPI - designed for multi-node and multi-GPU configurations
- **Easy to use**
  The syntax and structure of the library provide easy learning curves. With the help of the examples, anyone can try out the library - no knowledge in HIP, OpenMP or MPI programming required.
- **No special hardware requirements**
  There are no hardware requirements to install and run rocALUTION. If a GPU device and HIP is available, the library will use them.
- **Variety of iterative solvers**
  - Fixed-Point iteration - Jacobi, Gauss-Seidel, Symmetric-Gauss Seidel, SOR and SSOR
  - Krylov subspace methods - CR, CG, BiCGStab, BiCGStab(l), GMRES, IDR, QMRCGSTAB, Flexible CG/GMRES
  - Mixed-precision defect-correction scheme
  - Chebyshev iteration
  - Multiple MultiGrid schemes, geometric and algebraic
- **Various preconditioners**
  - Matrix splitting - Jacobi, (Multi-colored) Gauss-Seidel, Symmetric Gauss-Seidel, SOR, SSOR
  - Factorization - ILU(0), ILU(p) (based on levels), ILU(p,q) (power(q)-pattern method), Multi-Elimination ILU (nested/recursive), ILUT (based on threshold) and IC(0)

  - Approximate Inverse - Chebyshev matrix-valued polynomial, SPAI, FSAI and TNS
  - Diagonal-based preconditioner for Saddle-point problems
  - Block-type of sub-preconditioners/solvers
  - Additive Schwarz and Restricted Additive Schwarz
  - Variable type preconditioners

- **Generic and robust design**
  rocALUTION is based on a generic and robust design allowing expansion in the direction of new solvers and preconditioners and support for various hardware types. Furthermore, the design of the library allows the use of all solvers as preconditioners in other solvers. For example you can easily define a CG solver with a Multi-Elimination preconditioner, where the last-block is preconditioned with another Chebyshev iteration method which is preconditioned with a multi-colored Symmetric Gauss-Seidel scheme.

- **Portable code and results**
  All code based on rocALUTION is portable and independent of HIP or OpenMP. The code will compile and run everywhere. All solvers and preconditioners are based on a single source code, which delivers portable results across all supported backends (variations are possible due to different rounding modes on the hardware). The only difference which you can see for a hardware change is the performance variation.

- **Support for several sparse matrix formats**
  Compressed Sparse Row (CSR), Modified Compressed Sparse Row (MCSR), Dense (DENSE), Coordinate (COO), ELL, Diagonal (DIA), Hybrid format of ELL and COO (HYB).

The code is open-source under MIT license, see *License* and hosted on the GitHub rocALUTION page.

1.1.2 License

rocALUTION is distributed as open-source under the following license:

1.2 Building and Installing

1.2.1 Installing from AMD ROCm repository

rocALUTION can be installed from AMD ROCm repository. The repository hosts the single-node, accelerator enabled version of the library. If a different setup is required, e.g. multi-node support, rocALUTION need to be built from source, see *Building from GitHub repository*.

For detailed instructions on how to set up ROCm on different platforms, see the AMD ROCm Platform Installation Guide for Linux.

rocALUTION has the following run-time dependencies

- AMD ROCm 2.9 or later (optional, for HIP support)
- rocSPARSE (optional, for HIP support)
- rocBLAS (optional, for HIP support)
- OpenMP (optional, for OpenMP support)
- MPI (optional, for multi-node / multi-GPU support)

1.2.2 Building from GitHub repository

To build rocALUTION from source, the following compile-time and run-time dependencies must be met

- git
- CMake 3.5 or later
- AMD ROCm 2.9 or later (optional, for HIP support)
- rocSPARSE (optional, for HIP support)
- rocBLAS (optional, for HIP support)
- rocPRIM (optional, for HIP support)
- OpenMP (optional, for OpenMP support)
- MPI (optional, for multi-node / multi-GPU support)
- googletest (optional, for clients)

1.2.2.1 Download rocALUTION

The rocALUTION source code is available at the rocALUTION GitHub page. Download the master branch using:

```
$ git clone -b master https://github.com/ROCmSoftwarePlatform/rocALUTION.git
$ cd rocALUTION
```

Below are steps to build different packages of the library, including dependencies and clients. It is recommended to install rocALUTION using the *install.sh* script.

1.2.2.2 Using *install.sh* script to build rocALUTION with dependencies

The following table lists common uses of *install.sh* to build dependencies + library. Accelerator support via HIP and OpenMP will be enabled by default, whereas MPI is disabled.

Command	Description
./install.sh -h	Print help information.
./install.sh -d	Build dependencies and library in your local directory. The *-d* flag only needs to be used once. For subsequent invocations of *install.sh* it is not necessary to rebuild the dependencies.
./install.sh	Build library in your local directory. It is assumed dependencies are available.
./install.sh -i	Build library, then build and install rocALUTION package in */opt/rocm/rocalution*. You will be prompted for sudo access. This will install for all users.
./install.sh –host	Build library in your local directory without HIP support. It is assumed dependencies are available.
./install.sh –mpi=<dir>	Build library in your local directory with HIP and MPI support. It is assumed dependencies are available.

1.2.2.3 Using *install.sh* script to build rocALUTION with dependencies and clients

The client contains example code, unit tests and benchmarks. Common uses of *install.sh* to build them are listed in the table below.

Command	Description
./install.sh -h	Print help information.
./install.sh -dc	Build dependencies, library and client in your local directory. The *-d* flag only needs to be used once. For subsequent invocations of *install.sh* it is not necessary to rebuild the dependencies.
./install.sh -c	Build library and client in your local directory. It is assumed dependencies are available.
./install.sh -idc	Build library, dependencies and client, then build and install rocALUTION package in */opt/rocm/rocalution*. You will be prompted for sudo access. This will install for all users.
./install.sh -ic	Build library and client, then build and install rocALUTION package in *opt/rocm/rocalution*. You will be prompted for sudo access. This will install for all users.

1.2.2.4 Using individual commands to build rocALUTION

CMake 3.5 or later is required in order to build rocALUTION without the use of *install.sh*.

rocALUTION can be built with cmake using the following commands:

```
# Create and change to build directory
mkdir -p build/release ; cd build/release

# Default install path is /opt/rocm, use -DCMAKE_INSTALL_PREFIX=<path>
# to adjust it. In this case, rocALUTION is built with HIP and
# OpenMP support.
# MPI support is disabled.
cmake ../.. -DSUPPORT_HIP=ON \
            -DSUPPORT_MPI=OFF \
            -DSUPPORT_OMP=ON

# Compile rocALUTION library
make -j$(nproc)
```

(continues on next page)

(continued from previous page)

```
# Install rocALUTION to /opt/rocm
sudo make install
```

GoogleTest is required in order to build all rocALUTION clients.

rocALUTION with dependencies and clients can be built using the following commands:

```
# Install googletest
mkdir -p build/release/deps ; cd build/release/deps
cmake ../../../deps
sudo make -j$(nproc) install

# Change to build directory
cd ..

# Default install path is /opt/rocm, use -DCMAKE_INSTALL_PREFIX=<path>
# to adjust it. By default, HIP and OpenMP support are enabled,
# MPI support is disabled.
cmake ../.. -DBUILD_CLIENTS_TESTS=ON \
            -DBUILD_CLIENTS_SAMPLES=ON

# Compile rocALUTION library
make -j$(nproc)

# Install rocALUTION to /opt/rocm
sudo make install
```

The compilation process produces a shared library file *librocalution.so* and *librocalution_hip.so* if HIP support is enabled. Ensure that the library objects can be found in your library path. If you do not copy the library to a specific location you can add the path under Linux in the *LD_LIBRARY_PATH* variable.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<path_to_rocalution>
```

1.2.2.5 Common build problems

1. **Issue: Could not find a package file provided by "ROCM" with any of the following names:**
   ROCMConfig.cmake rocm-config.cmake

   Solution: Install ROCm cmake modules either from source or from AMD ROCm repository.

2. **Issue: Could not find a package file provided by "ROCSPARSE" with any of the following names:**
   ROCSPARSE.cmake rocsparse-config.cmake

   Solution: Install rocSPARSE either from source or from AMD ROCm repository.

3. **Issue: Could not find a package file provided by "ROCBLAS" with any of the following names:**
   ROCBLAS.cmake rocblas-config.cmake

   Solution: Install rocBLAS either from source or from AMD ROCm repository.

1.2.2.6 Simple Test

You can test the installation by running a CG solver on a sparse matrix. After successfully compiling the library, the CG solver example can be executed.

```
cd rocALUTION/build/release/clients/staging

wget ftp://math.nist.gov/pub/MatrixMarket2/Harwell-Boeing/laplace/gr_30_30.mtx.gz
gzip -d gr_30_30.mtx.gz

./cg gr_30_30.mtx
```

1.3 Basics

1.3.1 Operators and Vectors

The main objects in rocALUTION are linear operators and vectors. All objects can be moved to an accelerator at run time. The linear operators are defined as local or global matrices (i.e. on a single node or distributed/multi-node) and local stencils (i.e. matrix-free linear operations). The only template parameter of the operators and vectors is the data type (ValueType). The operator data type could be float, double, complex float or complex double, while the vector data type can be int, float, double, complex float or complex double (int is used mainly for the permutation vectors). In the current version, cross ValueType object operations are not supported. Fig. 1.1 gives an overview of supported operators and vectors. Further details are also given in the *Design Documentation*.

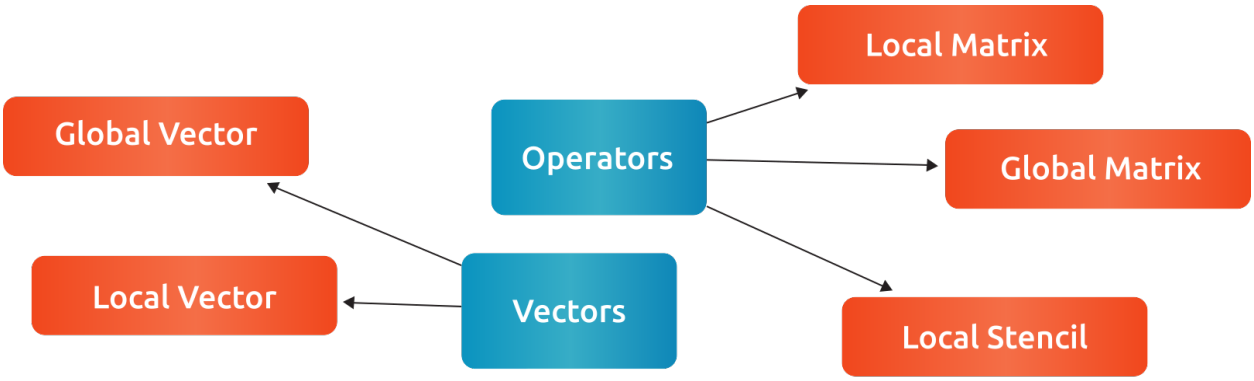

Fig. 1.1: Operator and vector classes.

Each of the objects contain a local copy of the hardware descriptor created by the *rocalution::init_rocalution()* function. This allows the user to modify it according to his needs and to obtain two or more objects with different hardware specifications (e.g. different amount of OpenMP threads, HIP block sizes, etc.).

1.3.1.1 Local Operators and Vectors

By Local Operators and Vectors we refer to Local Matrices and Stencils and to Local Vectors. By Local we mean the fact that they stay on a single system. The system can contain several CPUs via UMA or NUMA memory system, it can also contain an accelerator.

template<typename **ValueType**>

class **LocalMatrix** : public rocalution::*Operator*<*ValueType*>

LocalMatrix class.

A *LocalMatrix* is called local, because it will always stay on a single system. The system can contain several CPUs via UMA or NUMA memory system or it can contain an accelerator.

A number of matrix formats are supported. These are CSR, BCSR, MCSR, COO, DIA, ELL, HYB, and DENSE.

Note: For CSR type matrices, the column indices must be sorted in increasing order. For COO matrices, the row indices must be sorted in increasing order. The function `Check` can be used to check whether a matrix contains valid data. For CSR and COO matrices, the function `Sort` can be used to sort the row or column indices respectively.

Template Parameters

ValueType – - can be int, float, double, std::complex<float> and std::complex<double>

template<typename **ValueType**>

class **LocalStencil** : public rocalution::*Operator*<*ValueType*>

LocalStencil class.

A *LocalStencil* is called local, because it will always stay on a single system. The system can contain several CPUs via UMA or NUMA memory system or it can contain an accelerator.

Template Parameters

ValueType – - can be int, float, double, std::complex<float> and std::complex<double>

template<typename **ValueType**>

class **LocalVector** : public rocalution::*Vector*<*ValueType*>

LocalVector class.

A *LocalVector* is called local, because it will always stay on a single system. The system can contain several CPUs via UMA or NUMA memory system or it can contain an accelerator.

Template Parameters

ValueType – - can be int, float, double, std::complex<float> and std::complex<double>

1.3.1.2 Global Operators and Vectors

By Global Operators and Vectors we refer to Global Matrix and to Global Vectors. By Global we mean the fact they can stay on a single or multiple nodes in a network. For this type of computation, the communication is based on MPI.

template<typename **ValueType**>

class **GlobalMatrix** : public rocalution::*Operator*<*ValueType*>

GlobalMatrix class.

A *GlobalMatrix* is called global, because it can stay on a single or on multiple nodes in a network. For this type of communication, MPI is used.

A number of matrix formats are supported. These are CSR, BCSR, MCSR, COO, DIA, ELL, HYB, and DENSE.

Note: For CSR type matrices, the column indices must be sorted in increasing order. For COO matrices, the row indices must be sorted in increasing order. The function `Check` can be used to check whether a matrix contains valid data. For CSR and COO matrices, the function `Sort` can be used to sort the row or column indices respectively.

Template Parameters

ValueType – - can be int, float, double, std::complex<float> and std::complex<double>

template<typename **ValueType**>

class **GlobalVector** : public rocalution::*Vector*<*ValueType*>

GlobalVector class.

A *GlobalVector* is called global, because it can stay on a single or on multiple nodes in a network. For this type of communication, MPI is used.

Template Parameters

ValueType – - can be int, float, double, std::complex<float> and std::complex<double>

1.3.2 Backend Descriptor and User Control

Naturally, not all routines and algorithms can be performed efficiently on many-core systems (i.e. on accelerators). To provide full functionality, the library has internal mechanisms to check if a particular routine is implemented on the accelerator. If not, the object is moved to the host and the routine is computed there. This guarantees that your code will run (maybe not in the most efficient way) with any accelerator regardless of the available functionality for it.

1.3.2.1 Initialization of rocALUTION

The body of a rocALUTION code is very simple, it should contain the header file and the namespace of the library. The program must contain an initialization call to *init_rocalution* which will check and allocate the hardware and a finalizing call to *stop_rocalution* which will release the allocated hardware.

int `rocalution::`**init_rocalution**(int rank = -1, int dev_per_node = 1)

Initialize rocALUTION platform.

`init_rocalution` defines a backend descriptor with information about the hardware and its specifications. All objects created after that contain a copy of this descriptor. If the specifications of the global descriptor are

changed (e.g. set different number of threads) and new objects are created, only the new objects will use the new configurations.

For control, the library provides the following functions

- *set_device_rocalution()* is a unified function to select a specific device. If you have compiled the library with a backend and for this backend there are several available devices, you can use this function to select a particular one. This function has to be called before *init_rocalution()*.
- *set_omp_threads_rocalution()* sets the number of OpenMP threads. This function has to be called after *init_rocalution()*.

Example

```
#include <rocalution/rocalution.hpp>

using namespace rocalution;

int main(int argc, char* argv[])
{
    init_rocalution();

    // ...

    stop_rocalution();

    return 0;
}
```

Parameters

- **rank** – **[in]** specifies MPI rank when multi-node environment
- **dev_per_node** – **[in]** number of accelerator devices per node, when in multi-GPU environment

int `rocalution::`**stop_rocalution**(void)

Shutdown rocALUTION platform.

`stop_rocalution` shuts down the rocALUTION platform.

1.3.2.2 Thread-core Mapping

The number of threads which rocALUTION will use can be modified by the function `set_omp_threads_rocalution` or by the global OpenMP environment variable (for Unix-like OS this is *OMP_NUM_THREADS*). During the initialization phase, the library provides affinity thread-core mapping:

- If the number of cores (including SMT cores) is greater or equal than two times the number of threads, then all the threads can occupy every second core ID (e.g. 0,2,4,...). This is to avoid having two threads working on the same physical core, when SMT is enabled.
- If the number of threads is less or equal to the number of cores (including SMT), and the previous clause is false, then the threads can occupy every core ID (e.g. 0,1,2,3,...).
- If non of the above criteria is matched, then the default thread-core mapping is used (typically set by the operating system).

Note: The thread-core mapping is available for Unix-like operating systems only.

Note: The user can disable the thread affinity by *set_omp_affinity_rocalution*, before initializing the library.

1.3.2.3 OpenMP Threshold Size

Whenever working on a small problem, OpenMP host backend might be slightly slower than using no OpenMP. This is mainly attributed to the small amount of work, which every thread should perform and the large overhead of forking/joining threads. This can be avoid by the OpenMP threshold size parameter in rocALUTION. The default threshold is set to 10.000, which means that all matrices under (and equal to) this size will use only one thread (disregarding the number of OpenMP threads set in the system). The threshold can be modified with *set_omp_threshold_rocalution*.

1.3.2.4 Accelerator Selection

The accelerator device id that is supposed to be used for the computation can be selected by the user by *set_device_rocalution*.

1.3.2.5 Disable the Accelerator

Furthermore, the accelerator can be disabled without having to re-compile the library by calling *disable_accelerator_rocalution*.

1.3.2.6 Backend Information

Detailed information about the current backend / accelerator in use as well as the available accelerators can be printed by *info_rocalution*.

1.3.2.7 MPI and Multi-Accelerators

When initializing the library with MPI, the user need to pass the rank of the MPI process as well as the number of accelerators available on each node. Basically, this way the user can specify the mapping of MPI process and accelerators - the allocated accelerator will be *rank % num_dev_per_node*. Thus, the user can run two MPI processes on systems with two accelerators by specifying the number of devices to 2, as illustrated in the example code below.

```
#include <rocalution.hpp>
#include <mpi.h>

using namespace rocalution;

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Comm comm = MPI_COMM_WORLD;

    int num_processes;
```

(continues on next page)

(continued from previous page)

```
    int rank;

    MPI_Comm_size(comm, &num_processes);
    MPI_Comm_rank(comm, &rank);

    int nacc_per_node = 2;

    init_rocalution(rank, nacc_per_node);

    // ... do some work

    stop_rocalution();

    return 0;
}
```

1.3.3 Automatic Object Tracking

rocALUTION supports automatic object tracking. After the initialization of the library, all objects created by the user application can be tracked. Once *stop_rocalution* is called, all memory from tracked objects gets deallocated. This will avoid memory leaks when the objects are allocated but not freed. The user can enable or disable the tracking by editing *src/utils/def.hpp*. By default, automatic object tracking is disabled.

1.3.4 Verbose Output

rocALUTION provides different levels of output messages. The *VERBOSE_LEVEL* can be modified in *src/utils/def.hpp* before the compilation of the library. By setting a higher level, the user will obtain more detailed information about the internal calls and data transfers to and from the accelerators. By default, *VERBOSE_LEVEL* is set to 2.

1.3.5 Verbose Output and MPI

To prevent all MPI processes from printing information to *stdout*, the default configuration is that only *RANK 0* outputs information. The user can change the *RANK* or allow all processes to print setting *LOG_MPI_RANK* to 1 in *src/utils/def.hpp*. If file logging is enabled, all ranks write into the corresponding log files.

1.3.6 Debug Output

Debug output will print almost every detail in the program, including object constructor / destructor, address of the object, memory allocation, data transfers, all function calls for matrices, vectors, solvers and preconditioners. The flag *DEBUG_MODE* can be set in *src/utils/def.hpp*. When enabled, additional *assert()s* are being checked during the computation. This might decrease performance of some operations significantly.

1.3.7 File Logging

rocALUTION trace file logging can be enabled by setting the environment variable *ROCALUTION_LAYER* to 1. rocALUTION will then log each rocALUTION function call including object constructor / destructor, address of the object, memory allocation, data transfers, all function calls for matrices, vectors, solvers and preconditioners. The log file will be placed in the working directory. The log file naming convention is *rocalution-rank-<rank>-<time_since_epoch_in_msec>.log*. By default, the environment variable *ROCALUTION_LAYER* is unset, and logging is disabled.

Note: Performance might degrade when logging is enabled.

1.3.8 Versions

For checking the rocALUTION version in an application, pre-defined macros can be used:

```
#define __ROCALUTION_VER_MAJOR  // version major
#define __ROCALUTION_VER_MINOR  // version minor
#define __ROCALUTION_VER_PATCH  // version patch
#define __ROCALUTION_VER_TWEAK  // commit id (sha-1)

#define __ROCALUTION_VER_PRE    // version pre-release (alpha or beta)

#define __ROCALUTION_VER        // version
```

The final *__ROCALUTION_VER* holds the version number as *10000 * major + 100 * minor + patch*, as defined in *src/base/version.hpp.in*.

1.4 Single-node Computation

1.4.1 Introduction

In this chapter, all base objects (matrices, vectors and stencils) for computation on a single-node (shared-memory) system are described. A typical configuration is illustrated in Fig. 1.2.

The compute node contains none, one or more accelerators. The compute node could be any kind of shared-memory (single, dual, quad CPU) system.

Note: The host and accelerator memory can be physically different.

1.4.2 ValueType

The value (data) type of the vectors and the matrices is defined as a template. The matrix can be of type float (32-bit), double (64-bit) and complex (64/128-bit). The vector can be float (32-bit), double (64-bit), complex (64/128-bit) and int (32/64-bit). The information about the precision of the data type is shown in the `rocalution::BaseRocalution::Info()` function.

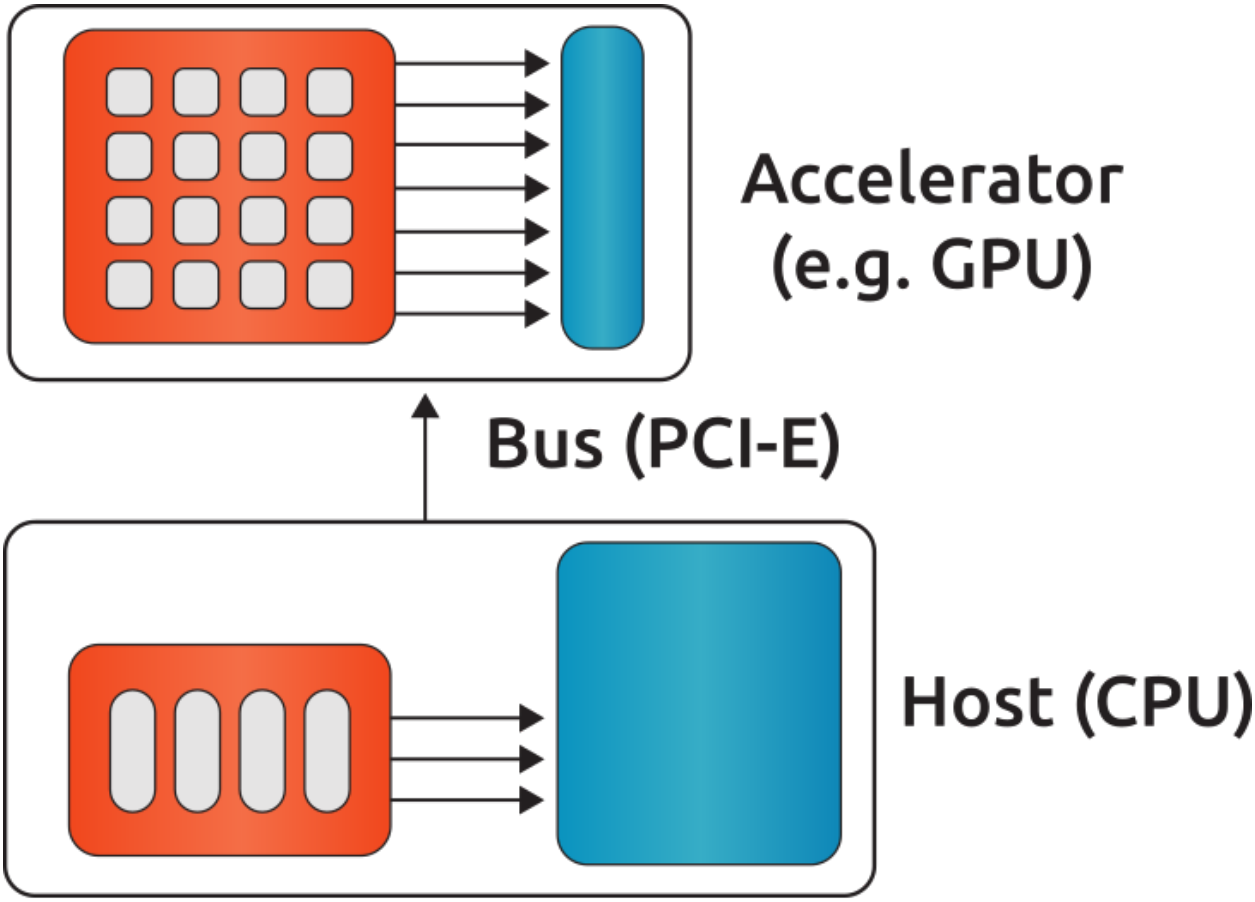

Fig. 1.2: A typical single-node configuration, where gray boxes represent the cores, blue boxes represent the memory and arrows represent the bandwidth.

1.4.3 Complex Support

Currently, rocALUTION does not support complex computation.

1.4.4 Allocation and Free

void rocalution::*LocalVector*::**Allocate**(std::string name, int64_t size)

Allocate a local vector with name and size.

The local vector allocation function requires a name of the object (this is only for information purposes) and corresponding size description for vector objects.

Example

```
LocalVector<ValueType> vec;

vec.Allocate("my vector", 100);
vec.Clear();
```

Parameters

- **name** – **[in]** object name
- **size** – **[in]** number of elements in the vector

virtual void rocalution::*LocalVector*::**Clear**()

Clear (free all data) the object.

void rocalution::*LocalMatrix*::**AllocateCOO**(const std::string &name, int64_t nnz, int64_t nrow, int64_t ncol)

void rocalution::*LocalMatrix*::**AllocateCSR**(const std::string &name, int64_t nnz, int64_t nrow, int64_t ncol)

void rocalution::*LocalMatrix*::**AllocateBCSR**(const std::string &name, int64_t nnzb, int64_t nrowb, int64_t ncolb, int blockdim)

void rocalution::*LocalMatrix*::**AllocateMCSR**(const std::string &name, int64_t nnz, int64_t nrow, int64_t ncol)

void rocalution::*LocalMatrix*::**AllocateELL**(const std::string &name, int64_t nnz, int64_t nrow, int64_t ncol, int max_row)

void rocalution::*LocalMatrix*::**AllocateDIA**(const std::string &name, int64_t nnz, int64_t nrow, int64_t ncol, int ndiag)

void rocalution::*LocalMatrix*::**AllocateHYB**(const std::string &name, int64_t ell_nnz, int64_t coo_nnz, int ell_max_row, int64_t nrow, int64_t ncol)

void rocalution::*LocalMatrix*::**AllocateDENSE**(const std::string &name, int64_t nrow, int64_t ncol)

Allocate a local matrix with name and sizes.

The local matrix allocation functions require a name of the object (this is only for information purposes) and corresponding number of non-zero elements, number of rows and number of columns. Furthermore, depending on the matrix format, additional parameters are required.

Example

```
LocalMatrix<ValueType> mat;

mat.AllocateCSR("my CSR matrix", 456, 100, 100);
mat.Clear();

mat.AllocateCOO("my COO matrix", 200, 100, 100);
mat.Clear();
```

Note: More detailed information on the additional parameters required for matrix allocation is given in *Matrix Formats*.

virtual void rocalution::*LocalMatrix*::**Clear**(void)

Clear (free all data) the object.

1.4.5 Matrix Formats

Matrices, where most of the elements are equal to zero, are called sparse. In most practical applications, the number of non-zero entries is proportional to the size of the matrix (e.g. typically, if the matrix $A \in \mathbb{R}^{N \times N}$, then the number of elements are of order $O(N)$). To save memory, storing zero entries can be avoided by introducing a structure corresponding to the non-zero elements of the matrix. rocALUTION supports sparse CSR, MCSR, COO, ELL, DIA, HYB and dense matrices (DENSE).

Note: The functionality of every matrix object is different and depends on the matrix format. The CSR format provides the highest support for various functions. For a few operations, an internal conversion is performed, however, for many routines an error message is printed and the program is terminated.

Note: In the current version, some of the conversions are performed on the host (disregarding the actual object allocation - host or accelerator).

```
// Convert mat to CSR storage format
mat.ConvertToCSR();
// Perform a matrix-vector multiplication y = mat * x in CSR format
mat.Apply(x, &y);

// Convert mat to ELL storage format
mat.ConvertToELL();
// Perform a matrix-vector multiplication y = mat * x in ELL format
mat.Apply(x, &y);
```

```
// Convert mat to CSR storage format
mat.ConvertTo(CSR);
// Perform a matrix-vector multiplication y = mat * x in CSR format
mat.Apply(x, &y);
```

(continues on next page)

(continued from previous page)

```
// Convert mat to ELL storage format
mat.ConvertTo(ELL);
// Perform a matrix-vector multiplication y = mat * x in ELL format
mat.Apply(x, &y);
```

1.4.5.1 COO storage format

The most intuitive sparse format is the coordinate format (COO). It represents the non-zero elements of the matrix by their coordinates and requires two index arrays (one for row and one for column indexing) and the values array. A $m \times n$ matrix is represented by

m	number of rows (integer).
n	number of columns (integer).
nnz	number of non-zero elements (integer).
coo_val	array of `nnz` elements containing the data (floating point).
coo_row_ind	array of `nnz` elements containing the row indices (integer).
coo_col_ind	array of `nnz` elements containing the column indices (integer).

Note: The COO matrix is expected to be sorted by row indices and column indices per row. Furthermore, each pair of indices should appear only once.

Consider the following 3×5 matrix and the corresponding COO structures, with $m = 3, n = 5$ and nnz $= 8$:

$$A = \begin{pmatrix} 1.0 & 2.0 & 0.0 & 3.0 & 0.0 \\ 0.0 & 4.0 & 5.0 & 0.0 & 0.0 \\ 6.0 & 0.0 & 0.0 & 7.0 & 8.0 \end{pmatrix}$$

where

$$\begin{aligned} \text{coo\_val}[8] &= \{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0\} \\ \text{coo\_row\_ind}[8] &= \{0, 0, 0, 1, 1, 2, 2, 2\} \\ \text{coo\_col\_ind}[8] &= \{0, 1, 3, 1, 2, 0, 3, 4\} \end{aligned}$$

1.4.5.2 CSR storage format

One of the most popular formats in many scientific codes is the compressed sparse row (CSR) format. In this format, instead of row indices, the row offsets to the beginning of each row are stored. Thus, each row elements can be accessed sequentially. However, this format does not allow sequential accessing of the column entries. The CSR storage format represents a $m \times n$ matrix by

m	number of rows (integer).
n	number of columns (integer).
nnz	number of non-zero elements (integer).
csr_val	array of `nnz` elements containing the data (floating point).
csr_row_ptr	array of `m+1` elements that point to the start of every row (integer).
csr_col_ind	array of `nnz` elements containing the column indices (integer).

Note: The CSR matrix is expected to be sorted by column indices within each row. Furthermore, each pair of indices should appear only once.

Consider the following 3×5 matrix and the corresponding CSR structures, with $m = 3, n = 5$ and nnz $= 8$:

$$A = \begin{pmatrix} 1.0 & 2.0 & 0.0 & 3.0 & 0.0 \\ 0.0 & 4.0 & 5.0 & 0.0 & 0.0 \\ 6.0 & 0.0 & 0.0 & 7.0 & 8.0 \end{pmatrix}$$

where

$$\begin{aligned} \text{csr\_val}[8] &= \{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0\} \\ \text{csr\_row\_ptr}[4] &= \{0, 3, 5, 8\} \\ \text{csr\_col\_ind}[8] &= \{0, 1, 3, 1, 2, 0, 3, 4\} \end{aligned}$$

1.4.5.3 BCSR storage format

The Block Compressed Sparse Row (BCSR) storage format represents a $(mb \cdot \text{bcsr\_dim}) \times (nb \cdot \text{bcsr\_dim})$ matrix by

mb	number of block rows (integer)
nb	number of block columns (integer)
nnzb	number of non-zero blocks (integer)
bcsr_val	array of `nnzb * bcsr_dim * bcsr_dim` elements containing the data (floating point). Data within each block is stored in column-major.
bcsr_row_ptr	array of `mb+1` elements that point to the start of every block row (integer).
bcsr_col_ind	array of `nnzb` elements containing the block column indices (integer).
bcsr_dim	dimension of each block (integer).

The BCSR matrix is expected to be sorted by column indices within each row. If m or n are not evenly divisible by the block dimension, then zeros are padded to the matrix, such that $mb = (m + \text{bcsr\_dim} - 1)/\text{bcsr\_dim}$ and $nb = (n + \text{bcsr\_dim} - 1)/\text{bcsr\_dim}$. Consider the following 4×3 matrix and the corresponding BCSR structures, with bcsr_dim $= 2, mb = 2, nb = 2$ and nnzb $= 4$ using zero based indexing and column-major storage:

$$A = \begin{pmatrix} 1.0 & 0.0 & 2.0 \\ 3.0 & 0.0 & 4.0 \\ 5.0 & 6.0 & 0.0 \\ 7.0 & 0.0 & 8.0 \end{pmatrix}$$

with the blocks A_{ij}

$$A_{00} = \begin{pmatrix} 1.0 & 0.0 \\ 3.0 & 0.0 \end{pmatrix}, A_{01} = \begin{pmatrix} 2.0 & 0.0 \\ 4.0 & 0.0 \end{pmatrix}, A_{10} = \begin{pmatrix} 5.0 & 6.0 \\ 7.0 & 0.0 \end{pmatrix}, A_{11} = \begin{pmatrix} 0.0 & 0.0 \\ 8.0 & 0.0 \end{pmatrix}$$

such that

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}$$

with arrays representation

$$\begin{aligned} \text{bcsr\_val}[16] &= \{1.0, 3.0, 0.0, 0.0, 2.0, 4.0, 0.0, 0.0, 5.0, 7.0, 6.0, 0.0, 0.0, 8.0, 0.0, 0.0\} \\ \text{bcsr\_row\_ptr}[3] &= \{0, 2, 4\} \\ \text{bcsr\_col\_ind}[4] &= \{0, 1, 0, 1\} \end{aligned}$$

1.4.5.4 ELL storage format

The Ellpack-Itpack (ELL) storage format can be seen as a modification of the CSR format without row offset pointers. Instead, a fixed number of elements per row is stored. It represents a $m \times n$ matrix by

m	number of rows (integer).
n	number of columns (integer).
ell_width	maximum number of non-zero elements per row (integer)
ell_val	array of `m times ell_width` elements containing the data (floating point).
ell_col_ind	array of `m times ell_width` elements containing the column indices (integer).

Note: The ELL matrix is assumed to be stored in column-major format. Rows with less than `ell_width` non-zero elements are padded with zeros (`ell_val`) and -1 (`ell_col_ind`).

Consider the following 3×5 matrix and the corresponding ELL structures, with $m = 3, n = 5$ and ell_width $= 3$:

$$A = \begin{pmatrix} 1.0 & 2.0 & 0.0 & 3.0 & 0.0 \\ 0.0 & 4.0 & 5.0 & 0.0 & 0.0 \\ 6.0 & 0.0 & 0.0 & 7.0 & 8.0 \end{pmatrix}$$

where

$$\begin{aligned} \text{ell\_val}[9] &= \{1.0, 4.0, 6.0, 2.0, 5.0, 7.0, 3.0, 0.0, 8.0\} \\ \text{ell\_col\_ind}[9] &= \{0, 1, 0, 1, 2, 3, 3, -1, 4\} \end{aligned}$$

1.4.5.5 DIA storage format

If all (or most) of the non-zero entries belong to a few diagonals of the matrix, they can be stored with the corresponding offsets. The values in DIA format are stored as array with size $D \times N_D$, where D is the number of diagonals in the matrix and N_D is the number of elements in the main diagonal. Since not all values in this array are occupied, the not accessible entries are denoted with $*$. They correspond to the offsets in the diagonal array (negative values represent offsets from the beginning of the array). The DIA storage format represents a $m \times n$ matrix by

m	number of rows (integer)
n	number of columns (integer)
ndiag	number of occupied diagonals (integer)
dia_offset	array of `ndiag` elements containing the offset with respect to the main diagonal (integer).
dia_val	array of `m times ndiag` elements containing the values (floating point).

Consider the following 5×5 matrix and the corresponding DIA structures, with $m = 5, n = 5$ and ndiag $= 4$:

$$A = \begin{pmatrix} 1 & 2 & 0 & 11 & 0 \\ 0 & 3 & 4 & 0 & 0 \\ 0 & 5 & 6 & 7 & 0 \\ 0 & 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 9 & 10 \end{pmatrix}$$

where

$$\begin{aligned} \text{dia\_val}[20] &= \{*, 0, 5, 0, 9, 1, 3, 6, 8, 10, 2, 4, 7, 0, *, 11, 0, *, *, *\} \\ \text{dia\_offset}[4] &= \{-1, 0, 1, 3\} \end{aligned}$$

1.4.5.6 HYB storage format

The DIA and ELL formats cannot represent efficiently completely unstructured sparse matrices. To keep the memory footprint low, DIA requires the elements to belong to a few diagonals and ELL needs a fixed number of elements per row. For many applications this is a too strong restriction. A solution to this issue is to represent the more regular part of the matrix in such a format and the remaining part in COO format. The HYB format is a mixture between ELL and COO, where the maximum elements per row for the ELL part is computed by *nnz/m*. It represents a $m \times n$ matrix by

m	number of rows (integer).
n	number of columns (integer).
nnz	number of non-zero elements of the COO part (integer)
ell_width	maximum number of non-zero elements per row of the ELL part (integer)
ell_val	array of `m times ell_width` elements containing the ELL part data (floating point).
ell_col_ind	array of `m times ell_width` elements containing the ELL part column indices (integer).
coo_val	array of `nnz` elements containing the COO part data (floating point).
coo_row_ind	array of `nnz` elements containing the COO part row indices (integer).
coo_col_ind	array of `nnz` elements containing the COO part column indices (integer).

1.4.5.7 Memory Usage

The memory footprint of the different matrix formats is presented in the following table, considering a $N \times N$ matrix, where the number of non-zero entries is denoted with *nnz*.

Format	Structure	Values
DENSE		$N \times N$
COO	$2 \times$ nnz	nnz
CSR	$N + 1 +$ nnz	nnz
ELL	$M \times N$	$M \times N$
DIA	D	$D \times N_D$

For the ELL matrix M characterizes the maximal number of non-zero elements per row and for the DIA matrix, D defines the number of diagonals and N_D defines the size of the main diagonal.

1.4.6 File I/O

virtual void `rocalution::LocalVector::ReadFileASCII`(const std::string &filename)

Read vector from ASCII file.

Read a vector from ASCII file.

Example

```
LocalVector<ValueType> vec;
vec.ReadFileASCII("my_vector.dat");
```

Parameters

`filename` – **[in]** name of the file containing the ASCII data.

virtual void rocalution::*LocalVector*::**WriteFileASCII**(const std::string &filename) const

Write vector to ASCII file.

Write a vector to ASCII file.

Example

```
LocalVector<ValueType> vec;

// Allocate and fill vec
// ...

vec.WriteFileASCII("my_vector.dat");
```

Parameters

filename – **[in]** name of the file to write the ASCII data to.

virtual void rocalution::*LocalVector*::**ReadFileBinary**(const std::string &filename)

Read vector from binary file.

Read a vector from binary file. For details on the format, see *WriteFileBinary()*.

Example

```
LocalVector<ValueType> vec;
vec.ReadFileBinary("my_vector.bin");
```

Parameters

filename – **[in]** name of the file containing the data.

virtual void rocalution::*LocalVector*::**WriteFileBinary**(const std::string &filename) const

Write vector to binary file.

Write a vector to binary file.

The binary format contains a header, the rocALUTION version and the vector data as follows

```
// Header
out << "#rocALUTION binary vector file" << std::endl;

// rocALUTION version
out.write((char*)&version, sizeof(int));

// Vector data
out.write((char*)&size, sizeof(int));
out.write((char*)vec_val, size * sizeof(double));
```

Example

```
LocalVector<ValueType> vec;

// Allocate and fill vec
// ...

vec.WriteFileBinary("my_vector.bin");
```

Note: *Vector* values array is always stored in double precision (e.g. double or std::complex<double>).

Parameters
filename – **[in]** name of the file to write the data to.

void rocalution::*LocalMatrix*::**ReadFileMTX**(const std::string &filename)

Read matrix from MTX (Matrix Market Format) file.

Read a matrix from Matrix Market Format file.

Example

```
LocalMatrix<ValueType> mat;
mat.ReadFileMTX("my_matrix.mtx");
```

Parameters
filename – **[in]** name of the file containing the MTX data.

void rocalution::*LocalMatrix*::**WriteFileMTX**(const std::string &filename) const

Write matrix to MTX (Matrix Market Format) file.

Write a matrix to Matrix Market Format file.

Example

```
LocalMatrix<ValueType> mat;

// Allocate and fill mat
// ...

mat.WriteFileMTX("my_matrix.mtx");
```

Parameters
filename – **[in]** name of the file to write the MTX data to.

void rocalution::*LocalMatrix*::**ReadFileCSR**(const std::string &filename)

Read matrix from CSR (rocALUTION binary format) file.

Read a CSR matrix from binary file. For details on the format, see *WriteFileCSR()*.

Example

```
LocalMatrix<ValueType> mat;
mat.ReadFileCSR("my_matrix.csr");
```

Parameters

filename – **[in]** name of the file containing the data.

void rocalution::*LocalMatrix*::**WriteFileCSR**(const std::string &filename) const

Write CSR matrix to binary file.

Write a CSR matrix to binary file.

The binary format contains a header, the rocALUTION version and the matrix data as follows

```
// Header
out << "#rocALUTION binary csr file" << std::endl;

// rocALUTION version
out.write((char*)&version, sizeof(int));

// CSR matrix data
out.write((char*)&m, sizeof(int));
out.write((char*)&n, sizeof(int));
out.write((char*)&nnz, sizeof(int64_t));
out.write((char*)csr_row_ptr, (m + 1) * sizeof(int));
out.write((char*)csr_col_ind, nnz * sizeof(int));
out.write((char*)csr_val, nnz * sizeof(double));
```

Example

```
LocalMatrix<ValueType> mat;

// Allocate and fill mat
// ...

mat.WriteFileCSR("my_matrix.csr");
```

Note: *Vector* values array is always stored in double precision (e.g. double or std::complex<double>).

Parameters

filename – **[in]** name of the file to write the data to.

1.4.7 Access

Warning: doxygenfunction: Unable to resolve function "rocalution::LocalVector::operator[]" with arguments (int) in doxygen xml output for project "rocALUTION" from directory: ../docBin/xml. Potential matches:

```
- ValueType &operator[](int64_t i)
- const ValueType &operator[](int64_t i) const
```

Warning: doxygenfunction: Unable to resolve function "rocalution::LocalVector::operator[]" with arguments (int) const in doxygen xml output for project "rocALUTION" from directory: ../docBin/xml. Potential matches:

```
- ValueType &operator[](int64_t i)
- const ValueType &operator[](int64_t i) const
```

Note: Accessing elements via the *[]* operators is slow. Use this for debugging purposes only. There is no direct access to the elements of matrices due to the sparsity structure. Matrices can be imported by a copy function. For CSR matrices, this is *rocalution::LocalMatrix::CopyFromCSR()* and *rocalution::LocalMatrix::CopyToCSR()*.

```
// Allocate the CSR matrix
int* csr_row_ptr   = new int[100 + 1];
int* csr_col_ind   = new int[345];
ValueType* csr_val = new ValueType[345];

// Fill the CSR matrix
// ...

// rocALUTION local matrix object
LocalMatrix<ValueType> mat;

// Import CSR matrix to rocALUTION
mat.AllocateCSR("my_matrix", 345, 100, 100);
mat.CopyFromCSR(csr_row_ptr, csr_col, csr_val);
```

1.4.8 Raw Access to the Data

1.4.8.1 SetDataPtr

For vector and matrix objects, direct access to the raw data can be obtained via pointers. Already allocated data can be set with *SetDataPtr*. Setting data pointers will leave the original pointers empty.

void `rocalution::LocalVector::SetDataPtr`(ValueType **ptr, std::string name, int64_t size)

Initialize a *LocalVector* on the host with externally allocated data.

`SetDataPtr` has direct access to the raw data via pointers. Already allocated data can be set by passing the pointer.

Example

```
// Allocate vector
ValueType* ptr_vec = new ValueType[200];

// Fill vector
// ...

// rocALUTION local vector object
LocalVector<ValueType> vec;

// Set the vector data, ptr_vec will become invalid
vec.SetDataPtr(&ptr_vec, "my_vector", 200);
```

Note: Setting data pointer will leave the original pointer empty (set to NULL).

void rocalution::*LocalMatrix*::**SetDataPtrCOO**(int **row, int **col, ValueType **val, std::string name, int64_t nnz, int64_t nrow, int64_t ncol)

void rocalution::*LocalMatrix*::**SetDataPtrCSR**(PtrType **row_offset, int **col, ValueType **val, std::string name, int64_t nnz, int64_t nrow, int64_t ncol)

void rocalution::*LocalMatrix*::**SetDataPtrMCSR**(int **row_offset, int **col, ValueType **val, std::string name, int64_t nnz, int64_t nrow, int64_t ncol)

void rocalution::*LocalMatrix*::**SetDataPtrELL**(int **col, ValueType **val, std::string name, int64_t nnz, int64_t nrow, int64_t ncol, int max_row)

void rocalution::*LocalMatrix*::**SetDataPtrDIA**(int **offset, ValueType **val, std::string name, int64_t nnz, int64_t nrow, int64_t ncol, int num_diag)

void rocalution::*LocalMatrix*::**SetDataPtrDENSE**(ValueType **val, std::string name, int64_t nrow, int64_t ncol)

Initialize a *LocalMatrix* on the host with externally allocated data.

SetDataPtr functions have direct access to the raw data via pointers. Already allocated data can be set by passing their pointers.

Example

```
// Allocate a CSR matrix
int* csr_row_ptr   = new int[100 + 1];
int* csr_col_ind   = new int[345];
ValueType* csr_val = new ValueType[345];

// Fill the CSR matrix
// ...

// rocALUTION local matrix object
LocalMatrix<ValueType> mat;

// Set the CSR matrix data, csr_row_ptr, csr_col and csr_val pointers become
// invalid
mat.SetDataPtrCSR(&csr_row_ptr, &csr_col, &csr_val, "my_matrix", 345, 100, 100);
```

Note: Setting data pointers will leave the original pointers empty (set to NULL).

1.4.8.2 LeaveDataPtr

With *LeaveDataPtr*, the raw data from the object can be obtained. This will leave the object empty.

void rocalution::*LocalVector*::**LeaveDataPtr**(ValueType **ptr)

Leave a *LocalVector* to host pointers.

LeaveDataPtr has direct access to the raw data via pointers. A *LocalVector* object can leave its raw data to a host pointer. This will leave the *LocalVector* empty.

Example

```
// rocALUTION local vector object
LocalVector<ValueType> vec;

// Allocate the vector
vec.Allocate("my_vector", 100);

// Fill vector
// ...

ValueType* ptr_vec = NULL;

// Get (steal) the data from the vector, this will leave the local vector␣
↪object empty
vec.LeaveDataPtr(&ptr_vec);
```

void rocalution::*LocalMatrix*::**LeaveDataPtrCOO**(int **row, int **col, ValueType **val)

void rocalution::*LocalMatrix*::**LeaveDataPtrCSR**(PtrType **row_offset, int **col, ValueType **val)

void rocalution::*LocalMatrix*::**LeaveDataPtrMCSR**(int **row_offset, int **col, ValueType **val)

void rocalution::*LocalMatrix*::**LeaveDataPtrELL**(int **col, ValueType **val, int &max_row)

void rocalution::*LocalMatrix*::**LeaveDataPtrDIA**(int **offset, ValueType **val, int &num_diag)

void rocalution::*LocalMatrix*::**LeaveDataPtrDENSE**(ValueType **val)

Leave a *LocalMatrix* to host pointers.

LeaveDataPtr functions have direct access to the raw data via pointers. A *LocalMatrix* object can leave its raw data to host pointers. This will leave the *LocalMatrix* empty.

Example

```
// rocALUTION CSR matrix object
LocalMatrix<ValueType> mat;

// Allocate the CSR matrix
```

(continues on next page)

(continued from previous page)

```
mat.AllocateCSR("my_matrix", 345, 100, 100);

// Fill CSR matrix
// ...

int* csr_row_ptr   = NULL;
int* csr_col_ind   = NULL;
ValueType* csr_val = NULL;

// Get (steal) the data from the matrix, this will leave the local matrix
// object empty
mat.LeaveDataPtrCSR(&csr_row_ptr, &csr_col_ind, &csr_val);
```

Note: If the object is allocated on the host, then the pointers obtained from *SetDataPtr* and *LeaveDataPtr* will be on the host. If the vector object is on the accelerator, then the data pointers will be on the accelerator.

Note: If the object is moved to and from the accelerator, then the original pointer will be invalid.

Note: Never rely on old pointers, hidden object movement to and from the accelerator will make them invalid.

Note: Whenever you pass or obtain pointers to/from a rocALUTION object, you need to use the same memory allocation/free functions. Please check the source code for that (for host *src/utils/allocate_free.cpp* and for HIP *src/base/hip/hip_allocate_free.cpp*)

1.4.9 Copy CSR Matrix Host Data

void `rocalution::`*`LocalMatrix`*`::`**`CopyFromHostCSR`**(const PtrType *row_offset, const int *col, const ValueType *val, const std::string &name, int64_t nnz, int64_t nrow, int64_t ncol)

Allocates and copies (imports) a host CSR matrix.

If the CSR matrix data pointers are only accessible as constant, the user can create a *LocalMatrix* object and pass const CSR host pointers. The *LocalMatrix* will then be allocated and the data will be copied to the corresponding backend, where the original object was located at.

Parameters

- **`row_offset`** – **[in]** CSR matrix row offset pointers.
- **`col`** – **[in]** CSR matrix column indices.
- **`val`** – **[in]** CSR matrix values array.
- **`name`** – **[in]** Matrix object name.
- **`nnz`** – **[in]** Number of non-zero elements.
- **`nrow`** – **[in]** Number of rows.

- **ncol** – **[in]** Number of columns.

1.4.10 Copy Data

The user can copy data to and from a local vector by using *CopyFromData() CopyToData()*.

void rocalution::*LocalVector*::**CopyFromData**(const ValueType *data)

Copy (import) vector.

Copy (import) vector data that is described in one array (values). The object data has to be allocated with *Allocate()*, using the corresponding size of the data, first.

Parameters
data – **[in]** data to be imported.

void rocalution::*LocalVector*::**CopyToData**(ValueType *data) const

Copy (export) vector.

Copy (export) vector data that is described in one array (values). The output array has to be allocated, using the corresponding size of the data, first. Size can be obtain by *GetSize()*.

Parameters
data – **[out]** exported data.

1.4.11 Object Info

virtual void rocalution::*BaseRocalution*::**Info**(void) const = 0

Print object information.

Info can print object information about any rocALUTION object. This information consists of object properties and backend data.

Example

```
mat.Info();
vec.Info();
```

1.4.12 Copy

All matrix and vector objects provide a *CopyFrom()* function. The destination object should have the same size or be empty. In the latter case, the object is allocated at the source platform.

virtual void rocalution::*LocalVector*::**CopyFrom**(const LocalVector<ValueType> &src)

Copy vector from another vector.

CopyFrom copies values from another vector.

Example

```
LocalVector<ValueType> vec1, vec2;

// Allocate and initialize vec1 and vec2
// ...

// Move vec1 to accelerator
// vec1.MoveToAccelerator();

// Now, vec1 is on the accelerator (if available)
// and vec2 is on the host

// Copy vec1 to vec2 (or vice versa) will move data between host and
// accelerator backend
vec1.CopyFrom(vec2);
```

Note: This function allows cross platform copying. One of the objects could be allocated on the accelerator backend.

Parameters

src – **[in]** *Vector*, where values should be copied from.

void rocalution::*LocalMatrix*::**CopyFrom**(const LocalMatrix<ValueType> &src)

Copy matrix from another *LocalMatrix*.

CopyFrom copies values and structure from another local matrix. Source and destination matrix should be in the same format.

Example

```
LocalMatrix<ValueType> mat1, mat2;

// Allocate and initialize mat1 and mat2
// ...

// Move mat1 to accelerator
// mat1.MoveToAccelerator();

// Now, mat1 is on the accelerator (if available)
// and mat2 is on the host

// Copy mat1 to mat2 (or vice versa) will move data between host and
// accelerator backend
mat1.CopyFrom(mat2);
```

Note: This function allows cross platform copying. One of the objects could be allocated on the accelerator backend.

Parameters

src – **[in]** Local matrix where values and structure should be copied from.

Note: For vectors, the user can specify source and destination offsets and thus copy only a part of the whole vector into another vector.

Warning: doxygenfunction: Unable to resolve function "rocalution::LocalVector::CopyFrom" with arguments (const LocalVector<ValueType>&, int, int, int) in doxygen xml output for project "rocALUTION" from directory: ../docBin/xml. Potential matches:

```
- void CopyFrom(const LocalVector<ValueType> &src)
- void CopyFrom(const LocalVector<ValueType> &src, int64_t src_offset, int64_t dst_
↪offset, int64_t size)
```

1.4.13 Clone

The copy operators allow you to copy the values of the object to another object, without changing the backend specification of the object. In many algorithms, you might need auxiliary vectors or matrices. These objects can be cloned with CloneFrom().

1.4.13.1 CloneFrom

virtual void rocalution::*LocalVector*::**CloneFrom**(const LocalVector<ValueType> &src)

Clone the vector.

`CloneFrom` clones the entire vector, with data and backend descriptor from another *Vector*.

Example

```
LocalVector<ValueType> vec;

// Allocate and initialize vec (host or accelerator)
// ...

LocalVector<ValueType> tmp;

// By cloning vec, tmp will have identical values and will be on the same
// backend as vec
tmp.CloneFrom(vec);
```

Parameters

src – **[in]** *Vector* to clone from.

void rocalution::*LocalMatrix*::**CloneFrom**(const LocalMatrix<ValueType> &src)

Clone the matrix.

`CloneFrom` clones the entire matrix, including values, structure and backend descriptor from another *LocalMatrix*.

Example

```
LocalMatrix<ValueType> mat;

// Allocate and initialize mat (host or accelerator)
// ...

LocalMatrix<ValueType> tmp;

// By cloning mat, tmp will have identical values and structure and will be on
// the same backend as mat
tmp.CloneFrom(mat);
```

Parameters

src – **[in]** *LocalMatrix* to clone from.

1.4.13.2 CloneBackend

virtual void `rocalution::BaseRocalution::CloneBackend`(const BaseRocalution<ValueType> &src)

Clone the Backend descriptor from another object.

With `CloneBackend`, the backend can be cloned without copying any data. This is especially useful, if several objects should reside on the same backend, but keep their original data.

Example

```
LocalVector<ValueType> vec;
LocalMatrix<ValueType> mat;

// Allocate and initialize vec and mat
// ...

LocalVector<ValueType> tmp;
// By cloning backend, tmp and vec will have the same backend as mat
tmp.CloneBackend(mat);
vec.CloneBackend(mat);

// The following matrix vector multiplication will be performed on the backend
// selected in mat
mat.Apply(vec, &tmp);
```

Parameters

src – **[in]** Object, where the backend should be cloned from.

1.4.14 Check

virtual bool rocalution::*LocalVector*::**Check**(void) const

Perform a sanity check of the vector.

Checks, if the vector contains valid data, i.e. if the values are not infinity and not NaN (not a number).

Return values

- **true** – if the vector is ok (empty vector is also ok).
- **false** – if there is something wrong with the values.

bool rocalution::*LocalMatrix*::**Check**(void) const

Perform a sanity check of the matrix.

Checks, if the matrix contains valid data, i.e. if the values are not infinity and not NaN (not a number) and if the structure of the matrix is correct (e.g. indices cannot be negative, CSR and COO matrices have to be sorted, etc.).

Return values

- **true** – if the matrix is ok (empty matrix is also ok).
- **false** – if there is something wrong with the structure or values.

Checks, if the object contains valid data. For vectors, the function checks if the values are not infinity and not NaN (not a number). For matrices, this function checks the values and if the structure of the matrix is correct (e.g. indices cannot be negative, CSR and COO matrices have to be sorted, etc.).

1.4.15 Sort

void rocalution::*LocalMatrix*::**Sort**(void)

Sort the matrix indices.

Sorts the matrix by indices.

- For CSR matrices, column values are sorted.
- For COO matrices, row indices are sorted.

1.4.16 Keying

void rocalution::*LocalMatrix*::**Key**(long int &row_key, long int &col_key, long int &val_key) const

Compute a unique hash key for the matrix arrays.

Typically, it is hard to compare if two matrices have the same structure (and values). To do so, rocALUTION provides a keying function, that generates three keys, for the row index, column index and values array.

Parameters

- **row_key** – **[out]** row index array key
- **col_key** – **[out]** column index array key
- **val_key** – **[out]** values array key

1.4.17 Graph Analyzers

The following functions are available for analyzing the connectivity in graph of the underlying sparse matrix.

- (R)CMK Ordering
- Maximal Independent Set
- Multi-Coloring
- Zero Block Permutation
- Connectivity Ordering

All graph analyzing functions return a permutation vector (integer type), which is supposed to be used with the *rocalution::LocalMatrix::Permute()* and *rocalution::LocalMatrix::PermuteBackward()* functions in the matrix and vector classes.

1.4.17.1 Cuthill-McKee Ordering

void rocalution::*LocalMatrix*::**CMK**(*LocalVector*<int> *permutation) const

Create permutation vector for CMK reordering of the matrix.

The Cuthill-McKee ordering minimize the bandwidth of a given sparse matrix.

Example

```
LocalVector<int> cmk;

mat.CMK(&cmk);
mat.Permute(cmk);
```

Parameters

permutation – **[out]** permutation vector for CMK reordering

void rocalution::*LocalMatrix*::**RCMK**(*LocalVector*<int> *permutation) const

Create permutation vector for reverse CMK reordering of the matrix.

The Reverse Cuthill-McKee ordering minimize the bandwidth of a given sparse matrix.

Example

```
LocalVector<int> rcmk;

mat.RCMK(&rcmk);
mat.Permute(rcmk);
```

Parameters

permutation – **[out]** permutation vector for reverse CMK reordering

1.4.17.2 Maximal Independent Set

void rocalution::*LocalMatrix*::**MaximalIndependentSet**(int &size, *LocalVector*<int> *permutation) const

Perform maximal independent set decomposition of the matrix.

The Maximal Independent Set algorithm finds a set with maximal size, that contains elements that do not depend on other elements in this set.

Example

```
LocalVector<int> mis;
int size;

mat.MaximalIndependentSet(size, &mis);
mat.Permute(mis);
```

Parameters

- **size** – **[out]** number of independent sets
- **permutation** – **[out]** permutation vector for maximal independent set reordering

1.4.17.3 Multi-Coloring

void rocalution::*LocalMatrix*::**MultiColoring**(int &num_colors, int **size_colors, *LocalVector*<int> *permutation) const

Perform multi-coloring decomposition of the matrix.

The Multi-Coloring algorithm builds a permutation (coloring of the matrix) in a way such that no two adjacent nodes in the sparse matrix have the same color.

Example

```
LocalVector<int> mc;
int num_colors;
int* block_colors = NULL;

mat.MultiColoring(num_colors, &block_colors, &mc);
mat.Permute(mc);
```

Parameters

- **num_colors** – **[out]** number of colors
- **size_colors** – **[out]** pointer to array that holds the number of nodes for each color
- **permutation** – **[out]** permutation vector for multi-coloring reordering

1.4.17.4 Zero Block Permutation

void rocalution::*LocalMatrix*::**ZeroBlockPermutation**(int &size, *LocalVector*<int> *permutation) const

Return a permutation for saddle-point problems (zero diagonal entries)

For Saddle-Point problems, (i.e. matrices with zero diagonal entries), the Zero Block Permutation maps all zero-diagonal elements to the last block of the matrix.

Example

```
LocalVector<int> zbp;
int size;

mat.ZeroBlockPermutation(size, &zbp);
mat.Permute(zbp);
```

Parameters

- **size** – **[out]**
- **permutation** – **[out]** permutation vector for zero block permutation

1.4.17.5 Connectivity Ordering

void rocalution::*LocalMatrix*::**ConnectivityOrder**(*LocalVector*<int> *permutation) const

Create permutation vector for connectivity reordering of the matrix.

Connectivity ordering returns a permutation, that sorts the matrix by non-zero entries per row.

Example

```
LocalVector<int> conn;

mat.ConnectivityOrder(&conn);
mat.Permute(conn);
```

Parameters

permutation – **[out]** permutation vector for connectivity reordering

1.4.18 Basic Linear Algebra Operations

For a full list of functions and routines involving operators and vectors, see the API specifications.

1.5 Multi-node Computation

1.5.1 Introduction

This chapter describes all base objects (matrices and vectors) for computation on multi-node (distributed memory) systems.

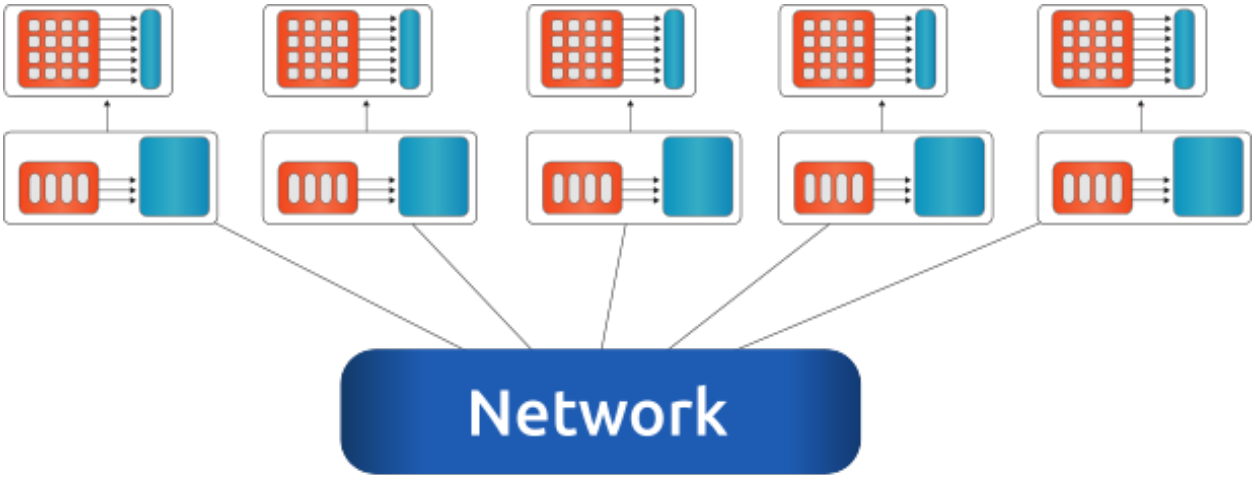

Fig. 1.3: An example for a multi-node configuration, where all nodes are connected via network. Single socket systems with a single accelerator.

To each compute node, one or more accelerators can be attached. The compute node could be any kind of shared-memory (single, dual, quad CPU) system, details on a single-node can be found in Fig. 1.2.

Note: The memory of accelerator and host are physically different. All nodes can communicate with each other via network.

For the communication channel between different nodes (and between the accelerators on single or multiple nodes) the MPI library is used.

rocALUTION supports non-overlapping type of distribution, where the computational domain is split into several sub-domain with the corresponding information about the boundary and ghost layers. An example is shown in Fig. 1.5. The square box domain is distributed into four sub-domains. Each subdomain belongs to a process *P0*, *P1*, *P2* and *P3*.

To perform a sparse matrix-vector multiplication (SpMV), each process need to multiply its own portion of the domain and update the corresponding ghost elements. For *P0*, this multiplication reads

$$Ax = y,$$
$$A_I x_I + A_G x_G = y_I,$$

where I stands for interior and G stands for ghost. x_G is a vector with three sections, coming from *P1*, *P2* and *P3*. The whole ghost part of the global vector is used mainly for the SpMV product. It does not play any role in the computation of vector-vector operations.

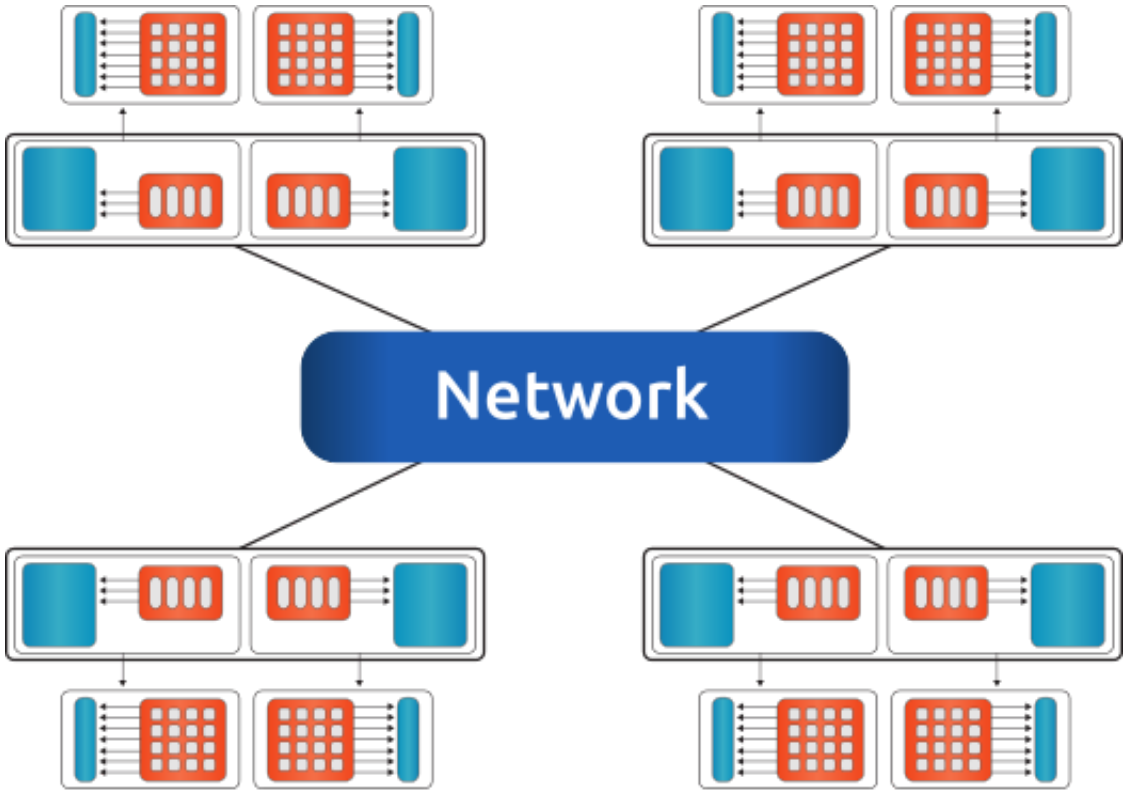

Fig. 1.4: An example for a multi-node configuration, where all nodes are connected via network. Dual socket systems with two accelerators attached to each node.

Fig. 1.5: An example for domain distribution.

1.5.2 Code Structure

Each object contains two local sub-objects. The global matrix stores interior and ghost matrix by local objects. Similarily, the global vector stores its data by two local objects. In addition to the local data, the global objects have information about the global communication through the parallel manager.

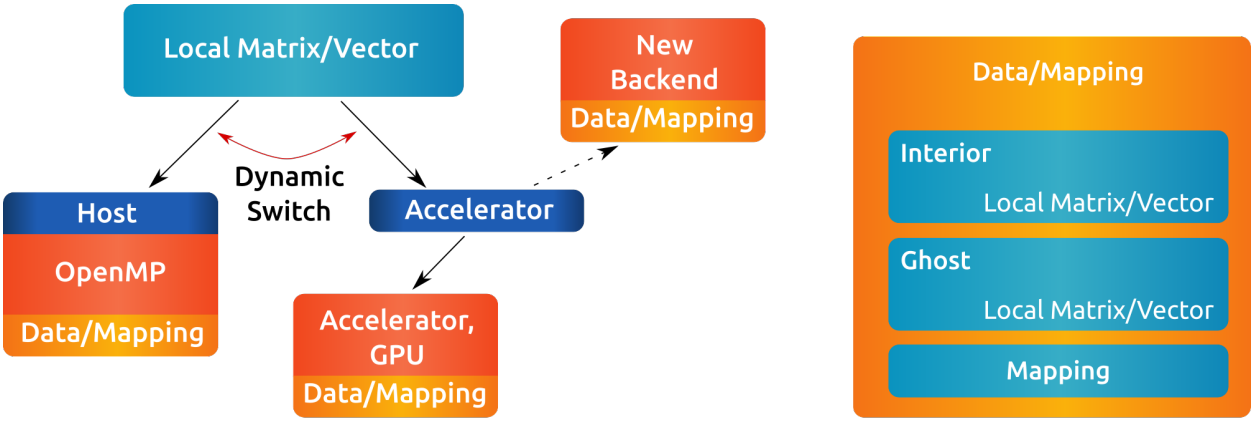

Fig. 1.6: Global matrices and vectors.

1.5.3 Parallel Manager

class **ParallelManager** : public rocalution::RocalutionObj

Parallel Manager class.

The parallel manager class handles the communication and the mapping of the global operators. Each global operator and vector need to be initialized with a valid parallel manager in order to perform any operation. For many distributed simulations, the underlying operator is already distributed. This information need to be passed to the parallel manager.

The parallel manager class hosts the following functions:

void rocalution::*ParallelManager*::**SetMPICommunicator**(const void *comm)

Set the MPI communicator.

void rocalution::*ParallelManager*::**Clear**(void)

Clear all allocated resources.

Warning: doxygenfunction: Cannot find function “rocalution::ParallelManager::GetGlobalSize” in doxygen xml output for project “rocALUTION” from directory: ../docBin/xml

Warning: doxygenfunction: Cannot find function “rocalution::ParallelManager::GetLocalSize” in doxygen xml output for project “rocALUTION” from directory: ../docBin/xml

int rocalution::*ParallelManager*::**GetNumReceivers**(void) const

Return the number of receivers.

int rocalution::*ParallelManager*::**GetNumSenders**(void) const

Return the number of senders.

int rocalution::*ParallelManager*::**GetNumProcs**(void) const

Return the number of involved processes.

Warning: doxygenfunction: Cannot find function "rocalution::ParallelManager::SetGlobalSize" in doxygen xml output for project "rocALUTION" from directory: ../docBin/xml

Warning: doxygenfunction: Cannot find function "rocalution::ParallelManager::SetLocalSize" in doxygen xml output for project "rocALUTION" from directory: ../docBin/xml

void rocalution::*ParallelManager*::**SetBoundaryIndex**(int size, const int *index)

Set all boundary indices of this ranks process.

void rocalution::*ParallelManager*::**SetReceivers**(int nrecv, const int *recvs, const int *recv_offset)

Number of processes, the current process is receiving data from, array of the processes, the current process is receiving data from and offsets, where the boundary for process 'receiver' starts.

void rocalution::*ParallelManager*::**SetSenders**(int nsend, const int *sends, const int *send_offset)

Number of processes, the current process is sending data to, array of the processes, the current process is sending data to and offsets where the ghost part for process 'sender' starts.

void rocalution::*ParallelManager*::**ReadFileASCII**(const std::string &filename)

Read file that contains all relevant parallel manager data.

void rocalution::*ParallelManager*::**WriteFileASCII**(const std::string &filename) const

Write file that contains all relevant parallel manager data.

To setup a parallel manager, the required information is:

- Global size
- Local size of the interior/ghost for each process
- Communication pattern (what information need to be sent to whom)

1.5.4 Global Matrices and Vectors

const *LocalMatrix*<ValueType> &rocalution::*GlobalMatrix*::**GetInterior**() const

const *LocalMatrix*<ValueType> &rocalution::*GlobalMatrix*::**GetGhost**() const

Warning: doxygenfunction: Unable to resolve function "rocalution::GlobalVector::GetInterior" with arguments None in doxygen xml output for project "rocALUTION" from directory: ../docBin/xml. Potential matches:

```
- LocalVector<ValueType> &GetInterior()
- const LocalVector<ValueType> &GetInterior() const
```

The global matrices and vectors store their data via two local objects. For the global matrix, the interior can be access via the `rocalution::GlobalMatrix::GetInterior()` and `rocalution::GlobalMatrix::GetGhost()`

functions, which point to two valid local matrices. Similarily, the global vector can be accessed by `rocalution::GlobalVector::GetInterior()`.

1.5.4.1 Asynchronous SpMV

To minimize latency and to increase scalability, rocALUTION supports asynchronous sparse matrix-vector multiplication. The implementation of the SpMV starts with asynchronous transfer of the required ghost buffers, while at the same time it computes the interior matrix-vector product. When the computation of the interior SpMV is done, the ghost transfer is synchronized and the ghost SpMV is performed. To minimize the PCI-E bus, the HIP implementation provides a special packaging technique for transferring all ghost data into a contiguous memory buffer.

1.5.5 File I/O

The user can store and load all global structures from and to files. For a solver, the necessary data would be

- the parallel manager
- the sparse matrix
- and the vector

Reading/writing from/to files can be done fully in parallel without any communication. Fig. 1.7 visualizes data of a 4×4 grid example which is distributed among 4 MPI processes (organized in 2×2). Each local matrix stores the local unknowns (with local indexing). Fig. 1.8 furthermore illustrates the data associated with *RANK0*.

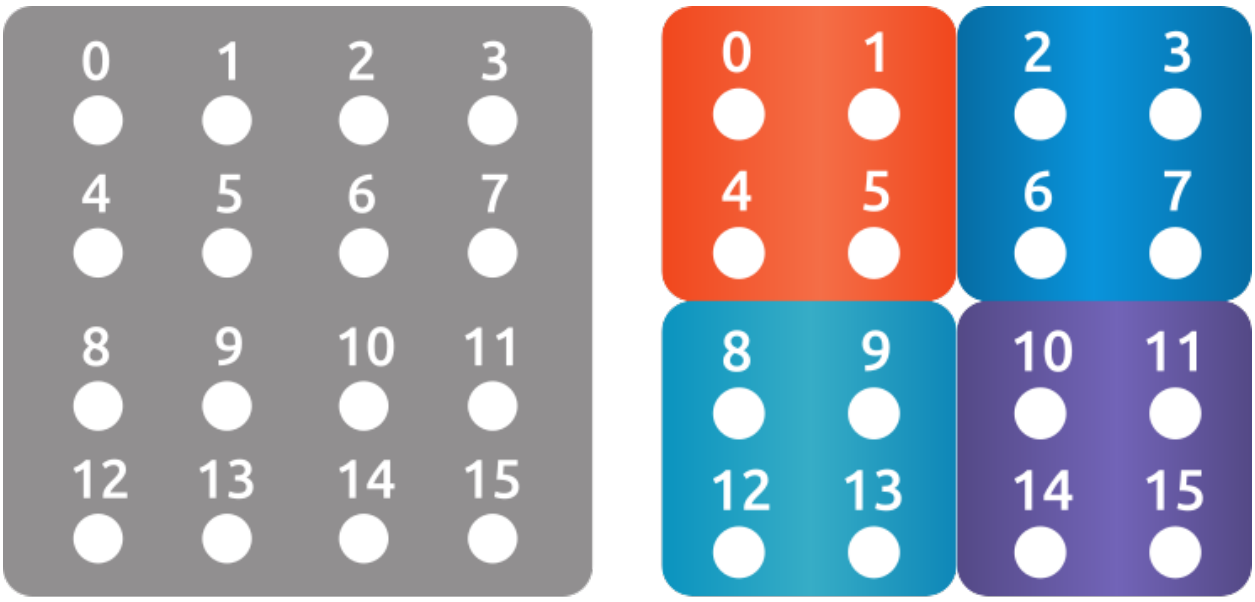

Fig. 1.7: An example of 4×4 grid, distributed in 4 domains (2×2).

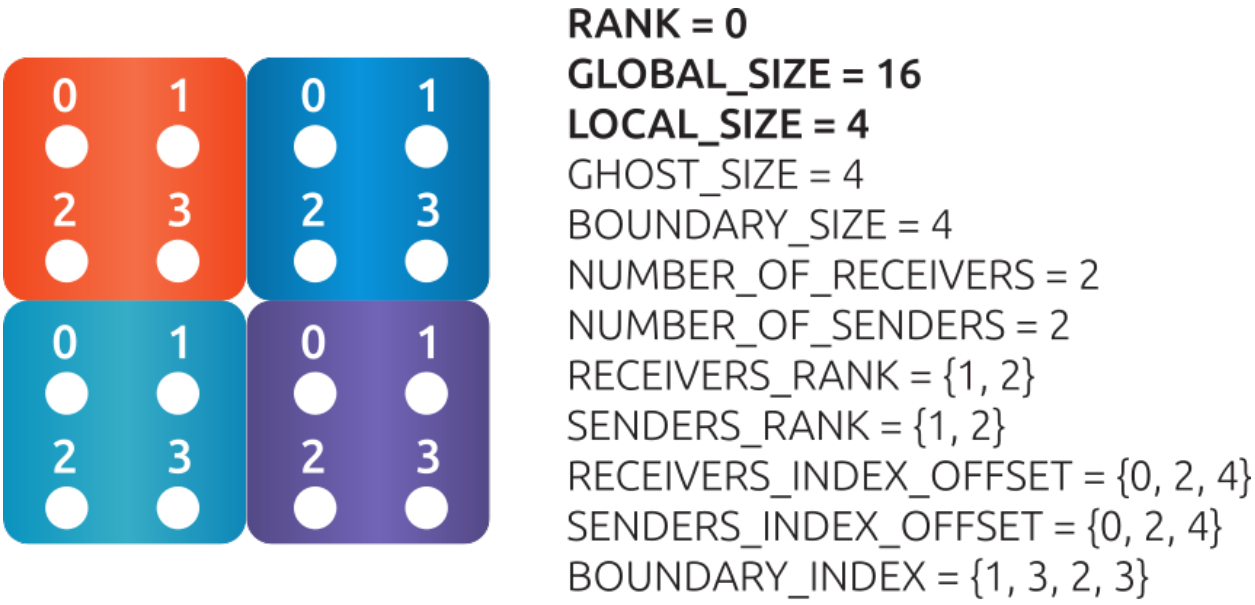

Fig. 1.8: An example of 4 MPI processes and the data associated with *RANK0*.

1.5.5.1 File Organization

When the parallel manager, global matrix or global vector are writing to a file, the main file (passed as a file name to this function) will contain information for all files on all ranks.

```
parallelmanager.dat.rank.0
parallelmanager.dat.rank.1
parallelmanager.dat.rank.2
parallelmanager.dat.rank.3
```

```
matrix.mtx.interior.rank.0
matrix.mtx.ghost.rank.0
matrix.mtx.interior.rank.1
matrix.mtx.ghost.rank.1
matrix.mtx.interior.rank.2
matrix.mtx.ghost.rank.2
matrix.mtx.interior.rank.3
matrix.mtx.ghost.rank.3
```

```
rhs.dat.rank.0
rhs.dat.rank.1
rhs.dat.rank.2
rhs.dat.rank.3
```

1.5.5.2 Parallel Manager

The data for each rank can be split into receiving and sending information. For receiving data from neighboring processes, see Fig. 1.9, *RANK0* need to know what type of data will be received and from whom. For sending data to neighboring processes, see Fig. 1.10, *RANK0* need to know where and what to send.

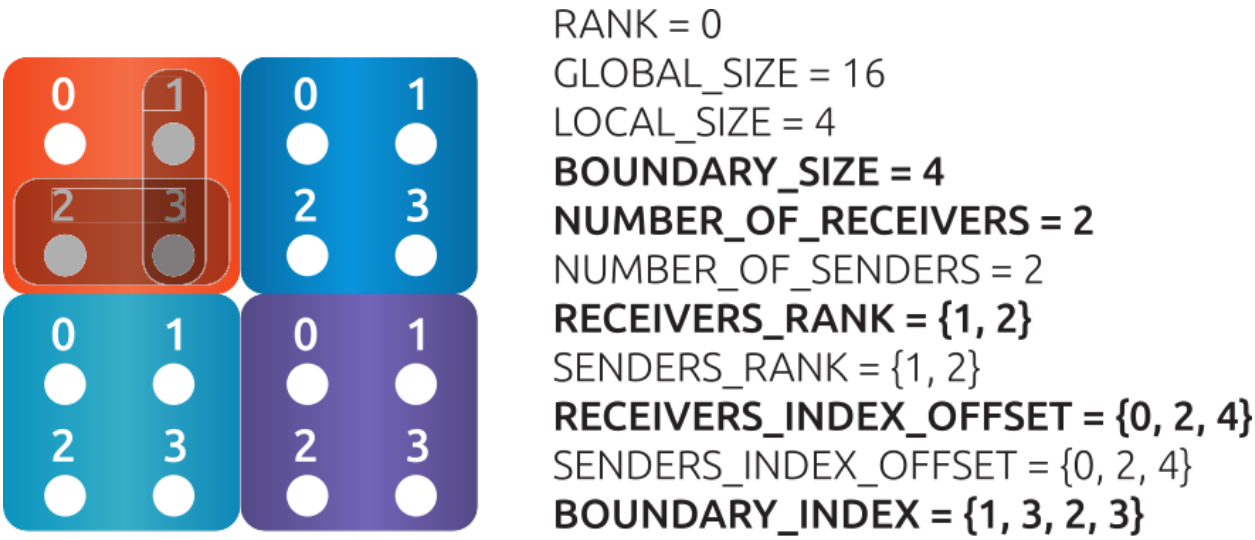

Fig. 1.9: An example of 4 MPI processes, *RANK0* receives data (the associated data is marked bold).

To receive data, *RANK0* requires:

- Number of MPI ranks, which will send data to *RANK0* (NUMBER_OF_RECEIVERS - integer value).
- Which are the MPI ranks, sending the data (RECEIVERS_RANK - integer array).
- How will the received data (from each rank) be stored in the ghost vector (RECEIVERS_INDEX_OFFSET - integer array). In this example, the first 30 elements will be received from *P1* $[0, 2)$ and the second 30 from *P2* $[2, 4)$.

To send data, *RANK0* requires:

- Total size of the sending information (BOUNDARY_SIZE - integer value).
- Number of MPI ranks, which will receive data from *RANK0* (NUMBER_OF_SENDERS - integer value).
- Which are the MPI ranks, receiving the data (SENDERS_RANK - integer array).
- How will the sending data (from each rank) be stored in the sending buffer (SENDERS_INDEX_OFFSET - integer array). In this example, the first 30 elements will be sent to *P1* $[0, 2)$ and the second 30 to *P2* $[2, 4)$.
- The elements, which need to be send (BOUNDARY_INDEX - integer array). In this example, the data which need to be send to *P1* and *P2* is the ghost layer, marked as ghost *P0*. The vertical stripe need to be send to *P1* and the horizontal stripe to *P2*. The numbering of local unknowns (in local indexing) for *P1* (the vertical stripes) are 1, 2 (size of 2) and stored in the BOUNDARY_INDEX. After 2 elements, the elements for *P2* are stored, they are 2, 3 (2 elements).

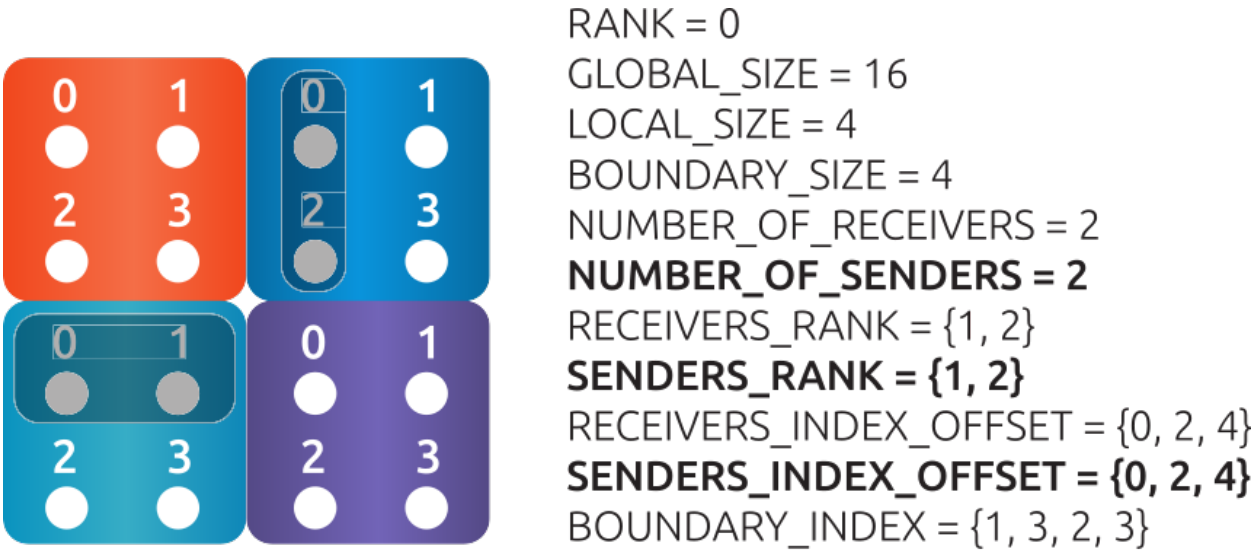

Fig. 1.10: An example of 4 MPI processes, *RANK0* sends data (the associated data is marked bold).

1.5.5.3 Matrices

Each rank hosts two local matrices, interior and ghost matrix. They can be stored in separate files, one for each matrix. The file format could be Matrix Market (MTX) or binary.

1.5.5.4 Vectors

Each rank holds the local interior vector only. It is stored in a single file. The file could be ASCII or binary.

1.6 Solvers

1.6.1 Code Structure

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **Solver** : public rocalution::RocalutionObj

Base class for all solvers and preconditioners.

Most of the solvers can be performed on linear operators *LocalMatrix*, *LocalStencil* and *GlobalMatrix* - i.e. the solvers can be performed locally (on a shared memory system) or in a distributed manner (on a cluster) via MPI. The only exception is the AMG (Algebraic Multigrid) solver which has two versions (one for *LocalMatrix* and one for *GlobalMatrix* class). The only pure local solvers (which do not support global/MPI operations) are the mixed-precision defect-correction solver and all direct solvers.

All solvers need three template parameters - Operators, Vectors and Scalar type.

The *Solver* class is purely virtual and provides an interface for

- *SetOperator()* to set the operator A, i.e. the user can pass the matrix here.
- *Build()* to build the solver (including preconditioners, sub-solvers, etc.). The user need to specify the operator first before calling *Build()*.

- *Solve()* to solve the system $Ax = b$. The user need to pass a right-hand-side b and a vector x, where the solution will be obtained.
- *Print()* to show solver information.
- *ReBuildNumeric()* to only re-build the solver numerically (if possible).
- *MoveToHost()* and *MoveToAccelerator()* to offload the solver (including preconditioners and sub-solvers) to the host/accelerator.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::DirectLinearSolver< OperatorType, VectorType, ValueType >*, *rocalution::IterativeLinearSolver< OperatorType, VectorType, ValueType >*, *rocalution::Preconditioner< OperatorType, VectorType, ValueType >*

It provides an interface for

void `rocalution::Solver::SetOperator`(const OperatorType &op)

Set the *Operator* of the solver.

virtual void `rocalution::Solver::Build`(void)

Build the solver (data allocation, structure and numerical computation)

virtual void `rocalution::Solver::Clear`(void)

Clear (free all local data) the solver.

virtual void `rocalution::Solver::Solve`(const VectorType &rhs, VectorType *x) = 0

Solve *Operator* x = rhs.

virtual void `rocalution::Solver::Print`(void) const = 0

Print information about the solver.

virtual void `rocalution::Solver::ReBuildNumeric`(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

virtual void `rocalution::Solver::MoveToHost`(void)

Move all data (i.e. move the solver) to the host.

virtual void `rocalution::Solver::MoveToAccelerator`(void)

Move all data (i.e. move the solver) to the accelerator.

1.6.2 Iterative Linear Solvers

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **IterativeLinearSolver** : public rocalution::*Solver*<*OperatorType*, *VectorType*, *ValueType*>

Base class for all linear iterative solvers.

The iterative solvers are controlled by an iteration control object, which monitors the convergence properties of the solver, i.e. maximum number of iteration, relative tolerance, absolute tolerance and divergence tolerance. The iteration control can also record the residual history and store it in an ASCII file.

- *Init()*, *InitMinIter()*, *InitMaxIter()* and *InitTol()* initialize the solver and set the stopping criteria.

- *RecordResidualHistory()* and *RecordHistory()* start the recording of the residual and write it into a file.
- *Verbose()* sets the level of verbose output of the solver (0 - no output, 2 - detailed output, including residual and iteration information).
- *SetPreconditioner()* sets the preconditioning.

All iterative solvers are controlled based on

- Absolute stopping criteria, when $|r_k|_{L_p} < \epsilon_{abs}$
- Relative stopping criteria, when $|r_k|_{L_p}/|r_1|_{L_p} \leq \epsilon_{rel}$
- Divergence stopping criteria, when $|r_k|_{L_p}/|r_1|_{L_p} \geq \epsilon_{div}$
- Maximum number of iteration N, when $k = N$

where k is the current iteration, r_k the residual for the current iteration k (i.e. $r_k = b - Ax_k$) and r_1 the starting residual (i.e. $r_1 = b - Ax_{init}$). In addition, the minimum number of iterations M can be specified. In this case, the solver will not stop to iterate, before $k \geq M$.

The L_p norm is used for the computation, where p could be 1, 2 and ∞. The norm computation can be set with *SetResidualNorm()* with 1 for L_1, 2 for L_2 and 3 for L_∞. For the computation with L_∞, the index of the maximum value can be obtained with *GetAmaxResidualIndex()*. If this function is called and L_∞ was not selected, this function will return -1.

The reached criteria can be obtained with *GetSolverStatus()*, returning

- 0, if no criteria has been reached yet
- 1, if absolute tolerance has been reached
- 2, if relative tolerance has been reached
- 3, if divergence tolerance has been reached
- 4, if maximum number of iteration has been reached

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::BaseMultiGrid< OperatorType, VectorType, ValueType >*, *rocalution::BiCGStab< OperatorType, VectorType, ValueType >*, *rocalution::BiCGStabl< OperatorType, VectorType, ValueType >*, *rocalution::CG< OperatorType, VectorType, ValueType >*, *rocalution::Chebyshev< OperatorType, VectorType, ValueType >*, *rocalution::CR< OperatorType, VectorType, ValueType >*, *rocalution::FCG< OperatorType, VectorType, ValueType >*, *rocalution::FGMRES< OperatorType, VectorType, ValueType >*, *rocalution::FixedPoint< OperatorType, VectorType, ValueType >*, *rocalution::GMRES< OperatorType, VectorType, ValueType >*, *rocalution::IDR< OperatorType, VectorType, ValueType >*, *rocalution::QMRCGStab< OperatorType, VectorType, ValueType >*

It provides an interface for

void rocalution::*IterativeLinearSolver*::**Init**(double abs_tol, double rel_tol, double div_tol, int max_iter)

Initialize the solver with absolute/relative/divergence tolerance and maximum number of iterations.

void rocalution::*IterativeLinearSolver*::**Init**(double abs_tol, double rel_tol, double div_tol, int min_iter, int max_iter)

Initialize the solver with absolute/relative/divergence tolerance and minimum/maximum number of iterations.

void rocalution::*IterativeLinearSolver*::**InitMinIter**(int min_iter)

Set the minimum number of iterations.

void rocalution::*IterativeLinearSolver*::**InitMaxIter**(int max_iter)

Set the maximum number of iterations.

void rocalution::*IterativeLinearSolver*::**InitTol**(double abs, double rel, double div)

Set the absolute/relative/divergence tolerance.

void rocalution::*IterativeLinearSolver*::**RecordResidualHistory**(void)

Record the residual history.

void rocalution::*IterativeLinearSolver*::**RecordHistory**(const std::string &filename) const

Write the history to file.

virtual void rocalution::*IterativeLinearSolver*::**Verbose**(int verb = 1)

Set the solver verbosity output.

virtual void rocalution::*IterativeLinearSolver*::**SetPreconditioner**(*Solver*<OperatorType, VectorType, ValueType> &precond)

Set a preconditioner of the linear solver.

void rocalution::*IterativeLinearSolver*::**SetResidualNorm**(int resnorm)

Set the residual norm to L_1, L_2 or L_∞ norm.

- resnorm = 1 -> L_1 norm
- resnorm = 2 -> L_2 norm
- resnorm = 3 -> L_∞ norm

virtual int64_t rocalution::*IterativeLinearSolver*::**GetAmaxResidualIndex**(void)

Return absolute maximum index of residual vector when using L_∞ norm.

virtual int rocalution::*IterativeLinearSolver*::**GetSolverStatus**(void)

Return the current status.

1.6.3 Building and Solving Phase

Each iterative solver consists of a building step and a solving step. During the building step all necessary auxiliary data is allocated and the preconditioner is constructed. After that, the user can call the solving procedure, the solving step can be called several times.

When the initial matrix associated with the solver is on the accelerator, the solver will try to build everything on the accelerator. However, some preconditioners and solvers (such as FSAI and AMG) need to be constructed on the host before they can be transferred to the accelerator. If the initial matrix is on the host and we want to run the solver on the accelerator then we need to move the solver to the accelerator as well as the matrix, the right-hand-side and the solution vector.

Note: If you have a preconditioner associate with the solver, it will be moved automatically to the accelerator when you move the solver.

```
// CG solver
CG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType> ls;
// Multi-Colored ILU preconditioner
MultiColoredILU<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType> p;

// Move matrix and vectors to the accelerator
mat.MoveToAccelerator();
rhs.MoveToAccelerator();
x.MoveToAccelerator();

// Set mat to be the operator
ls.SetOperator(mat);
// Set p as the preconditioner of ls
ls.SetPreconditioner(p);

// Build the solver and preconditioner on the accelerator
ls.Build();

// Compute the solution on the accelerator
ls.Solve(rhs, &x);
```

```
// CG solver
CG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType> ls;
// Multi-Colored ILU preconditioner
MultiColoredILU<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType> p;

// Set mat to be the operator
ls.SetOperator(mat);
// Set p as the preconditioner of ls
ls.SetPreconditioner(p);

// Build the solver and preconditioner on the host
ls.Build();

// Move matrix and vectors to the accelerator
mat.MoveToAccelerator();
rhs.MoveToAccelerator();
x.MoveToAccelerator();

// Move linear solver to the accelerator
ls.MoveToAccelerator();

// Compute the solution on the accelerator
ls.Solve(rhs, &x);
```

1.6.4 Clear Function and Destructor

The *rocalution::Solver::Clear()* function clears all the data which is in the solver, including the associated preconditioner. Thus, the solver is not anymore associated with this preconditioner.

Note: The preconditioner is not deleted (via destructor), only a `rocalution::Preconditioner::Clear()` is called.

Note: When the destructor of the solver class is called, it automatically calls the *Clear()* function. Be careful, when declaring your solver and preconditioner in different places - we highly recommend to manually call the *Clear()* function of the solver and not to rely on the destructor of the solver.

1.6.5 Numerical Update

Some preconditioners require two phases in the their construction: an algebraic (e.g. compute a pattern or structure) and a numerical (compute the actual values) phase. In cases, where the structure of the input matrix is a constant (e.g. Newton-like methods) it is not necessary to fully re-construct the preconditioner. In this case, the user can apply a numerical update to the current preconditioner and pass the new operator with *rocalution::Solver::ReBuildNumeric()*. If the preconditioner/solver does not support the numerical update, then a full *rocalution::Solver::Clear()* and *rocalution::Solver::Build()* will be performed.

1.6.6 Fixed-Point Iteration

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **FixedPoint** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Fixed-Point Iteration Scheme.

The Fixed-Point iteration scheme is based on additive splitting of the matrix $A = M + N$. The scheme reads

$$x_{k+1} = M^{-1}(b - Nx_k).$$

It can also be reformulated as a weighted defect correction scheme

$$x_{k+1} = x_k - \omega M^{-1}(Ax_k - b).$$

The inversion of M can be performed by preconditioners (*Jacobi*, Gauss-Seidel, *ILU*, etc.) or by any type of solvers.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

void `rocalution::`*FixedPoint*`::`**SetRelaxation**(ValueType omega)

Set relaxation parameter ω.

1.6.7 Krylov Subspace Solvers

1.6.7.1 CG

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **CG** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Conjugate Gradient Method.

The Conjugate Gradient method is the best known iterative method for solving sparse symmetric positive definite (SPD) linear systems $Ax = b$. It is based on orthogonal projection onto the Krylov subspace $\mathcal{K}_m(r_0, A)$, where r_0 is the initial residual. The method can be preconditioned, where the approximation should also be SPD. SAAD

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

1.6.7.2 CR

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **CR** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Conjugate Residual Method.

The Conjugate Residual method is an iterative method for solving sparse symmetric semi-positive definite linear systems $Ax = b$. It is a Krylov subspace method and differs from the much more popular Conjugate Gradient method that the system matrix is not required to be positive definite. The method can be preconditioned where the approximation should also be SPD or semi-positive definite. SAAD

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

1.6.7.3 GMRES

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **GMRES** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Generalized Minimum Residual Method.

The Generalized Minimum Residual method (*GMRES*) is a projection method for solving sparse (non) symmetric linear systems $Ax = b$, based on restarting technique. The solution is approximated in a Krylov subspace $\mathcal{K} = \mathcal{K}_m$ and $\mathcal{L} = A\mathcal{K}_m$ with minimal residual, where $\mathcal{K}_m$ is the m-th Krylov subspace with $v_1 = r_0/||r_0||_2$. SAAD

The Krylov subspace basis size can be set using *SetBasisSize()*. The default size is 30.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*

- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

virtual void rocalution::*GMRES*::**SetBasisSize**(int size_basis)

Set the size of the Krylov subspace basis.

1.6.7.4 FGMRES

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **FGMRES** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Flexible Generalized Minimum Residual Method.

The Flexible Generalized Minimum Residual method (*FGMRES*) is a projection method for solving sparse (non) symmetric linear systems $Ax = b$. It is similar to the *GMRES* method with the only difference, the *FGMRES* is based on a window shifting of the Krylov subspace and thus allows the preconditioner M^{-1} to be not a constant operator. This can be especially helpful if the operation $M^{-1}x$ is the result of another iterative process and not a constant operator. SAAD

The Krylov subspace basis size can be set using *SetBasisSize()*. The default size is 30.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

virtual void rocalution::*FGMRES*::**SetBasisSize**(int size_basis)

Set the size of the Krylov subspace basis.

1.6.7.5 BiCGStab

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **BiCGStab** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Bi-Conjugate Gradient Stabilized Method.

The Bi-Conjugate Gradient Stabilized method is a variation of CGS and solves sparse (non) symmetric linear systems $Ax = b$. SAAD

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

1.6.7.6 IDR

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **IDR** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Induced Dimension Reduction Method.

The Induced Dimension Reduction method is a Krylov subspace method for solving sparse (non) symmetric linear systems $Ax = b$. IDR(s) generates residuals in a sequence of nested subspaces. IDR1 IDR2

The dimension of the shadow space can be set by *SetShadowSpace()*. The default size of the shadow space is 4.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

void `rocalution::`*IDR*`::SetShadowSpace`(int s)

Set the size of the Shadow Space.

1.6.7.7 FCG

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **FCG** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Flexible Conjugate Gradient Method.

The Flexible Conjugate Gradient method is an iterative method for solving sparse symmetric positive definite linear systems $Ax = b$. It is similar to the Conjugate Gradient method with the only difference, that it allows the preconditioner M^{-1} to be not a constant operator. This can be especially helpful if the operation $M^{-1}x$ is the result of another iterative process and not a constant operator. fcg

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

1.6.7.8 QMRCGStab

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **QMRCGStab** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Quasi-Minimal Residual Conjugate Gradient Stabilized Method.

The Quasi-Minimal Residual Conjugate Gradient Stabilized method is a variant of the Krylov subspace *BiCGStab* method for solving sparse (non) symmetric linear systems $Ax = b$. qmrcgstab

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

1.6.7.9 BiCGStab(l)

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **BiCGStabl** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Bi-Conjugate Gradient Stabilized (l) Method.

The Bi-Conjugate Gradient Stabilized (l) method is a generalization of *BiCGStab* for solving sparse (non) symmetric linear systems $Ax = b$. It minimizes residuals over l-dimensional Krylov subspaces. The degree l can be set with *SetOrder()*. bicgstabl

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

virtual void `rocalution::BiCGStabl::SetOrder`(int l)

Set the order.

1.6.8 Chebyshev Iteration Scheme

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **Chebyshev** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Chebyshev Iteration Scheme.

The *Chebyshev* Iteration scheme (also known as acceleration scheme) is similar to the *CG* method but requires minimum and maximum eigenvalues of the operator. templates

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

1.6.9 Mixed-Precision Defect Correction Scheme

template<class **OperatorTypeH**, class **VectorTypeH**, typename **ValueTypeH**, class **OperatorTypeL**, class **VectorTypeL**, typename **ValueTypeL**>
class **MixedPrecisionDC** : public rocalution::*IterativeLinearSolver*<*OperatorTypeH*, *VectorTypeH*, *ValueTypeH*>

Mixed-Precision Defect Correction Scheme.

The Mixed-Precision solver is based on a defect-correction scheme. The current implementation of the library is using host based correction in double precision and accelerator computation in single precision. The solver is implemeting the scheme

$$x_{k+1} = x_k + A^{-1} r_k,$$

where the computation of the residual $r_k = b - Ax_k$ and the update $x_{k+1} = x_k + d_k$ are performed on the host in double precision. The computation of the residual system $Ad_k = r_k$ is performed on the accelerator in single precision. In addition to the setup functions of the iterative solver, the user need to specify the inner ($Ad_k = r_k$) solver.

Template Parameters

- **OperatorTypeH** – - can be *LocalMatrix*
- **VectorTypeH** – - can be *LocalVector*
- **ValueTypeH** – - can be double
- **OperatorTypeL** – - can be *LocalMatrix*
- **VectorTypeL** – - can be *LocalVector*
- **ValueTypeL** – - can be float

1.6.10 MultiGrid Solvers

The library provides algebraic multigrid as well as a skeleton for geometric multigrid methods. The BaseMultigrid class itself is not constructing the data for the method. It contains the solution procedure for V, W and K-cycles. The AMG has two different versions for Local (non-MPI) and for Global (MPI) type of computations.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **BaseMultiGrid** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Base class for all multigrid solvers Trottenberg2003.

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::BaseAMG< OperatorType, VectorType, ValueType >*, *rocalution::MultiGrid< OperatorType, VectorType, ValueType >*

1.6.10.1 Geometric MultiGrid

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **MultiGrid** : public rocalution::*BaseMultiGrid*<*OperatorType*, *VectorType*, *ValueType*>

MultiGrid Method.

The *MultiGrid* method can be used with external data, such as externally computed restriction, prolongation and operator hierarchy. The user need to pass all this information for each level and for its construction. This includes smoothing step, prolongation/restriction, grid traversing and coarse grid solver. This data need to be passed to the solver. Trottenberg2003

- Restriction and prolongation operations can be performed in two ways, based on Restriction() and Prolongation() of the *LocalVector* class, or by matrix-vector multiplication. This is configured by a set function.
- Smoothers can be of any iterative linear solver. Valid options are *Jacobi*, Gauss-Seidel, *ILU*, etc. using a *FixedPoint* iteration scheme with pre-defined number of iterations. The smoothers could also be a solver such as *CG*, *BiCGStab*, etc.
- Coarse grid solver could be of any iterative linear solver type. The class also provides mechanisms to specify, where the coarse grid solver has to be performed, on the host or on the accelerator. The coarse grid solver can be preconditioned.
- Grid scaling based on a L_2 norm ratio.

- *Operator* matrices need to be passed on each grid level.

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

1.6.10.2 Algebraic MultiGrid

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **BaseAMG** : public rocalution::*BaseMultiGrid*<*OperatorType*, *VectorType*, *ValueType*>

Base class for all algebraic multigrid solvers.

The Algebraic *MultiGrid* solver is based on the *BaseMultiGrid* class. The coarsening is obtained by different aggregation techniques. The smoothers can be constructed inside or outside of the class.

All parameters in the Algebraic *MultiGrid* class can be set externally, including smoothers and coarse grid solver.

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::PairwiseAMG< OperatorType, VectorType, ValueType >*, *rocalution::RugeStuebenAMG< OperatorType, VectorType, ValueType >*, *rocalution::SAAMG< OperatorType, VectorType, ValueType >*, *rocalution::UAAMG< OperatorType, VectorType, ValueType >*

virtual void `rocalution::`*BaseAMG*`::`**BuildHierarchy**(void)

Create AMG hierarchy.

virtual void `rocalution::`*BaseAMG*`::`**BuildSmoothers**(void)

Create AMG smoothers.

void `rocalution::`*BaseAMG*`::`**SetCoarsestLevel**(int coarse_size)

Set coarsest level for hierarchy creation.

void `rocalution::`*BaseAMG*`::`**SetManualSmoothers**(bool sm_manual)

Set flag to pass smoothers manually for each level.

void `rocalution::`*BaseAMG*`::`**SetManualSolver**(bool s_manual)

Set flag to pass coarse grid solver manually.

void `rocalution::`*BaseAMG*`::`**SetDefaultSmootherFormat**(unsigned int op_format)

Set the smoother operator format.

void `rocalution::`*BaseAMG*`::`**SetOperatorFormat**(unsigned int op_format, int op_blockdim)

Set the operator format.

int `rocalution::`*BaseAMG*`::`**GetNumLevels**(void)

Returns the number of levels in hierarchy.

1.6.11 Unsmoothed Aggregation AMG

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **UAAMG** : public rocalution::*BaseAMG*<*OperatorType*, *VectorType*, *ValueType*>

Unsmoothed Aggregation Algebraic *MultiGrid* Method.

The Unsmoothed Aggregation Algebraic *MultiGrid* method is based on unsmoothed aggregation based interpolation scheme. stueben

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

void `rocalution::`*UAAMG*`::`**SetCouplingStrength**(ValueType eps)

Set coupling strength.

void `rocalution::`*UAAMG*`::`**SetOverInterp**(ValueType overInterp)

Set over-interpolation parameter for aggregation.

1.6.12 Smoothed Aggregation AMG

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **SAAMG** : public rocalution::*BaseAMG*<*OperatorType*, *VectorType*, *ValueType*>

Smoothed Aggregation Algebraic *MultiGrid* Method.

The Smoothed Aggregation Algebraic *MultiGrid* method is based on smoothed aggregation based interpolation scheme. vanek

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

void `rocalution::`*SAAMG*`::`**SetCouplingStrength**(ValueType eps)

Set coupling strength.

void `rocalution::`*SAAMG*`::`**SetInterpRelax**(ValueType relax)

Set the relaxation parameter.

1.6.13 Ruge-Stueben AMG

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **RugeStuebenAMG** : public rocalution::*BaseAMG*<*OperatorType*, *VectorType*, *ValueType*>

Ruge-Stueben Algebraic *MultiGrid* Method.

The Ruge-Stueben Algebraic *MultiGrid* method is based on the classic Ruge-Stueben coarsening with direct interpolation. The solver provides high-efficiency in terms of complexity of the solver (i.e. number of iterations). However, most of the time it has a higher building step and requires higher memory usage. stueben

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Warning: doxygenfunction: Cannot find function "rocalution::RugeStuebenAMG::SetCouplingStrength" in doxygen xml output for project "rocALUTION" from directory: ../docBin/xml

1.6.14 Pairwise AMG

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **PairwiseAMG** : public rocalution::*BaseAMG*<*OperatorType*, *VectorType*, *ValueType*>

Pairwise Aggregation Algebraic *MultiGrid* Method.

The Pairwise Aggregation Algebraic *MultiGrid* method is based on a pairwise aggregation matching scheme. It delivers very efficient building phase which is suitable for Poisson-like equation. Most of the time it requires K-cycle for the solving phase to provide low number of iterations. This version has multi-node support. pairwiseamg

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

void `rocalution::PairwiseAMG::`**SetBeta**(ValueType beta)

Set beta for pairwise aggregation.

void `rocalution::PairwiseAMG::`**SetOrdering**(unsigned int ordering)

Set re-ordering for aggregation.

void `rocalution::PairwiseAMG::`**SetCoarseningFactor**(double factor)

Set target coarsening factor.

1.6.15 Direct Linear Solvers

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **DirectLinearSolver** : public rocalution::*Solver*<*OperatorType*, *VectorType*, *ValueType*>

Base class for all direct linear solvers.

The library provides three direct methods - *LU*, *QR* and *Inversion* (based on *QR* decomposition). The user can pass a sparse matrix, internally it will be converted to dense and then the selected method will be applied. These methods are not very optimal and due to the fact that the matrix is converted to a dense format, these methods should be used only for very small matrices.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*

- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::Inversion< OperatorType, VectorType, ValueType >*, *rocalution::LU< OperatorType, VectorType, ValueType >*, *rocalution::QR< OperatorType, VectorType, ValueType >*

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **LU** : public rocalution::*DirectLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

LU Decomposition.

Lower-Upper Decomposition factors a given square matrix into lower and upper triangular matrix, such that $A = LU$.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **QR** : public rocalution::*DirectLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

QR Decomposition.

The *QR* Decomposition decomposes a given matrix into $A = QR$, such that Q is an orthogonal matrix and R an upper triangular matrix.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **Inversion** : public rocalution::*DirectLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Matrix *Inversion*.

Full matrix inversion based on *QR* decomposition.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Note: These methods can only be used with local-type problems.

1.7 Preconditioners

In this chapter, all preconditioners are presented. All preconditioners support local operators. They can be used as a global preconditioner via block-jacobi scheme which works locally on each interior matrix. To provide fast application, all preconditioners require extra memory to keep the approximated operator.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **Preconditioner** : public rocalution::*Solver*<*OperatorType*, *VectorType*, *ValueType*>

Base class for all preconditioners.

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::AIChebyshev< OperatorType, VectorType, ValueType >*, *rocalution::AS< OperatorType, VectorType, ValueType >*, *rocalution::BlockJacobi< OperatorType, VectorType, ValueType >*, *rocalution::BlockPreconditioner< OperatorType, VectorType, ValueType >*, *rocalution::DiagJacobiSaddlePointPrecond< OperatorType, VectorType, ValueType >*, *rocalution::FSAI< OperatorType, VectorType, ValueType >*, *rocalution::GS< OperatorType, VectorType, ValueType >*, *rocalution::IC< OperatorType, VectorType, ValueType >*, *rocalution::ILU< OperatorType, VectorType, ValueType >*, *rocalution::ILUT< OperatorType, VectorType, ValueType >*, *rocalution::Jacobi< OperatorType, VectorType, ValueType >*, *rocalution::MultiColored< OperatorType, VectorType, ValueType >*, *rocalution::MultiElimination< OperatorType, VectorType, ValueType >*, *rocalution::SGS< OperatorType, VectorType, ValueType >*, *rocalution::SPAI< OperatorType, VectorType, ValueType >*, *rocalution::TNS< OperatorType, VectorType, ValueType >*, *rocalution::VariablePreconditioner< OperatorType, VectorType, ValueType >*

1.7.1 Code Structure

The preconditioners provide a solution to the system $Mz = r$, where either the solution z is directly computed by the approximation scheme or it is iteratively obtained with $z = 0$ initial guess.

1.7.2 Jacobi Method

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **Jacobi** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Jacobi Method.

The *Jacobi* method is for solving a diagonally dominant system of linear equations $Ax = b$. It solves for each diagonal element iteratively until convergence, such that

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i}^{n} a_{ij}x_j^{(k)}\right)$$

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Note: Damping parameter ω can be adjusted by *rocalution::FixedPoint::SetRelaxation()*.

1.7.3 (Symmetric) Gauss-Seidel / (S)SOR Method

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **GS** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Gauss-Seidel / Successive Over-Relaxation Method.

The Gauss-Seidel / SOR method is for solving system of linear equations $Ax = b$. It approximates the solution iteratively with

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i}^{n} a_{ij}x_j^{(k)}\right),$$

with $\omega \in (0, 2)$.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **SGS** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Symmetric Gauss-Seidel / Symmetric Successive Over-Relaxation Method.

The Symmetric Gauss-Seidel / SSOR method is for solving system of linear equations $Ax = b$. It approximates the solution iteratively.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Note: Relaxation parameter ω can be adjusted by *rocalution::FixedPoint::SetRelaxation()*.

1.7.4 Incomplete Factorizations

1.7.4.1 ILU

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **ILU** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Incomplete LU Factorization based on levels.

The Incomplete LU Factorization based on levels computes a sparse lower and sparse upper triangular matrix such that $A = LU - R$.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

virtual void rocalution::*ILU*::**Set**(int p, bool level = true)

Initialize ILU(p) factorization.

Initialize ILU(p) factorization based on power. SAAD

- level = true build the structure based on levels
- level = false build the structure only based on the power(p+1)

1.7.4.2 ILUT

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **ILUT** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Incomplete *LU* Factorization based on threshold.

The Incomplete *LU* Factorization based on threshold computes a sparse lower and sparse upper triangular matrix such that $A = LU - R$. Fill-in values are dropped depending on a threshold and number of maximal fill-ins per row. SAAD

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

virtual void rocalution::*ILUT*::**Set**(double t)

Set drop-off threshold.

virtual void rocalution::*ILUT*::**Set**(double t, int maxrow)

Set drop-off threshold and maximum fill-ins per row.

1.7.4.3 IC

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **IC** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Incomplete Cholesky Factorization without fill-ins.

The Incomplete Cholesky Factorization computes a sparse lower triangular matrix such that $A = LL^T - R$. Additional fill-ins are dropped and the sparsity pattern of the original matrix is preserved.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

1.7.5 AI Chebyshev

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **AIChebyshev** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Approximate Inverse - *Chebyshev Preconditioner*.

The Approximate Inverse - *Chebyshev Preconditioner* is an inverse matrix preconditioner with values from a linear combination of matrix-valued *Chebyshev* polynomials. chebpoly

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

void `rocalution::AIChebyshev::Set`(int p, ValueType lambda_min, ValueType lambda_max)

Set order, min and max eigenvalues.

1.7.6 FSAI

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **FSAI** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Factorized Approximate Inverse *Preconditioner*.

The Factorized Sparse Approximate Inverse preconditioner computes a direct approximation of M^{-1} by minimizing the Frobenius norm $||I - GL||_F$, where L denotes the exact lower triangular part of A and $G := M^{-1}$. The *FSAI* preconditioner is initialized by q, based on the sparsity pattern of $|A^q|$. However, it is also possible to supply external sparsity patterns in form of the *LocalMatrix* class. kolotilina

---

Note: The *FSAI* preconditioner is only suited for symmetric positive definite matrices.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

void `rocalution::FSAI::Set`(int power)

Set the power of the system matrix sparsity pattern.

void `rocalution::FSAI::Set`(const OperatorType &pattern)

Set an external sparsity pattern.

void `rocalution::FSAI::SetPrecondMatrixFormat`(unsigned int mat_format, int blockdim = 1)

Set the matrix format of the preconditioner.

1.7.7 SPAI

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **SPAI** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

SParse Approximate Inverse *Preconditioner*.

The SParse Approximate Inverse algorithm is an explicitly computed preconditioner for general sparse linear systems. In its current implementation, only the sparsity pattern of the system matrix is supported. The *SPAI* computation is based on the minimization of the Frobenius norm $||AM - I||_F$. grote

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

void `rocalution::`*SPAI*`::`**SetPrecondMatrixFormat**(unsigned int mat_format, int blockdim = 1)

Set the matrix format of the preconditioner.

1.7.8 TNS

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **TNS** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Truncated Neumann Series *Preconditioner*.

The Truncated Neumann Series (*TNS*) preconditioner is based on $M^{-1} = K^T D^{-1} K$, where $K = (I - LD^{-1} + (LD^{-1})^2)$, with the diagonal D of A and the strictly lower triangular part L of A. The preconditioner can be computed in two forms - explicitly and implicitly. In the explicit form, the full construction of M is performed via matrix-matrix operations, whereas in the implicit from, the application of the preconditioner is based on matrix-vector operations only. The matrix format for the stored matrices can be specified.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

void `rocalution::`*TNS*`::`**Set**(bool imp)

Set implicit (true) or explicit (false) computation.

void `rocalution::`*TNS*`::`**SetPrecondMatrixFormat**(unsigned int mat_format, int blockdim = 1)

Set the matrix format of the preconditioner.

1.7.9 MultiColored Preconditioners

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **MultiColored** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Base class for all multi-colored preconditioners.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::MultiColoredILU< OperatorType, VectorType, ValueType >*, *rocalution::MultiColoredSGS< OperatorType, VectorType, ValueType >*

void `rocalution::MultiColored::SetPrecondMatrixFormat`(unsigned int mat_format, int blockdim = 1)

Set a specific matrix type of the decomposed block matrices.

void `rocalution::MultiColored::SetDecomposition`(bool decomp)

Set if the preconditioner should be decomposed or not.

1.7.9.1 MultiColored (Symmetric) Gauss-Seidel / (S)SOR

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **MultiColoredGS** : public rocalution::*MultiColoredSGS*<*OperatorType*, *VectorType*, *ValueType*>

Multi-Colored Gauss-Seidel / SOR *Preconditioner*.

The Multi-Colored Symmetric Gauss-Seidel / SOR preconditioner is based on the splitting of the original matrix. Higher parallelism in solving the forward substitution is obtained by performing a multi-colored decomposition. Details on the Gauss-Seidel / SOR algorithm can be found in the *GS* preconditioner.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **MultiColoredSGS** : public rocalution::*MultiColored*<*OperatorType*, *VectorType*, *ValueType*>

Multi-Colored Symmetric Gauss-Seidel / SSOR *Preconditioner*.

The Multi-Colored Symmetric Gauss-Seidel / SSOR preconditioner is based on the splitting of the original matrix. Higher parallelism in solving the forward and backward substitution is obtained by performing a multi-colored decomposition. Details on the Symmetric Gauss-Seidel / SSOR algorithm can be found in the *SGS* preconditioner.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::MultiColoredGS< OperatorType, VectorType, ValueType >*

void rocalution::*MultiColoredSGS*::**SetRelaxation**(ValueType omega)

Set the relaxation parameter for the SOR/SSOR scheme.

Note: The preconditioner matrix format can be changed using *rocalution::MultiColored::SetPrecondMatrixFormat()*.

1.7.9.2 MultiColored Power(q)-pattern method ILU(p,q)

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **MultiColoredILU** : public rocalution::*MultiColored*<*OperatorType*, *VectorType*, *ValueType*>

Multi-Colored Incomplete *LU* Factorization *Preconditioner*.

Multi-Colored Incomplete *LU* Factorization based on the ILU(p) factorization with a power(q)-pattern method. This method provides a higher degree of parallelism of forward and backward substitution compared to the standard ILU(p) preconditioner. Lukarski2012

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

void rocalution::*MultiColoredILU*::**Set**(int p)

Initialize a multi-colored *ILU*(p, p+1) preconditioner.

void rocalution::*MultiColoredILU*::**Set**(int p, int q, bool level = true)

Initialize a multi-colored ILU(p, q) preconditioner.

level = true will perform the factorization with levels

level = false will perform the factorization only on the power(q)-pattern

Note: The preconditioner matrix format can be changed using *rocalution::MultiColored::SetPrecondMatrixFormat()*.

1.7.10 Multi-Elimination Incomplete LU

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **MultiElimination** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Multi-Elimination Incomplete *LU* Factorization *Preconditioner*.

The Multi-Elimination Incomplete *LU* preconditioner is based on the following decomposition

$$A = \begin{pmatrix} D & F \\ E & C \end{pmatrix} = \begin{pmatrix} I & 0 \\ ED^{-1} & I \end{pmatrix} \times \begin{pmatrix} D & F \\ 0 & \hat{A} \end{pmatrix},$$

where $\hat{A} = C - ED^{-1}F$. To make the inversion of D easier, we permute the preconditioning before the factorization with a permutation P to obtain only diagonal elements in D. The permutation here is based on a maximal independent set. This procedure can be applied to the block matrix $\hat{A}$, in this way we can perform the factorization recursively. In the last level of the recursion, we need to provide a solution procedure. By the design of the library, this can be any kind of solver. SAAD

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

inline int rocalution::*MultiElimination*::**GetSizeDiagBlock**(void) const

Returns the size of the first (diagonal) block of the preconditioner.

inline int rocalution::*MultiElimination*::**GetLevel**(void) const

Return the depth of the current level.

void rocalution::*MultiElimination*::**Set**(*Solver*<OperatorType, VectorType, ValueType> &AA_Solver, int level, double drop_off = 0.0)

Initialize (recursively) ME-ILU with level (depth of recursion)

AA_Solvers - defines the last-block solver

drop_off - defines drop-off tolerance

void rocalution::*MultiElimination*::**SetPrecondMatrixFormat**(unsigned int mat_format, int blockdim = 1)

Set a specific matrix type of the decomposed block matrices.

1.7.11 Diagonal Preconditioner for Saddle-Point Problems

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **DiagJacobiSaddlePointPrecond** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Diagonal *Preconditioner* for Saddle-Point Problems.

Consider the following saddle-point problem

$$A = \begin{pmatrix} K & F \\ E & 0 \end{pmatrix}.$$

For such problems we can construct a diagonal Jacobi-type preconditioner of type

$$P = \begin{pmatrix} K & 0 \\ 0 & S \end{pmatrix},$$

with $S = ED^{-1}F$, where D are the diagonal elements of K. The matrix S is fully constructed (via sparse matrix-matrix multiplication). The preconditioner needs to be initialized with two external solvers/preconditioners - one for the matrix K and one for the matrix S.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

void rocalution::*DiagJacobiSaddlePointPrecond*::**Set**(*Solver*<OperatorType, VectorType, ValueType> &K_Solver, *Solver*<OperatorType, VectorType, ValueType> &S_Solver)

Initialize solver for K and S.

1.7.12 (Restricted) Additive Schwarz Preconditioner

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **AS** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Additive Schwarz *Preconditioner*.

The Additive Schwarz preconditioner relies on a preconditioning technique, where the linear system $Ax = b$ can be decomposed into small sub-problems based on $A_i = R_i^T A R_i$, where R_i are restriction operators. Those restriction operators produce sub-matrices wich overlap. This leads to contributions from two preconditioners on the overlapped area which are scaled by $1/2$. RAS

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::RAS< OperatorType, VectorType, ValueType >*

void `rocalution::AS::Set`(int nb, int overlap, *Solver*<OperatorType, VectorType, ValueType> **preconds)

Set number of blocks, overlap and array of preconditioners.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **RAS** : public rocalution::*AS*<*OperatorType*, *VectorType*, *ValueType*>

Restricted Additive Schwarz *Preconditioner*.

The Restricted Additive Schwarz preconditioner relies on a preconditioning technique, where the linear system $Ax = b$ can be decomposed into small sub-problems based on $A_i = R_i^T A R_i$, where R_i are restriction operators. The *RAS* method is a mixture of block *Jacobi* and the *AS* scheme. In this case, the sub-matrices contain overlapped areas from other blocks, too. RAS

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

The overlapped area is shown in Fig. 1.11.

1.7.13 Block-Jacobi (MPI) Preconditioner

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **BlockJacobi** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Block-Jacobi *Preconditioner*.

The Block-Jacobi preconditioner is designed to wrap any local preconditioner and apply it in a global block fashion locally on each interior matrix.

Template Parameters

- **OperatorType** – - can be *GlobalMatrix*
- **VectorType** – - can be *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

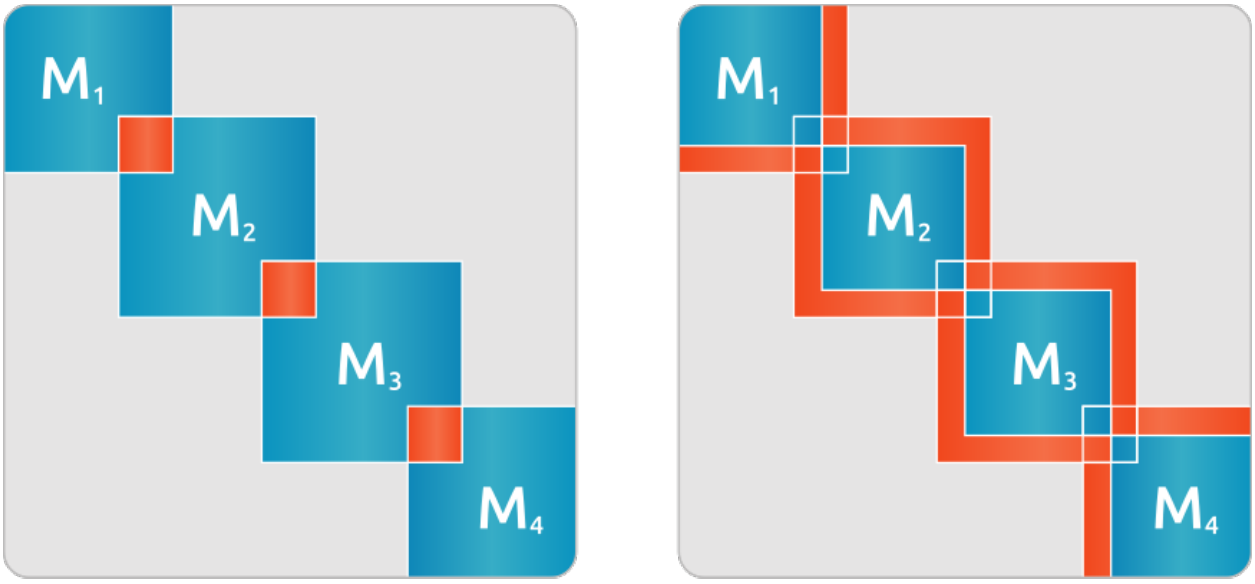

Fig. 1.11: Example of a 4 block-decomposed matrix - Additive Schwarz with overlapping preconditioner (left) and Restricted Additive Schwarz preconditioner (right).

void `rocalution::BlockJacobi::Set`(*Solver*<*LocalMatrix*<ValueType>, *LocalVector*<ValueType>, ValueType> &precond)

Set local preconditioner.

The Block-Jacobi (MPI) preconditioner is shown in Fig. 1.12.

1.7.14 Block Preconditioner

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **BlockPreconditioner** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Block-Preconditioner.

When handling vector fields, typically one can try to use different preconditioners and/or solvers for the different blocks. For such problems, the library provides a block-type preconditioner. This preconditioner builds the following block-type matrix

$$P = \begin{pmatrix} A_d & 0 & . & 0 \\ B_1 & B_d & . & 0 \\ . & . & . & . \\ Z_1 & Z_2 & . & Z_d \end{pmatrix}$$

The solution of P can be performed in two ways. It can be solved by block-lower-triangular sweeps with inversion of the blocks $A_d \dots Z_d$ and with a multiplication of the corresponding blocks. This is set by *SetLSolver()* (which is the default solution scheme). Alternatively, it can be used only with an inverse of the diagonal $A_d \dots Z_d$ (Block-Jacobi type) by using *SetDiagonalSolver()*.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*

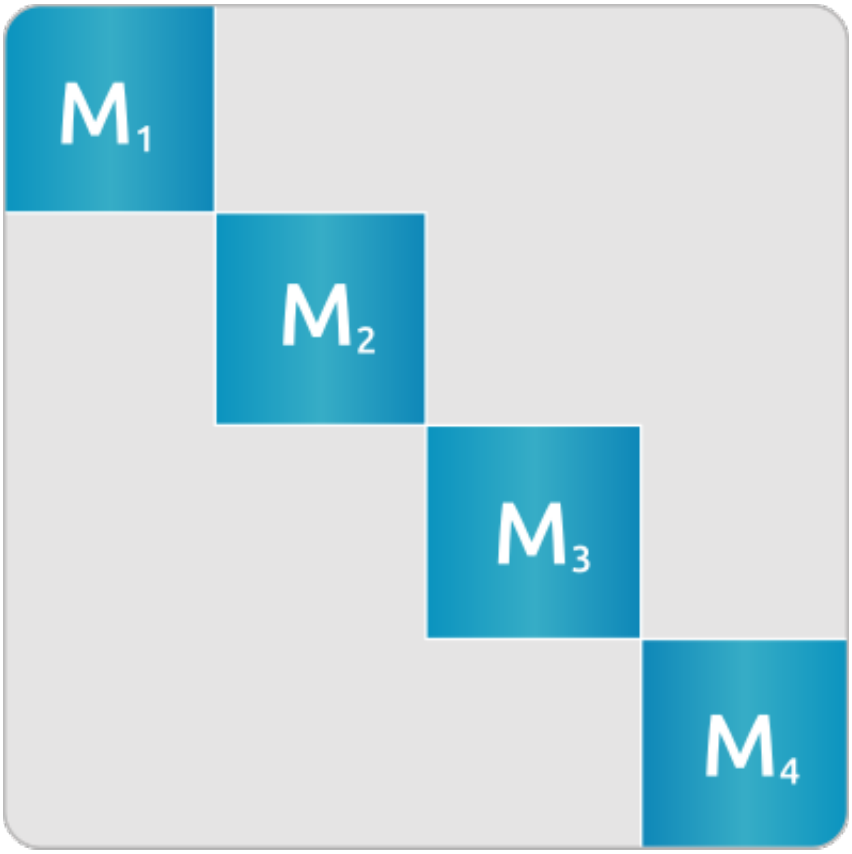

Fig. 1.12: Example of a 4 block-decomposed matrix - Block-Jacobi preconditioner.

- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

void rocalution::*BlockPreconditioner*::**Set**(int n, const int *size, *Solver*<OperatorType, VectorType, ValueType> **D_solver)

Set number, size and diagonal solver.

void rocalution::*BlockPreconditioner*::**SetDiagonalSolver**(void)

Set diagonal solver mode.

void rocalution::*BlockPreconditioner*::**SetLSolver**(void)

Set lower triangular sweep mode.

void rocalution::*BlockPreconditioner*::**SetExternalLastMatrix**(const OperatorType &mat)

Set external last block matrix.

virtual void rocalution::*BlockPreconditioner*::**SetPermutation**(const *LocalVector*<int> &perm)

Set permutation vector.

1.7.15 Variable Preconditioner

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **VariablePreconditioner** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Variable *Preconditioner*.

The Variable *Preconditioner* can hold a selection of preconditioners. Thus, any type of preconditioners can be combined. As example, the variable preconditioner can combine *Jacobi*, *GS* and *ILU* - then, the first iteration of the iterative solver will apply *Jacobi*, the second iteration will apply *GS* and the third iteration will apply *ILU*. After that, the solver will start again with *Jacobi*, *GS*, *ILU*.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

virtual void rocalution::*VariablePreconditioner*::**SetPreconditioner**(int n, *Solver*<OperatorType, VectorType, ValueType> **precond)

Set the preconditioner sequence.

1.8 Backends

The support of accelerator devices is embedded in the structure of rocALUTION. The primary goal is to use this technology whenever possible to decrease the computational time. .. note:: Not all functions are ported and present on the accelerator backend. This limited functionality is natural, since not all operations can be performed efficiently on the accelerators (e.g. sequential algorithms, I/O from the file system, etc.).

Currently, rocALUTION supports HIP capable GPUs starting with ROCm 1.9. Due to its design, the library can be easily extended to support future accelerator technologies. Such an extension of the library will not reflect the algorithms which are based on it.

If a particular function is not implemented for the used accelerator, the library will move the object to the host and compute the routine there. In this case a warning message of level 2 will be printed. For example, if the user wants to

perform an ILUT factorization on the HIP backend which is currently not available, the library will move the object to the host, perform the routine there and print the following warning message

```
*** warning: LocalMatrix::ILUTFactorize() is performed on the host
```

1.8.1 Moving Objects To and From the Accelerator

All objects in rocALUTION can be moved to the accelerator and to the host.

virtual void rocalution::*BaseRocalution*::**MoveToAccelerator**(void) = 0

Move the object to the accelerator backend.

virtual void rocalution::*BaseRocalution*::**MoveToHost**(void) = 0

Move the object to the host backend.

```
LocalMatrix<ValueType> mat;
LocalVector<ValueType> vec1, vec2;

// Perform matrix vector multiplication on the host
mat.Apply(vec1, &vec2);

// Move data to the accelerator
mat.MoveToAccelerator();
vec1.MoveToAccelerator();
vec2.MoveToAccelerator();

// Perform matrix vector multiplication on the accelerator
mat.Apply(vec1, &vec2);

// Move data to the host
mat.MoveToHost();
vec1.MoveToHost();
vec2.MoveToHost();
```

1.8.2 Asynchronous Transfers

The rocALUTION library also provides asynchronous transfers of data between host and HIP backend.

virtual void rocalution::*BaseRocalution*::**MoveToAcceleratorAsync**(void)

Move the object to the accelerator backend with async move.

virtual void rocalution::*BaseRocalution*::**MoveToHostAsync**(void)

Move the object to the host backend with async move.

virtual void rocalution::*BaseRocalution*::**Sync**(void)

Sync (the async move)

This can be done with *rocalution::LocalVector::CopyFromAsync()* and *rocalution::LocalMatrix::CopyFromAsync()* or with *MoveToAcceleratorAsync()* and *MoveToHostAsync()*. These functions return immediately and perform the asynchronous transfer in background mode. The synchronization is done with *Sync()*.

When using the *MoveToAcceleratorAsync()* and *MoveToHostAsync()* functions, the object will still point to its original location (i.e. host for calling *MoveToAcceleratorAsync()* and accelerator for *MoveToHostAsync()*). The object will switch to the new location after the *Sync()* function is called.

Note: The objects should not be modified during an active asynchronous transfer. However, if this happens, the values after the synchronization might be wrong.

Note: To use the asynchronous transfers, you need to enable the pinned memory allocation. Uncomment *#define ROCALUTION_HIP_PINNED_MEMORY* in *src/utils/allocate_free.hpp*.

1.8.3 Systems without Accelerators

rocALUTION provides full code compatibility on systems without accelerators, the user can take the code from the GPU system, re-compile the same code on a machine without a GPU and it will provide the same results. Any calls to `rocalution::BaseRocalution::MoveToAccelerator()` and `rocalution::BaseRocalution::MoveToHost()` will be ignored.

1.8.4 Memory Allocations

All data which is passed to and from rocALUTION is using the memory handling functions described in the code. By default, the library uses standard C++ *new* and *delete* functions for the host data. This can be changed by modifying *src/utils/allocate_free.cpp*.

1.8.4.1 Allocation Problems

If the allocation fails, the library will report an error and exits. If the user requires a special treatment, it has to be placed in *src/utils/allocate_free.cpp*.

1.8.4.2 Memory Alignment

The library can also handle special memory alignment functions. This feature need to be uncommented before the compilation process in *src/utils/allocate_free.cpp*.

1.8.4.3 Pinned Memory Allocation (HIP)

By default, the standard host memory allocation is realized by C++ *new* and *delete*. For faster PCI-Express transfers on HIP backend, the user can also use pinned host memory. This can be activated by uncommenting the corresponding macro in *src/utils/allocate_free.hpp*.

1.9 Remarks

1.9.1 Performance

- Solvers can be built on the accelerator. In many cases, this is faster compared to building on the host.
- Small-sized problems tend to perform better on the host (CPU), due to the good caching system, while large-sized problems typically perform better on the accelerator devices.
- Avoid accessing vectors using [] operators. Use techniques based on *rocalution::LocalVector::SetDataPtr()* and *rocalution::LocalVector::LeaveDataPtr()* instead.
- By default, the OpenMP backend picks the maximum number of threads available. However, if your CPU supports SMT, it will allow to run two times more threads than number of cores. This, in many cases, leads to lower performance. You may observe a performance increase by setting the number of threads (see *rocalution::set_omp_threads_rocalution()*) equal to the number of physical cores.
- If you need to solve a system with multiple right-hand-sides, avoid constructing the solver/preconditioner every time.
- If you are solving similar linear systems, you might want to consider to use the same preconditioner to avoid long building phases.
- In most of the cases, the classical CSR matrix format performs very similar to all other formats on the CPU. On accelerators with many-cores (such as GPUs), formats such as DIA and ELL typically perform better. However, for general sparse matrices one could use HYB format to avoid large memory overhead. The multi-colored preconditioners can be performed in ELL for most of the matrices.
- Not all matrix conversions are performed on the device, the platform will give you a warning if the object need to be moved.
- If you are deploying the rocALUTION library into another software framework try to design your integration functions to avoid *rocalution::init_rocalution()* and *rocalution::stop_rocalution()* every time you call a solver in the library.
- Be sure to compile the library with the correct optimization level (-O3).
- Check, if your solver is really performed on the accelerator by printing the matrix information (*rocalution::BaseRocalution::Info()*) just before calling the *rocalution::Solver::Solve()* function.
- Check the configuration of the library for your hardware with *rocalution::info_rocalution()*.
- Mixed-Precision defect correction technique is recommended for accelerators (e.g. GPUs) with partial or no double precision support. The stopping criteria for the inner solver has to be tuned well for good performance.

1.9.2 Accelerators

- Avoid PCI-Express communication whenever possible (such as copying data from/to the accelerator). Also check the internal structure of the functions.
- Pinned memory allocation (page-locked) can be used for all host memory allocations when using the HIP backend. This provides faster transfers over the PCI-Express and allows asynchronous data movement. By default, this option is disabled. To enable the pinned memory allocation uncomment *#define ROCALUTION_HIP_PINNED_MEMORY* in file *src/utils/allocate_free.hpp*.
- Asynchronous transfers are available for the HIP backend.

1.9.3 Correctness

- If you are assembling or modifying your matrix, you can check it in octave/MATLAB by just writing it into a matrix-market file and read it via *mmread()* function. You can also input a MATLAB/octave matrix in such a way.
- Be sure, to set the correct relative and absolute tolerance values for your problem.
- Check the computation of the relative stopping criteria, if it is based on $|b-Ax^k|_2/|b-Ax^0|_2$ or $|b-Ax^k|_2/|b|_2$.
- Solving very ill-conditioned problems by iterative methods without a proper preconditioning technique might produce wrong results. The solver could stop by showing a low relative tolerance based on the residual but this might be wrong.
- Building the Krylov subspace for many ill-conditioned problems could be a tricky task. To ensure orthogonality in the subspace you might want to perform double orthogonalization (i.e. re-orthogonalization) to avoid rounding errors.
- If you read/write matrices/vectors from files, check the ASCII format of the values (e.g. 34.3434 or 3.43434E + 01).

1.10 Supported Targets

Currently, rocALUTION is supported under the following operating systems

- Ubuntu 16.04, Ubuntu 18.04
- CentOS 7
- SLES 15

To compile and run rocALUTION with HIP support, AMD ROCm Platform 2.9 or newer is required.

The following HIP devices are currently supported

- gfx803 (e.g. Fiji)
- gfx900 (e.g. Vega10, MI25)
- gfx906 (e.g. Vega20, MI50, MI60)
- gfx908

CHAPTER

TWO

DESIGN DOCUMENTATION

2.1 Design and Philosophy

rocALUTION is written in C++ and HIP.

The main idea of the rocALUTION objects is that they are separated from the actual hardware specification. Once you declare a matrix, a vector or a solver they are initially allocated on the host (CPU). Then, every object can be moved to a selected accelerator by a simple function call. The whole execution mechanism is based on run-time type information (RTTI), which allows you to select where and how you want to perform the operations at run time. This is in contrast to the template-based libraries, which need this information at compile time.

The philosophy of the library is to abstract the hardware-specific functions and routines from the actual program, that describes the algorithm. It is hard and almost impossible for most of the large simulation software based on sparse computation, to adapt and port their implementation in order to use every new technology. On the other hand, the new high performance accelerators and devices have the capability to decrease the computational time significantly in many critical parts.

This abstraction layer of the hardware specific routines is the core of the rocALUTION design. It is built to explore fine-grained level of parallelism suited for multi/many-core devices. This is in contrast to most of the parallel sparse libraries available which are mainly based on domain decomposition techniques. Thus, the design of the iterative solvers the preconditioners is very different. Another cornerstone of rocALUTION is the native support of accelerators - the memory allocation, transfers and specific hardware functions are handled internally in the library.

rocALUTION helps you to use accelerator technologies but does not force you to use them. Even if you offload your algorithms and solvers to the accelerator device, the same source code can be compiled and executed in a system without any accelerator.

Naturally, not all routines and algorithms can be performed efficiently on many-core systems (i.e. on accelerators). To provide full functionality, the library has internal mechanisms to check if a particular routine is implemented on the accelerator. If not, the object is moved to the host and the routine is computed there. This guarantees that your code will run with any accelerator, regardless of the available functionality for it.

2.2 Library Source Code Organization

2.2.1 Library Source Code Organization

The rocALUTION library is split into three major parts:

- The *src/base/* directory contains all source code that is built on top of the `BaseRocalution` object as well as the backend structure.
- *src/solvers/* contains all solvers, preconditioners and its control classes.

- In *src/utils/* memory (de)allocation, logging, communication, timing and math helper functions are placed.

2.2.1.1 The *src/base/* directory

Backend Manager

The support of accelerator devices is embedded in the structure of rocALUTION. The primary goal is to use this technology whenever possible to decrease the computational time.

Each technology has its own backend implementation, dealing with platform specific initialization, synchronization, reservation, etc. functionality. Currently available backends are for CPU (naive, OpenMP, MPI) and GPU (HIP).

Note: Not all functions are ported and present on the accelerator backend. This limited functionality is natural, since not all operations can be performed efficiently on the accelerators (e.g. sequential algorithms, I/O from the file system, etc.).

The Operator and Vector classes

The `Operator` and `Vector` classes and its derived local and global classes, are the classes available by the rocALUTION API. While granting the user access to all relevant functionality, all hardware relevant implementation details are hidden. Those linear operators and vectors are the main objects in rocALUTION. They can be moved to an accelerator at run time.

The linear operators are defined as local or global matrices (i.e. on a single node or distributed/multi-node) and local stencils (i.e. matrix-free linear operations). The only template parameter of the operators and vectors is the data type (ValueType). Fig. 2.1 gives an overview of supported operators and vectors.

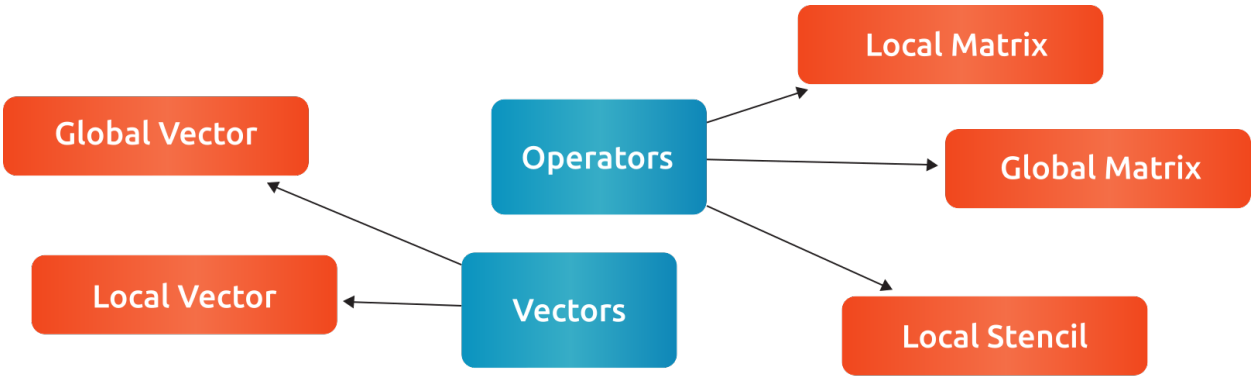

Fig. 2.1: Operator and vector classes.

Each of the objects contain a local copy of the hardware descriptor created by the `init_rocalution` function. Additionally, each local object that is derived from an operator or vector, contains a pointer to a *Base*-class, a *Host*-class and an *Accelerator*-class of same type (e.g. a `LocalMatrix` contains pointers to a `BaseMatrix`, `HostMatrix` and `AcceleratorMatrix`). The *Base*-class pointer will always point towards either the *Host*-class or the *Accelerator*-class pointer, dependend on the runtime decision of the local object. *Base*-classes and their derivatives are further explained in *The BaseMatrix and BaseVector classes*.

Furthermore, each global object, derived from an operator or vector, embeds two *Local*-classes of same type to store the interior and ghost part of the global object (e.g. a `GlobalVector` contains two `LocalVector`). For more details on distributed data structures, see the user manual.

The BaseMatrix and BaseVector classes

The *data* is an object, pointing to the BaseMatrix class. The pointing is coming from either a HostMatrix or an AcceleratorMatrix. The AcceleratorMatrix is created by an object with an implementation in the backend and a matrix format. Switching between host and accelerator matrices is performed in the LocalMatrix class. The LocalVector is organized in the same way.

Each matrix format has its own class for the host and for the accelerator backend. All matrix classes are derived from the BaseMatrix, which provides the base interface for computation as well as for data accessing.

Each local object contains a pointer to a *Base*-class object. While the *Base*-class is mainly pure virtual, their derivatives implement all platform specific functionality. Each of them is coupled to a rocALUTION backend descriptor. While the `HostMatrix`, `HostStencil` and `HostVector` classes implements all host functionality, `AcceleratorMatrix`, `AcceleratorStencil` and `AcceleratorVector` contain accelerator related device code. Each of the backend specializations are located in a different directory, e.g. *src/base/host* for host related classes and *src/base/hip* for accelerator / HIP related classes.

ParallelManager

The parallel manager class handles the communication and the mapping of the global operators. Each global operator and vector need to be initialized with a valid parallel manager in order to perform any operation. For many distributed simulations, the underlying operator is already distributed. This information need to be passed to the parallel manager. All communication functionality for the implementation of global algorithms is available in the rocALUTION communicator in *src/utils/communicator.hpp*. For more details on distributed data structures, see the user manual.

2.2.1.2 The *src/solvers/* directory

The `Solver` and its derived classes can be found in *src/solvers*. The directory structure is further split into the sub-classes `DirectLinearSolver` in *src/solvers/direct*, `IterativeLinearSolver` in *src/solvers/krylov*, `BaseMultiGrid` in *src/solvers/multigrid* and `Preconditioner` in *src/solvers/preconditioners*. Each of the solver is using an `Operator`, `Vector` and data type as template parameters to solve a linear system of equations. The actual solver algorithm is implemented by the `Operator` and `Vector` functionality.

Most of the solvers can be performed on linear operators, e.g. `LocalMatrix`, `LocalStencil` and `GlobalMatrix` - i.e. the solvers can be performed locally (on a shared memory system) or in a distributed manner (on a cluster) via MPI. All solvers and preconditioners need three template parameters - Operators, Vectors and Scalar type. The Solver class is purely virtual and provides an interface for

- `SetOperator` to set the operator, i.e. the user can pass the matrix here.
- `Build` to build the solver (including preconditioners, sub-solvers, etc.). The user need to specify the operator first before building the solver.
- `Solve` to solve the sparse linear system. The user need to pass a right-hand side and a solution / initial guess vector.
- `Print` to show solver information.
- `ReBuildNumeric` to only re-build the solver numerically (if possible).
- `MoveToHost` and `MoveToAccelerator` to offload the solver (including preconditioners and sub-solvers) to the host / accelerator.

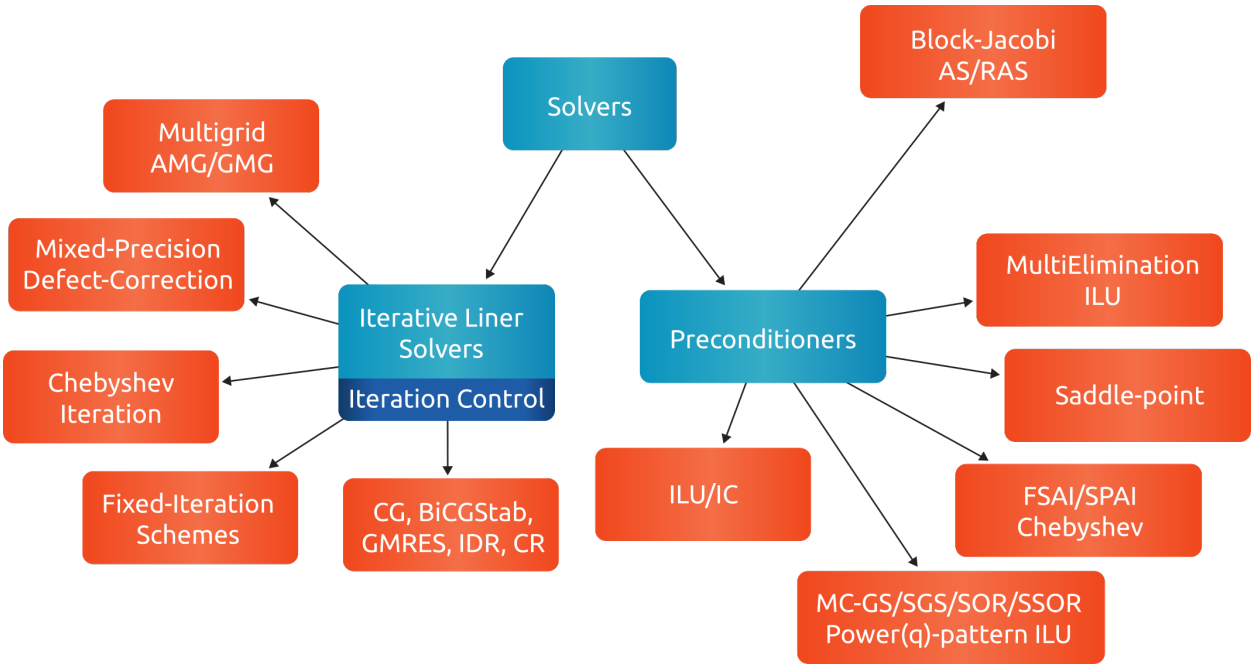

Fig. 2.2: Solver and preconditioner classes.

2.2.1.3 The *src/utils/* directory

In the *src/utils* directory, all commonly used host (de)allocation, timing, math, communication and logging functionality is gathered.

Furthermore, the rocALUTION *GlobalType*, which is the indexing type for global, distributed structures, can be adjusted in *src/utils/types.hpp*. By default, rocALUTION uses 64-bit wide global indexing.

Note: It is not recommended to switch to 32-bit global indexing.

In *src/utils/def.hpp*

- verbosity level *VERBOSE_LEVEL* can be adjusted, see *Verbose Output*,
- debug mode *DEBUG_MODE* can be enabled, see *Debug Output*,
- MPI logging *LOG_MPI_RANK* can be modified, see *Verbose Output and MPI*,
- and object tracking *OBJ_TRACKING_OFF* can be enabled, see *Automatic Object Tracking*.

2.3 Functionality Extension Guidelines

The main purpose of this chapter is to give an overview of different ways to implement user-specific routines, solvers or preconditioners to the rocALUTION library package. Additional features can be added in multiple ways. Additional solver and preconditioner functionality that uses already implemented backend functionality will perform well on accelerator devices without the need for expert GPU programming knowledge. Also, users that are not interested in using accelerators will not be confronted with HIP and GPU related programming tasks to add additional functionality.

In the following sections, different levels of functionality enhancements are illustrated. These examples can be used as guidelines to extend rocALUTION step by step with your own routines. Please note, that user added routines can also

be added to the main GitHub repository using pull requests.

2.3.1 LocalMatrix Functionlity Extension

In this example, the *LocalMatrix* class is extended by an additional routine. The routine shall support both, Host and Accelerator backend. Furthermore, the routine requires the matrix to be in CSR format.

2.3.1.1 API Enhancement

To make the new routine available by the API, we first need to modify the *LocalMatrix* class. The corresponding header file *local_matrix.hpp* is located in *src/base/*. The new routines can be added as public member function, e.g.

```
...
void ConvertTo(unsigned int matrix_format, int blockdim);

void MyNewFunctionality(void);

virtual void Apply(const LocalVector<ValueType>& in, LocalVector<ValueType>* out) const;
virtual void ApplyAdd(const LocalVector<ValueType>& in,
...
```

For the implementation of the new API function, it is important to know where this functionality will be available. To add support for any backend and matrix format, format conversions are required, if *MyNewFunctionality()* is only supported for CSR matrices. This will be subject to the API function implementation:

```
template <typename ValueType>
void LocalMatrix<ValueType>::MyNewFunctionality(void)
{
    // Debug logging
    log_debug(this, "LocalMatrix::MyNewFunctionality()");

#ifdef DEBUG_MODE
    // If we are in debug mode, perform an additional matrix sanity check
    this->Check();
#endif

    // If no non-zero entries, do nothing
    if(this->GetNnz() > 0)
    {
        // As we want to implement our function only for CSR, we first need to convert
        // the matrix to CSR format
        unsigned int format = this->GetFormat();
        int blockdim = this->GetBlockDimension();
        this->ConvertToCSR();

        // Call the corresponding base matrix implementation
        bool err = this->matrix_->MyNewFunctionality();

        // Check its return type
        if((err == false) && (this->is_host_() == true))
        {
            // If our matrix is on the host, the function call failed.
```

(continues on next page)

(continued from previous page)

```
            LOG_INFO("Computation of LocalMatrix::MyNewFunctionality() failed");
            this->Info();
            FATAL_ERROR(__FILE__, __LINE__);
        }

        // Run backup algorithm on host, in case the accelerator version failed
        if(err == false)
        {
            // Move matrix to host
            bool is_accel = this->is_accel_();
            this->MoveToHost();

            // Try again
            if(this->matrix_->MyNewFunctionality() == false)
            {
                LOG_INFO("Computation of LocalMatrix::MyNewFunctionality() failed");
                this->Info();
                FATAL_ERROR(__FILE__, __LINE__);
            }

            // On a successful host call, move the data back to the accelerator
            // if initial data was on the accelerator
            if(is_accel == true)
            {
                // Print a warning, that the algorithm was performed on the host
                // even though the initial data was on the device
                LOG_VERBOSE_INFO(2, "*** warning: LocalMatrix::MyNewFunctionality() was␣
↪performed on the host");

                this->MoveToAccelerator();
            }
        }

        // Convert the matrix back to CSR format
        if(format != CSR)
        {
            // Print a warning, that the algorithm was performed in CSR format
            // even though the initial matrix format was different
            LOG_VERBOSE_INFO(2, "*** warning: LocalMatrix::MyNewFunctionality() was␣
↪performed in CSR format");

            this->ConvertTo(format, blockdim);
        }
    }

#ifdef DEBUG_MODE
    // Perform additional sanity check in debug mode, because this is a non-const␣
↪function
    this->Check();
#endif
}
```

Similarly, host-only functions can be implemented. In this case, initial data explicitly need to be moved to the host

backend by the API implementation.

The next step is the implementation of the actual functionality in the *BaseMatrix* class.

2.3.1.2 Enhancement of the BaseMatrix class

To make the new routine available in the base class, we first need to modify the *BaseMatrix* class. The corresponding header file *base_matrix.hpp* is located in *src/base/*. The new routines can be added as public member function, e.g.

```
...
virtual bool ILU0Factorize(void);

/// Perform MyNewFunctionality algorithm
virtual bool MyNewFunctionality(void);

/// Perform LU factorization
...
```

We do not implement *MyNewFunctionality()* purely virtual, as we do not supply an implementation for all base classes. We decided to implement it only for CSR format, and thus need to return an error flag, such that the *LocalMatrix* class is aware of the failure and can convert it to CSR.

```
template <typename ValueType>
bool MyNewFunctionality(void)
{
    return false;
}
```

Platform-specific Host Implementation

So far, our new function will always fail, as there is no backend implementation available yet. To satisfy the rocALUTION host backup philosophy, we need to make sure that there is always a host implementation available. This host implementation need to be placed in *src/base/host/host_matrix_csr.cpp* as we decided to make it available for CSR format.

```
...
virtual bool ILUTFactorize(double t, int maxrow);

virtual bool MyNewFunctionality(void);

virtual void LUAnalyse(void);
...
```

```
template <typename ValueType>
bool HostMatrixCSR<ValueType>::MyNewFunctionality(void)
{
    // Place some asserts to verify sanity of input data

    // Our algorithm works only for squared matrices
    assert(this->nrow_ == this->ncol_);
    assert(this->nnz_ > 0);

```

(continues on next page)

(continued from previous page)

```
    // place the actual host based algorithm here:
    // for illustration, we scale the matrix by its inverse diagonal
    for(int i = 0; i < this->nrow_; ++i)
    {
        int row_begin = this->mat_.row_offset[i];
        int row_end   = this->mat_.row_offset[i + 1];

        bool diag_found = false;
        ValueType inv_diag;

        // Find the diagonal entry
        for(int j = row_begin; j < row_end; ++j)
        {
            if(this->mat_.col[j] == i)
            {
                diag_found = true;
                inv_diag = static_cast<ValueType>(1) / this->mat_.val[j];
            }
        }

        // Our algorithm works only with full rank
        assert(diag_found == true);

        // Scale the row
        for(int j = row_begin; j < row_end; ++j)
        {
            this->mat_.val[j] *= inv_diag;
        }
    }

    return true;
}
```

Platform-specific HIP Implementation

We can now add an additional implementation for the HIP backend, using HIP programming framework. This will make our algorithm available on accelerators and rocALUTION will not switch to the host backend on function calls anymore. The HIP implementation needs to be added to *src/base/hip/hip_matrix_csr.cpp* in this case.

```
...
virtual bool ILU0Factorize(void);

virtual bool MyNewFunctionality(void);

virtual bool ICFactorize(BaseVector<ValueType>* inv_diag = NULL);
...
```

```
template <typename ValueType>
bool HIPAcceleratorMatrixCSR<ValueType>::MyNewFunctionality(void)
{
```

(continues on next page)

(continued from previous page)

```
    // Place some asserts to verify sanity of input data

    // Our algorithm works only for squared matrices
    assert(this->nrow_ == this->ncol_);
    assert(this->nnz_ > 0);

    // Enqueue the HIP kernel
    hipLaunchKernelGGL((kernel_csr_mynewfunctionality),
                       dim3((this->nrow_ - 1) / this->local_backend_.HIP_block_size + 1),
                       dim3(this->local_backend_.HIP_block_size),
                       0,
                       0,
                       this->mat_.row_offset,
                       this->mat_.col,
                       this->mat_.val);

    // Check for HIP execution error before successfully returning
    CHECK_HIP_ERROR(__FILE__, __LINE__);

    return true;
}
```

The corresponding HIP kernel should be placed in *src/base/hip/hip_kernels_csr.hpp*.

2.3.2 Adding a Solver

In this example, a new solver shall be added to rocALUTION.

2.3.2.1 API Enhancement

First, the API for the new solver must be defined. In this example, a new `IterativeLinearSolver` is added. To achieve this, the `CG` is a good template. Thus, we first copy *src/solvers/krylov/cg.hpp* to *src/solvers/krylov/mysolver.hpp* and *src/solvers/krylov.cg.cpp* to *src/solvers/krylov/mysolver.cpp* (assuming we add a krylov subspace solvers).

Next, modify the *cg.hpp* and *cg.cpp* to your needs (e.g. change the solver name from *CG* to *MySolver*). Each of the virtual functions in the class need an implementation.

- **MySolver()**: The constructor of the new solver class.
- **~MySolver()**: The destructor of the new solver class. It should call the *Clear()* function.
- **void Print(void) const**: This function should print some informations about the solver.
- **void Build(void)**: This function creates all required structures of the solver, e.g. allocates memory and sets the backend of temporary objects.
- **void BuildMoveToAcceleratorAsync(void)**: This function should moves all solver related objects asynchronously to the accelerator device.
- **void Sync(void)**: This function should synchronize all solver related objects.
- **void ReBuildNumeric(void)**: This function should re-build the solver only numerically.
- **void Clear(void)**: This function should clean up all solver relevant structures that have been created using *Build()*.

- **void SolveNonPrecond_(const VectorType& rhs, VectorType* x)**: This function should perform the solving phase $Ax=y$ without the use of a preconditioner.
- **void SolvePrecond_(const VectorType& rhs, VectorType* x)**: This function should perform the solving phase $Ax=y$ with the use of a preconditioner.
- **void PrintStart_(void) const**: This protected function is called upton solver start.
- **void PrintEnd_(void) const**: This protected function is called when the solver ends.
- **void MoveToHostLocalData_(void)**: This protected function should move all local solver objects to the host.
- **void MoveToAcceleratorLocalData_(void)**: This protected function should move all local solver objects to the accelerator.

Of course, additional member functions that are solver specific, can be introduced.

Then, to make the new solver visible, we have to add it to the *src/rocalution.hpp* header:

```
...
#include "solvers/krylov/cg.hpp"
#include "solvers/krylov/mysolver.hpp"
#include "solvers/krylov/cr.hpp"
...
```

Finally, the new solver must be added to the CMake compilation list, found in *src/solvers/CMakeLists.txt*:

```
...
set(SOLVERS_SOURCES
  solvers/krylov/cg.cpp
  solvers/krylov/mysolver.cpp
  solvers/krylov/fcg.cpp
...
```

2.4 Functionality Table

The following tables give an overview whether a rocALUTION routine is implemented on host backend, accelerator backend, or both.

2.4.1 LocalMatrix and LocalVector classes

All matrix operations (except SpMV) require a CSR matrix.

Note: If the input matrix is not a CSR matrix, an internal conversion will be performed to CSR format, followed by a back conversion to the previous format after the operation. In this case, a warning message on verbosity level 2 will be printed.

LocalMatrix function	Comment	Host	HIP
GetFormat	Obtain the matrix format	Yes	Yes
Check	Check the matrix for structure and value validity	Yes	No
AllocateCSR	Allocate CSR matrix	Yes	Yes
AllocateBCSR	Allocate BCSR matrix	Yes	Yes

continues on next page

Table 2.1 – continued from previous page

LocalMatrix function	Comment	Host	HIP
AllocateMCSR	Allocate MCSR matrix	Yes	Yes
AllocateCOO	Allocate COO matrix	Yes	Yes
AllocateDIA	Allocate DIA matrix	Yes	Yes
AllocateELL	Allocate ELL matrix	Yes	Yes
AllocateHYB	Allocate HYB matrix	Yes	Yes
AllocateDENSE	Allocate DENSE matrix	Yes	Yes
SetDataPtrCSR	Initialize matrix with externally allocated CSR data	Yes	Yes
SetDataPtrMCSR	Initialize matrix with externally allocated MCSR data	Yes	Yes
SetDataPtrCOO	Initialize matrix with externally allocated COO data	Yes	Yes
SetDataPtrDIA	Initialize matrix with externally allocated DIA data	Yes	Yes
SetDataPtrELL	Initialize matrix with externally allocated ELL data	Yes	Yes
SetDataPtrDENSE	Initialize matrix with externally allocated DENSE data	Yes	Yes
LeaveDataPtrCSR	Direct Memory access	Yes	Yes
LeaveDataPtrMCSR	Direct Memory access	Yes	Yes
LeaveDataPtrCOO	Direct Memory access	Yes	Yes
LeaveDataPtrDIA	Direct Memory access	Yes	Yes
LeaveDataPtrELL	Direct Memory access	Yes	Yes
LeaveDataPtrDENSE	Direct Memory access	Yes	Yes
Zeros	Set all matrix entries to zero	Yes	Yes
Scale	Scale all matrix non-zeros	Yes	Yes
ScaleDiagonal	Scale matrix diagonal	Yes	Yes
ScaleOffDiagonal	Scale matrix off-diagonal entries	Yes	Yes
AddScalar	Add scalar to all matrix non-zeros	Yes	Yes
AddScalarDiagonal	Add scalar to matrix diagonal	Yes	Yes
AddScalarOffDiagonal	Add scalar to matrix off-diagonal entries	Yes	Yes
ExtractSubMatrix	Extract sub-matrix	Yes	Yes
ExtractSubMatrices	Extract array of non-overlapping sub-matrices	Yes	Yes
ExtractDiagonal	Extract matrix diagonal	Yes	Yes
ExtractInverseDiagonal	Extract inverse matrix diagonal	Yes	Yes
ExtractL	Extract lower triangular matrix	Yes	Yes
ExtractU	Extract upper triangular matrix	Yes	Yes
Permute	(Forward) permute the matrix	Yes	Yes
PermuteBackward	(Backward) permute the matrix	Yes	Yes
CMK	Create CMK permutation vector	Yes	No
RCMK	Create reverse CMK permutation vector	Yes	No
ConnectivityOrder	Create connectivity (increasing nnz per row) permutation vector	Yes	No
MultiColoring	Create multi-coloring decomposition of the matrix	Yes	No
MaximalIndependentSet	Create maximal independent set decomposition of the matrix	Yes	No
ZeroBlockPermutation	Create permutation where zero diagonal entries are mapped to the last block	Yes	No
ILU0Factorize	Create ILU(0) factorization	Yes	No
LUFactorize	Create LU factorization	Yes	No
ILUTFactorize	Create ILU(t,m) factorization	Yes	No
ILUpFactorize	Create ILU(p) factorization	Yes	No
ICFactorize	Create IC factorization	Yes	No
QRDecompose	Create QR decomposition	Yes	No
ReadFileMTX	Read matrix from matrix market file	Yes	No
WriteFileMTX	Write matrix to matrix market file	Yes	No
ReadFileCSR	Read matrix from binary file	Yes	No
WriteFileCSR	Write matrix to binary file	Yes	No

continues on next page

Table 2.1 – continued from previous page

LocalMatrix function	Comment	Host	HIP
CopyFrom	Copy matrix (values and structure) from another LocalMatrix	Yes	Yes
CopyFromAsync	Copy matrix asynchronously	Yes	Yes
CloneFrom	Clone an entire matrix (values, structure and backend) from another LocalMatrix	Yes	Yes
UpdateValuesCSR	Update CSR matrix values (structure remains identical)	Yes	Yes
CopyFromCSR	Copy (import) CSR matrix	Yes	Yes
CopyToCSR	Copy (export) CSR matrix	Yes	Yes
CopyFromCOO	Copy (import) COO matrix	Yes	Yes
CopyToCOO	Copy (export) COO matrix	Yes	Yes
CopyFromHostCSR	Allocate and copy (import) a CSR matrix from host	Yes	No
ConvertToCSR	Convert a matrix to CSR format	Yes	No
ConvertToMCSR	Convert a matrix to MCSR format	Yes	No
ConvertToBCSR	Convert a matrix to BCSR format	Yes	No
ConvertToCOO	Convert a matrix to COO format	Yes	Yes
ConvertToELL	Convert a matrix to ELL format	Yes	Yes
ConvertToDIA	Convert a matrix to DIA format	Yes	Yes
ConvertToHYB	Convert a matrix to HYB format	Yes	Yes
ConvertToDENSE	Convert a matrix to DENSE format	Yes	No
ConvertTo	Convert a matrix	Yes	
SymbolicPower	Perform symbolic power computation (structure only)	Yes	No
MatrixAdd	Matrix addition	Yes	No
MatrixMult	Multiply two matrices	Yes	No
DiagonalMatrixMult	Multiply matrix with diagonal matrix (stored in LocalVector)	Yes	Yes
DiagonalMatrixMultL	Multiply matrix with diagonal matrix (stored in LocalVector) from left	Yes	Yes
DiagonalMatrixMultR	Multiply matrix with diagonal matrix (stored in LocalVector) from right	Yes	Yes
Gershgorin	Compute the spectrum approximation with Gershgorin circles theorem	Yes	No
Compess	Delete all entries where *abs(a_ij) <= drop_off*	Yes	Yes
Transpose	Transpose the matrix	Yes	No
Sort	Sort the matrix indices	Yes	No
Key	Compute a unique matrix key	Yes	No
ReplaceColumnVector	Replace a column vector of a matrix	Yes	No
ReplaceRowVector	Replace a row vector of a matrix	Yes	No
ExtractColumnVector	Extract a column vector of a matrix	Yes	No
ExtractRowVector	Extract a row vector of a matrix	Yes	No

LocalVector function	Comment	Host	HIP
GetSize	Obtain vector size	Yes	Yes
Check	Check vector for valid entries	Yes	No
Allocate	Allocate vector	Yes	Yes
Sync	Synchronize	Yes	Yes
SetDataPtr	Initialize vector with external data	Yes	Yes
LeaveDataPtr	Direct Memory Access	Yes	Yes
Zeros	Set vector entries to zero	Yes	Yes
Ones	Set vector entries to one	Yes	Yes
SetValues	Set vector entries to scalar	Yes	Yes
SetRandomUniform	Initialize vector with uniformly distributed random numbers	Yes	No
SetRandomNormal	Initialize vector with normally distributed random numbers	Yes	No
ReadFileASCII	Read vector for ASCII file	Yes	No
WriteFileASCII	Write vector to ASCII file	Yes	No

continues on next page

Table 2.2 – continued from previous page

LocalVector function	Comment	Host	HIP
ReadFileBinary	Read vector from binary file	Yes	No
WriteFileBinary	Write vector to binary file	Yes	No
CopyFrom	Copy vector (values) from another LocalVector	Yes	Yes
CopyFromAsync	Copy vector asynchronously	Yes	Yes
CopyFromFloat	Copy vector from another LocalVector<float>	Yes	Yes
CopyFromDouble	Copy vector from another LocalVector<double>	Yes	Yes
CopyFromPermute	Copy vector under specified (forward) permutation	Yes	Yes
CopyFromPermuteBackward	Copy vector under specified (backward) permutation	Yes	Yes
CloneFrom	Clone vector (values and backend descriptor) from another LocalVector	Yes	Yes
CopyFromData	Copy (import) vector from array	Yes	Yes
CopyToData	Copy (export) vector to array	Yes	Yes
Permute	(Foward) permute vector in-place	Yes	Yes
PermuteBackward	(Backward) permute vector in-place	Yes	Yes
AddScale	$y = a * x + y$	Yes	Yes
ScaleAdd	$y = x + a * y$	Yes	Yes
ScaleAddScale	$y = b * x + a * y$	Yes	Yes
ScaleAdd2	$z = a * x + b * y + c * z$	Yes	Yes
Scale	$x = a * x$	Yes	Yes
ExclusiveScan	Compute exclusive sum	Yes	No
Dot	Compute dot product	Yes	Yes
DotNonConj	Compute non-conjugated dot product	Yes	Yes
Norm	Compute L2 norm	Yes	Yes
Reduce	Obtain the sum of all vector entries	Yes	Yes
Asum	Obtain the absolute sum of all vector entries	Yes	Yes
Amax	Obtain the absolute maximum entry of the vector	Yes	Yes
PointWiseMult	Perform point wise multiplication of two vectors	Yes	Yes
Power	Compute vector power	Yes	Yes

2.4.2 Solver and Preconditioner classes

Note: The building phase of the iterative solver also depends on the selected preconditioner.

Solver	Functionality	Host	HIP
CG	Building	Yes	Yes
CG	Solving	Yes	Yes
FCG	Building	Yes	Yes
FCG	Solving	Yes	Yes
CR	Building	Yes	Yes
CR	Solving	Yes	Yes
BiCGStab	Building	Yes	Yes
BiCGStab	Solving	Yes	Yes
BiCGStab(l)	Building	Yes	Yes
BiCGStab(l)	Solving	Yes	Yes
QMRCGStab	Building	Yes	Yes
QMRCGStab	Solving	Yes	Yes
GMRES	Building	Yes	Yes

continues on next page

Table 2.3 – continued from previous page

Solver	Functionality	Host	HIP
GMRES	Solving	Yes	Yes
FGMRES	Building	Yes	Yes
FGMRES	Solving	Yes	Yes
Chebyshev	Building	Yes	Yes
Chebyshev	Solving	Yes	Yes
Mixed-Precision	Building	Yes	Yes
Mixed-Precision	Solving	Yes	Yes
Fixed-Point Iteration	Building	Yes	Yes
Fixed-Point Iteration	Solving	Yes	Yes
AMG (Plain Aggregation)	Building	Yes	No
AMG (Plain Aggregation)	Solving	Yes	Yes
AMG (Smoothed Aggregation)	Building	Yes	No
AMG (Smoothed Aggregation)	Solving	Yes	Yes
AMG (Ruge Stueben)	Building	Yes	No
AMG (Ruge Stueben)	Solving	Yes	Yes
AMG (Pairwise Aggregation)	Building	Yes	No
AMG (Pairwise Aggregation)	Solving	Yes	Yes
LU	Building	Yes	No
LU	Solving	Yes	No
QR	Building	Yes	No
QR	Solving	Yes	No
Inversion	Building	Yes	No
Inversion	Solving	Yes	Yes

Preconditioner	Functionality	Host	HIP
Jacobi	Building	Yes	Yes
Jacobi	Solving	Yes	Yes
BlockJacobi	Building	Yes	Yes
BlockJacobi	Solving	Yes	Yes
MultiColoredILU(0,1)	Building	Yes	Yes
MultiColoredILU(0,1)	Solving	Yes	Yes
MultiColoredILU(>0, >1)	Building	Yes	No
MultiColoredILU(>0, >1)	Solving	Yes	Yes
MultiElimination(I)LU	Building	Yes	No
MultiElimination(I)LU	Solving	Yes	Yes
ILU(0)	Building	Yes	Yes
ILU(0)	Solving	Yes	Yes
ILU(>0)	Building	Yes	No
ILU(>0)	Solving	Yes	No
ILUT	Building	Yes	No
ILUT	Solving	Yes	No
IC(0)	Building	Yes	No
IC(0)	Solving	Yes	No
FSAI	Building	Yes	No
FSAI	Solving	Yes	Yes
SPAI	Building	Yes	No
SPAI	Solving	Yes	Yes
Chebyshev	Building	Yes	No

continues on next page

Table 2.4 – continued from previous page

Preconditioner	Functionality	Host	HIP
Chebyshev	Solving	Yes	Yes
MultiColored(S)GS	Building	Yes	No
MultiColored(S)GS	Solving	Yes	Yes
(S)GS	Building	Yes	No
(S)GS	Solving	Yes	No
(R)AS	Building	Yes	Yes
(R)AS	Solving	Yes	Yes
BlockPreconditioner	Building	Yes	Yes
BlockPreconditioner	Solving	Yes	Yes
SaddlePoint	Building	Yes	No
SaddlePoint	Solving	Yes	Yes

2.5 Clients

rocALUTION clients host a variety of different examples as well as a unit test package. For detailed instructions on how to build rocALUTION with clients, see *Building from GitHub repository*.

2.5.1 Examples

The examples collection offers different possible set-ups of solvers and preconditioners. The following tables gives a short overview on the different examples:

Example	Description
amg	Algebraic Multigrid solver (smoothed aggregation scheme, GS smoothing)
as-precond	GMRES solver with Additive Schwarz preconditioning
async	Asynchronous rocALUTION object transfer
benchmark	Benchmarking important sparse functions
bicgstab	BiCGStab solver with multicolored Gauss-Seidel preconditioning
block-precond	GMRES solver with blockwise multicolored ILU preconditioning
cg-amg	CG solver with Algebraic Multigrid (smoothed aggregation scheme) preconditioning
cg	CG solver with Jacobi preconditioning
cmk	CG solver with ILU preconditioning using Cuthill McKee ordering
direct	Matrix inversion
fgmres	Flexible GMRES solver with multicolored Gauss-Seidel preconditioning
fixed-point	Fixed-Point iteration scheme using Jacobi relaxation
gmres	GMRES solver with multicolored Gauss-Seidel preconditioning
idr	Induced Dimension Reduction solver with Jacobi preconditioning
key	Sparse matrix unique key computation
me-preconditioner	CG solver with multi-elimination preconditioning
mixed-precision	Mixed-precision CG solver with multicolored ILU preconditioning
power-method	CG solver using Chebyshev preconditioning and power method for eigenvalue approximation
simple-spmv	Sparse Matrix Vector multiplication
sp-precond	BiCGStab solver with multicolored ILU preconditioning for saddle point problems
stencil	CG solver using stencil as operator
tns	CG solver with Truncated Neumann Series preconditioning
var-precond	FGMRES solver with variable preconditioning

Example (MPI)	Description
benchmark_mpi	Benchmarking important sparse functions
bicgstab_mpi	BiCGStab solver with multicolored Gauss-Seidel preconditioning
cg-amg_mpi	CG solver with Algebraic Multigrid (pairwise aggregation scheme) preconditioning
cg_mpi	CG solver with Jacobi preconditioning
fcg_mpi	Flexible CG solver with ILU preconditioning
fgmres_mpi	Flexible GMRES solver with SParse Approximate Inverse preconditioning
global-io_mpi	File I/O with CG solver and Factorized Sparse Approximate Inverse preconditioning
idr_mpi	IDR solver with Factorized Sparse Approximate Inverse preconditioning
qmrcgstab_mpi	QMRCGStab solver with ILU-T preconditioning

2.5.2 Unit Tests

Multiple unit tests are available to test for bad arguments, invalid parameters and solver and preconditioner functionality. The unit tests are based on google test. The tests cover a variety of different solver, preconditioning and matrix format combinations and can be performed on all available backends.

CHAPTER

THREE

API

This section provides a detailed list of the library API

3.1 Host Utility Functions

template<typename **DataType**>
void `rocalution::`**allocate_host**(int64_t size, *DataType* **ptr)

Allocate buffer on the host.

`allocate_host` allocates a buffer on the host.

Parameters

- **size** – **[in]** number of elements the buffer need to be allocated for
- **ptr** – **[out]** pointer to the position in memory where the buffer should be allocated, it is expected that `*ptr == NULL`

Template Parameters

DataType – can be char, int, unsigned int, float, double, std::complex<float> or std::complex<double>.

template<typename **DataType**>
void `rocalution::`**free_host**(*DataType* **ptr)

Free buffer on the host.

`free_host` deallocates a buffer on the host. `*ptr` will be set to NULL after successful deallocation.

Parameters

ptr – **[inout]** pointer to the position in memory where the buffer should be deallocated, it is expected that `*ptr != NULL`

Template Parameters

DataType – can be char, int, unsigned int, float, double, std::complex<float> or std::complex<double>.

template<typename **DataType**>
void `rocalution::`**set_to_zero_host**(int64_t size, *DataType* *ptr)

Set a host buffer to zero.

`set_to_zero_host` sets a host buffer to zero.

Parameters

- **size** – **[in]** number of elements
- **ptr** – **[inout]** pointer to the host buffer

Template Parameters

DataType – can be char, int, unsigned int, float, double, std::complex<float> or std::complex<double>.

double rocalution::**rocalution_time**(void)

Return current time in microseconds.

3.2 Backend Manager

int rocalution::**init_rocalution**(int rank = -1, int dev_per_node = 1)

Initialize rocALUTION platform.

init_rocalution defines a backend descriptor with information about the hardware and its specifications. All objects created after that contain a copy of this descriptor. If the specifications of the global descriptor are changed (e.g. set different number of threads) and new objects are created, only the new objects will use the new configurations.

For control, the library provides the following functions

- *set_device_rocalution()* is a unified function to select a specific device. If you have compiled the library with a backend and for this backend there are several available devices, you can use this function to select a particular one. This function has to be called before *init_rocalution()*.
- *set_omp_threads_rocalution()* sets the number of OpenMP threads. This function has to be called after *init_rocalution()*.

Example

```
#include <rocalution/rocalution.hpp>

using namespace rocalution;

int main(int argc, char* argv[])
{
    init_rocalution();

    // ...

    stop_rocalution();

    return 0;
}
```

Parameters

- **rank** – **[in]** specifies MPI rank when multi-node environment
- **dev_per_node** – **[in]** number of accelerator devices per node, when in multi-GPU environment

int rocalution::**stop_rocalution**(void)

Shutdown rocALUTION platform.

stop_rocalution shuts down the rocALUTION platform.

void rocalution::**set_device_rocalution**(int dev)

Set the accelerator device.

set_device_rocalution lets the user select the accelerator device that is supposed to be used for the computation.

Parameters
dev – **[in]** accelerator device ID for computation

void rocalution::**set_omp_threads_rocalution**(int nthreads)

Set number of OpenMP threads.

The number of threads which rocALUTION will use can be set with set_omp_threads_rocalution or by the global OpenMP environment variable (for Unix-like OS this is OMP_NUM_THREADS). During the initialization phase, the library provides affinity thread-core mapping:

- If the number of cores (including SMT cores) is greater or equal than two times the number of threads, then all the threads can occupy every second core ID (e.g. 0, 2, 4, ...). This is to avoid having two threads working on the same physical core, when SMT is enabled.
- If the number of threads is less or equal to the number of cores (including SMT), and the previous clause is false, then the threads can occupy every core ID (e.g. 0, 1, 2, 3, ...).
- If non of the above criteria is matched, then the default thread-core mapping is used (typically set by the OS).

Note: The thread-core mapping is available only for Unix-like OS.

Note: The user can disable the thread affinity by calling *set_omp_affinity_rocalution()*, before initializing the library (i.e. before *init_rocalution()*).

Parameters
nthreads – **[in]** number of OpenMP threads

void rocalution::**set_omp_affinity_rocalution**(bool affinity)

Enable/disable OpenMP host affinity.

set_omp_affinity_rocalution enables / disables OpenMP host affinity.

Parameters
affinity – **[in]** boolean to turn on/off OpenMP host affinity

void rocalution::**set_omp_threshold_rocalution**(int threshold)

Set OpenMP threshold size.

Whenever you want to work on a small problem, you might observe that the OpenMP host backend is (slightly) slower than using no OpenMP. This is mainly attributed to the small amount of work, which every thread should perform and the large overhead of forking/joining threads. This can be avoid by the OpenMP threshold size parameter in rocALUTION. The default threshold is set to 10000, which means that all matrices under (and equal) this size will use only one thread (disregarding the number of OpenMP threads set in the system). The threshold can be modified with set_omp_threshold_rocalution.

Parameters
threshold – **[in]** OpenMP threshold size

void rocalution::**info_rocalution**(void)

Print info about rocALUTION.

`info_rocalution` prints information about the rocALUTION platform

void rocalution::**info_rocalution**(const struct Rocalution_Backend_Descriptor &backend_descriptor)

Print info about specific rocALUTION backend descriptor.

`info_rocalution` prints information about the rocALUTION platform of the specific backend descriptor.

Parameters
backend_descriptor – **[in]** rocALUTION backend descriptor

void rocalution::**disable_accelerator_rocalution**(bool onoff = true)

Disable/Enable the accelerator.

If you want to disable the accelerator (without re-compiling the code), you need to call `disable_accelerator_rocalution` before *init_rocalution()*.

Parameters
onoff – **[in]** boolean to turn on/off the accelerator

void rocalution::**_rocalution_sync**(void)

Sync rocALUTION.

`_rocalution_sync` blocks the host until all active asynchronous transfers are completed (this is a global sync).

3.3 Base Rocalution

template<typename **ValueType**>

class **BaseRocalution** : public rocalution::RocalutionObj

Base class for all operators and vectors.

Template Parameters
ValueType – - can be int, float, double, std::complex<float> and std::complex<double>

Subclassed by *rocalution::Operator< ValueType >*, *rocalution::Vector< ValueType >*

Public Functions

virtual void **MoveToAccelerator**(void) = 0

Move the object to the accelerator backend.

virtual void **MoveToHost**(void) = 0

Move the object to the host backend.

virtual void **MoveToAcceleratorAsync**(void)

Move the object to the accelerator backend with async move.

virtual void **MoveToHostAsync**(void)

Move the object to the host backend with async move.

virtual void **Sync**(void)

Sync (the async move)

virtual void **CloneBackend**(const *BaseRocalution*<*ValueType*> &src)

Clone the Backend descriptor from another object.

With `CloneBackend`, the backend can be cloned without copying any data. This is especially useful, if several objects should reside on the same backend, but keep their original data.

Example

```
LocalVector<ValueType> vec;
LocalMatrix<ValueType> mat;

// Allocate and initialize vec and mat
// ...

LocalVector<ValueType> tmp;
// By cloning backend, tmp and vec will have the same backend as mat
tmp.CloneBackend(mat);
vec.CloneBackend(mat);

// The following matrix vector multiplication will be performed on the␣
↪backend
// selected in mat
mat.Apply(vec, &tmp);
```

Parameters

src – **[in]** Object, where the backend should be cloned from.

virtual void **Info**(void) const = 0

Print object information.

`Info` can print object information about any rocALUTION object. This information consists of object properties and backend data.

Example

```
mat.Info();
vec.Info();
```

virtual void **Clear**(void) = 0

Clear (free all data) the object.

3.4 Operator

template<typename **ValueType**>

class **Operator** : public rocalution::*BaseRocalution*<*ValueType*>

Operator class.

The *Operator* class defines the generic interface for applying an operator (e.g. matrix or stencil) from/to global and local vectors.

Template Parameters

ValueType – - can be int, float, double, std::complex<float> and std::complex<double>

Subclassed by *rocalution::GlobalMatrix< ValueType >*, *rocalution::LocalMatrix< ValueType >*, *rocalution::LocalStencil< ValueType >*

Public Functions

virtual int64_t **GetM**(void) const = 0

Return the number of rows in the matrix/stencil.

virtual int64_t **GetN**(void) const = 0

Return the number of columns in the matrix/stencil.

virtual int64_t **GetNnz**(void) const = 0

Return the number of non-zeros in the matrix/stencil.

virtual int64_t **GetLocalM**(void) const

Return the number of rows in the local matrix/stencil.

virtual int64_t **GetLocalN**(void) const

Return the number of columns in the local matrix/stencil.

virtual int64_t **GetLocalNnz**(void) const

Return the number of non-zeros in the local matrix/stencil.

virtual int64_t **GetGhostM**(void) const

Return the number of rows in the ghost matrix/stencil.

virtual int64_t **GetGhostN**(void) const

Return the number of columns in the ghost matrix/stencil.

virtual int64_t **GetGhostNnz**(void) const

Return the number of non-zeros in the ghost matrix/stencil.

virtual void **Transpose**(void)

Transpose the operator.

virtual void **Apply**(const *LocalVector*<*ValueType*> &in, *LocalVector*<*ValueType*> *out) const

Apply the operator, out = Operator(in), where in and out are local vectors.

virtual void **ApplyAdd**(const *LocalVector*<*ValueType*> &in, *ValueType* scalar, *LocalVector*<*ValueType*> *out) const

Apply and add the operator, out += scalar * Operator(in), where in and out are local vectors.

virtual void **Apply**(const *GlobalVector*<*ValueType*> &in, *GlobalVector*<*ValueType*> *out) const

Apply the operator, out = Operator(in), where in and out are global vectors.

virtual void **ApplyAdd**(const *GlobalVector*<*ValueType*> &in, *ValueType* scalar, *GlobalVector*<*ValueType*> *out) const

Apply and add the operator, out += scalar * Operator(in), where in and out are global vectors.

3.5 Vector

template<typename **ValueType**>

class **Vector** : public rocalution::*BaseRocalution*<*ValueType*>

Vector class.

The *Vector* class defines the generic interface for local and global vectors.

Template Parameters
ValueType – - can be int, float, double, std::complex<float> and std::complex<double>

Subclassed by *rocalution::LocalVector< int >*, *rocalution::GlobalVector< ValueType >*, *rocalution::LocalVector< ValueType >*

Unnamed Group

virtual void **CopyFrom**(const *LocalVector*<*ValueType*> &src)

Copy vector from another vector.

`CopyFrom` copies values from another vector.

Example

```
LocalVector<ValueType> vec1, vec2;

// Allocate and initialize vec1 and vec2
// ...

// Move vec1 to accelerator
// vec1.MoveToAccelerator();

// Now, vec1 is on the accelerator (if available)
// and vec2 is on the host

// Copy vec1 to vec2 (or vice versa) will move data between host and
// accelerator backend
vec1.CopyFrom(vec2);
```

Note: This function allows cross platform copying. One of the objects could be allocated on the accelerator backend.

Parameters
src – **[in]** *Vector*, where values should be copied from.

virtual void **CopyFrom**(const *GlobalVector*<*ValueType*> &src)

Copy vector from another vector.

`CopyFrom` copies values from another vector.

Example

```
LocalVector<ValueType> vec1, vec2;

// Allocate and initialize vec1 and vec2
// ...

// Move vec1 to accelerator
// vec1.MoveToAccelerator();

// Now, vec1 is on the accelerator (if available)
// and vec2 is on the host

// Copy vec1 to vec2 (or vice versa) will move data between host and
// accelerator backend
vec1.CopyFrom(vec2);
```

Note: This function allows cross platform copying. One of the objects could be allocated on the accelerator backend.

Parameters

src – **[in]** *Vector*, where values should be copied from.

Unnamed Group

virtual void **CloneFrom**(const *LocalVector*<*ValueType*> &src)

Clone the vector.

`CloneFrom` clones the entire vector, with data and backend descriptor from another *Vector*.

Example

```
LocalVector<ValueType> vec;

// Allocate and initialize vec (host or accelerator)
// ...

LocalVector<ValueType> tmp;

// By cloning vec, tmp will have identical values and will be on the same
// backend as vec
tmp.CloneFrom(vec);
```

Parameters

src – **[in]** *Vector* to clone from.

virtual void **CloneFrom**(const *GlobalVector*<*ValueType*> &src)

Clone the vector.

`CloneFrom` clones the entire vector, with data and backend descriptor from another *Vector*.

Example

```
LocalVector<ValueType> vec;

// Allocate and initialize vec (host or accelerator)
// ...

LocalVector<ValueType> tmp;

// By cloning vec, tmp will have identical values and will be on the same
// backend as vec
tmp.CloneFrom(vec);
```

Parameters
src – **[in]** *Vector* to clone from.

Public Functions

virtual int64_t **GetSize**(void) const = 0

Return the size of the vector.

virtual int64_t **GetLocalSize**(void) const

Return the size of the local vector.

virtual bool **Check**(void) const = 0

Perform a sanity check of the vector.

Checks, if the vector contains valid data, i.e. if the values are not infinity and not NaN (not a number).

Return values

- **true** – if the vector is ok (empty vector is also ok).
- **false** – if there is something wrong with the values.

virtual void **Clear**(void) = 0

Clear (free all data) the object.

virtual void **Zeros**(void) = 0

Set all values of the vector to 0.

virtual void **Ones**(void) = 0

Set all values of the vector to 1.

virtual void **SetValues**(*ValueType* val) = 0

Set all values of the vector to given argument.

virtual void **SetRandomUniform**(unsigned long long seed, *ValueType* a = static_cast<*ValueType*>(-1), *ValueType* b = static_cast<*ValueType*>(1)) = 0

Fill the vector with random values from interval [a,b].

virtual void **SetRandomNormal**(unsigned long long seed, *ValueType* mean = static_cast<*ValueType*>(0), *ValueType* var = static_cast<*ValueType*>(1)) = 0

Fill the vector with random values from normal distribution.

virtual void **ReadFileASCII**(const std::string &filename) = 0

Read vector from ASCII file.

Read a vector from ASCII file.

Example

```
LocalVector<ValueType> vec;
vec.ReadFileASCII("my_vector.dat");
```

Parameters
filename – **[in]** name of the file containing the ASCII data.

virtual void **WriteFileASCII**(const std::string &filename) const = 0

Write vector to ASCII file.

Write a vector to ASCII file.

Example

```
LocalVector<ValueType> vec;

// Allocate and fill vec
// ...

vec.WriteFileASCII("my_vector.dat");
```

Parameters
filename – **[in]** name of the file to write the ASCII data to.

virtual void **ReadFileBinary**(const std::string &filename) = 0

Read vector from binary file.

Read a vector from binary file. For details on the format, see *WriteFileBinary()*.

Example

```
LocalVector<ValueType> vec;
vec.ReadFileBinary("my_vector.bin");
```

Parameters
filename – **[in]** name of the file containing the data.

virtual void **WriteFileBinary**(const std::string &filename) const = 0

Write vector to binary file.

Write a vector to binary file.

The binary format contains a header, the rocALUTION version and the vector data as follows

```
// Header
out << "#rocALUTION binary vector file" << std::endl;

// rocALUTION version
out.write((char*)&version, sizeof(int));

// Vector data
out.write((char*)&size, sizeof(int));
out.write((char*)vec_val, size * sizeof(double));
```

Example

```
LocalVector<ValueType> vec;

// Allocate and fill vec
// ...

vec.WriteFileBinary("my_vector.bin");
```

Note: *Vector* values array is always stored in double precision (e.g. double or std::complex<double>).

Parameters

filename – **[in]** name of the file to write the data to.

virtual void **CopyFromAsync**(const *LocalVector*<*ValueType*> &src)

Async copy from another local vector.

virtual void **CopyFromFloat**(const *LocalVector*<float> &src)

Copy values from another local float vector.

virtual void **CopyFromDouble**(const *LocalVector*<double> &src)

Copy values from another local double vector.

virtual void **CopyFrom**(const *LocalVector*<*ValueType*> &src, int64_t src_offset, int64_t dst_offset, int64_t size)

Copy vector from another vector with offsets and size.

`CopyFrom` copies values with specific source and destination offsets and sizes from another vector.

Note: This function allows cross platform copying. One of the objects could be allocated on the accelerator backend.

Parameters

- **src** – **[in]** *Vector*, where values should be copied from.
- **src_offset** – **[in]** source offset.
- **dst_offset** – **[in]** destination offset.
- **size** – **[in]** number of entries to be copied.

virtual void **AddScale**(const *LocalVector*<*ValueType*> &x, *ValueType* alpha)

Perform vector update of type this = this + alpha * x.

virtual void **AddScale**(const *GlobalVector*<*ValueType*> &x, *ValueType* alpha)

Perform vector update of type this = this + alpha * x.

virtual void **ScaleAdd**(*ValueType* alpha, const *LocalVector*<*ValueType*> &x)

Perform vector update of type this = alpha * this + x.

virtual void **ScaleAdd**(*ValueType* alpha, const *GlobalVector*<*ValueType*> &x)

Perform vector update of type this = alpha * this + x.

virtual void **ScaleAddScale**(*ValueType* alpha, const *LocalVector*<*ValueType*> &x, *ValueType* beta)

Perform vector update of type this = alpha * this + x * beta.

virtual void **ScaleAddScale**(*ValueType* alpha, const *GlobalVector*<*ValueType*> &x, *ValueType* beta)

Perform vector update of type this = alpha * this + x * beta.

virtual void **ScaleAddScale**(*ValueType* alpha, const *LocalVector*<*ValueType*> &x, *ValueType* beta, int64_t src_offset, int64_t dst_offset, int64_t size)

Perform vector update of type this = alpha * this + x * beta with offsets.

virtual void **ScaleAddScale**(*ValueType* alpha, const *GlobalVector*<*ValueType*> &x, *ValueType* beta, int64_t src_offset, int64_t dst_offset, int64_t size)

Perform vector update of type this = alpha * this + x * beta with offsets.

virtual void **ScaleAdd2**(*ValueType* alpha, const *LocalVector*<*ValueType*> &x, *ValueType* beta, const *LocalVector*<*ValueType*> &y, *ValueType* gamma)

Perform vector update of type this = alpha * this + x * beta + y * gamma.

virtual void **ScaleAdd2**(*ValueType* alpha, const *GlobalVector*<*ValueType*> &x, *ValueType* beta, const *GlobalVector*<*ValueType*> &y, *ValueType* gamma)

Perform vector update of type this = alpha * this + x * beta + y * gamma.

virtual void **Scale**(*ValueType* alpha) = 0

Perform vector scaling this = alpha * this.

virtual *ValueType* **Dot**(const *LocalVector*<*ValueType*> &x) const

Compute dot (scalar) product, return this^T y.

virtual *ValueType* **Dot**(const *GlobalVector*<*ValueType*> &x) const

Compute dot (scalar) product, return this^T y.

virtual *ValueType* **DotNonConj**(const *LocalVector*<*ValueType*> &x) const

Compute non-conjugate dot (scalar) product, return this^T y.

virtual *ValueType* **DotNonConj**(const *GlobalVector*<*ValueType*> &x) const

Compute non-conjugate dot (scalar) product, return this^T y.

virtual *ValueType* **Norm**(void) const = 0

Compute L_2 norm of the vector, return = srqt(this^T this)

virtual *ValueType* **Reduce**(void) const = 0

Reduce the vector.

virtual *ValueType* **InclusiveSum**(void) = 0

Compute Inclusive sum.

virtual *ValueType* **ExclusiveSum**(void) = 0

Compute exclusive sum.

virtual *ValueType* **Asum**(void) const = 0

Compute the sum of absolute values of the vector, return = sum(|this|)

virtual int64_t **Amax**(*ValueType* &value) const = 0

Compute the absolute max of the vector, return = index(max(|this|))

virtual void **PointWiseMult**(const *LocalVector*<*ValueType*> &x)

Perform point-wise multiplication (element-wise) of this = this * x.

virtual void **PointWiseMult**(const *GlobalVector*<*ValueType*> &x)

Perform point-wise multiplication (element-wise) of this = this * x.

virtual void **PointWiseMult**(const *LocalVector*<*ValueType*> &x, const *LocalVector*<*ValueType*> &y)

Perform point-wise multiplication (element-wise) of this = x * y.

virtual void **PointWiseMult**(const *GlobalVector*<*ValueType*> &x, const *GlobalVector*<*ValueType*> &y)

Perform point-wise multiplication (element-wise) of this = x * y.

virtual void **Power**(double power) = 0

Perform power operation to a vector.

3.6 Local Matrix

template<typename **ValueType**>

class **LocalMatrix** : public rocalution::*Operator*<*ValueType*>

LocalMatrix class.

A *LocalMatrix* is called local, because it will always stay on a single system. The system can contain several CPUs via UMA or NUMA memory system or it can contain an accelerator.

A number of matrix formats are supported. These are CSR, BCSR, MCSR, COO, DIA, ELL, HYB, and DENSE.

Note: For CSR type matrices, the column indices must be sorted in increasing order. For COO matrices, the row indices must be sorted in increasing order. The function `Check` can be used to check whether a matrix contains valid data. For CSR and COO matrices, the function `Sort` can be used to sort the row or column indices respectively.

Template Parameters

ValueType – - can be int, float, double, std::complex<float> and std::complex<double>

Unnamed Group

void **AllocateCSR**(const std::string &name, int64_t nnz, int64_t nrow, int64_t ncol)

Allocate a local matrix with name and sizes.

The local matrix allocation functions require a name of the object (this is only for information purposes) and corresponding number of non-zero elements, number of rows and number of columns. Furthermore, depending on the matrix format, additional parameters are required.

Example

```
LocalMatrix<ValueType> mat;

mat.AllocateCSR("my CSR matrix", 456, 100, 100);
mat.Clear();

mat.AllocateCOO("my COO matrix", 200, 100, 100);
mat.Clear();
```

void **AllocateBCSR**(const std::string &name, int64_t nnzb, int64_t nrowb, int64_t ncolb, int blockdim)

Allocate a local matrix with name and sizes.

The local matrix allocation functions require a name of the object (this is only for information purposes) and corresponding number of non-zero elements, number of rows and number of columns. Furthermore, depending on the matrix format, additional parameters are required.

Example

```
LocalMatrix<ValueType> mat;

mat.AllocateCSR("my CSR matrix", 456, 100, 100);
mat.Clear();

mat.AllocateCOO("my COO matrix", 200, 100, 100);
mat.Clear();
```

void **AllocateMCSR**(const std::string &name, int64_t nnz, int64_t nrow, int64_t ncol)

Allocate a local matrix with name and sizes.

The local matrix allocation functions require a name of the object (this is only for information purposes) and corresponding number of non-zero elements, number of rows and number of columns. Furthermore, depending on the matrix format, additional parameters are required.

Example

```
LocalMatrix<ValueType> mat;

mat.AllocateCSR("my CSR matrix", 456, 100, 100);
mat.Clear();

mat.AllocateCOO("my COO matrix", 200, 100, 100);
mat.Clear();
```

void **AllocateCOO**(const std::string &name, int64_t nnz, int64_t nrow, int64_t ncol)

Allocate a local matrix with name and sizes.

The local matrix allocation functions require a name of the object (this is only for information purposes) and corresponding number of non-zero elements, number of rows and number of columns. Furthermore, depending on the matrix format, additional parameters are required.

Example

```
LocalMatrix<ValueType> mat;

mat.AllocateCSR("my CSR matrix", 456, 100, 100);
mat.Clear();

mat.AllocateCOO("my COO matrix", 200, 100, 100);
mat.Clear();
```

void **AllocateDIA**(const std::string &name, int64_t nnz, int64_t nrow, int64_t ncol, int ndiag)

Allocate a local matrix with name and sizes.

The local matrix allocation functions require a name of the object (this is only for information purposes) and corresponding number of non-zero elements, number of rows and number of columns. Furthermore, depending on the matrix format, additional parameters are required.

Example

```
LocalMatrix<ValueType> mat;

mat.AllocateCSR("my CSR matrix", 456, 100, 100);
mat.Clear();

mat.AllocateCOO("my COO matrix", 200, 100, 100);
mat.Clear();
```

void **AllocateELL**(const std::string &name, int64_t nnz, int64_t nrow, int64_t ncol, int max_row)

Allocate a local matrix with name and sizes.

The local matrix allocation functions require a name of the object (this is only for information purposes) and corresponding number of non-zero elements, number of rows and number of columns. Furthermore, depending on the matrix format, additional parameters are required.

Example

```
LocalMatrix<ValueType> mat;

mat.AllocateCSR("my CSR matrix", 456, 100, 100);
mat.Clear();

mat.AllocateCOO("my COO matrix", 200, 100, 100);
mat.Clear();
```

void **AllocateHYB**(const std::string &name, int64_t ell_nnz, int64_t coo_nnz, int ell_max_row, int64_t nrow, int64_t ncol)

Allocate a local matrix with name and sizes.

The local matrix allocation functions require a name of the object (this is only for information purposes) and corresponding number of non-zero elements, number of rows and number of columns. Furthermore, depending on the matrix format, additional parameters are required.

Example

```
LocalMatrix<ValueType> mat;

mat.AllocateCSR("my CSR matrix", 456, 100, 100);
mat.Clear();

mat.AllocateCOO("my COO matrix", 200, 100, 100);
mat.Clear();
```

void **AllocateDENSE**(const std::string &name, int64_t nrow, int64_t ncol)

Allocate a local matrix with name and sizes.

The local matrix allocation functions require a name of the object (this is only for information purposes) and corresponding number of non-zero elements, number of rows and number of columns. Furthermore, depending on the matrix format, additional parameters are required.

Example

```
LocalMatrix<ValueType> mat;

mat.AllocateCSR("my CSR matrix", 456, 100, 100);
mat.Clear();

mat.AllocateCOO("my COO matrix", 200, 100, 100);
mat.Clear();
```

Unnamed Group

void **SetDataPtrCOO**(int **row, int **col, *ValueType* **val, std::string name, int64_t nnz, int64_t nrow, int64_t ncol)

Initialize a *LocalMatrix* on the host with externally allocated data.

`SetDataPtr` functions have direct access to the raw data via pointers. Already allocated data can be set by passing their pointers.

Example

```
// Allocate a CSR matrix
int* csr_row_ptr   = new int[100 + 1];
int* csr_col_ind   = new int[345];
ValueType* csr_val = new ValueType[345];
```

(continues on next page)

(continued from previous page)

```
// Fill the CSR matrix
// ...

// rocALUTION local matrix object
LocalMatrix<ValueType> mat;

// Set the CSR matrix data, csr_row_ptr, csr_col and csr_val pointers become
// invalid
mat.SetDataPtrCSR(&csr_row_ptr, &csr_col, &csr_val, "my_matrix", 345, 100,␣
↪100);
```

Note: Setting data pointers will leave the original pointers empty (set to NULL).

void **SetDataPtrCSR**(PtrType **row_offset, int **col, *ValueType* **val, std::string name, int64_t nnz, int64_t nrow, int64_t ncol)

Initialize a *LocalMatrix* on the host with externally allocated data.

SetDataPtr functions have direct access to the raw data via pointers. Already allocated data can be set by passing their pointers.

Example

```
// Allocate a CSR matrix
int* csr_row_ptr   = new int[100 + 1];
int* csr_col_ind   = new int[345];
ValueType* csr_val = new ValueType[345];

// Fill the CSR matrix
// ...

// rocALUTION local matrix object
LocalMatrix<ValueType> mat;

// Set the CSR matrix data, csr_row_ptr, csr_col and csr_val pointers become
// invalid
mat.SetDataPtrCSR(&csr_row_ptr, &csr_col, &csr_val, "my_matrix", 345, 100,␣
↪100);
```

Note: Setting data pointers will leave the original pointers empty (set to NULL).

void **SetDataPtrBCSR**(int **row_offset, int **col, *ValueType* **val, std::string name, int64_t nnzb, int64_t nrowb, int64_t ncolb, int blockdim)

Initialize a *LocalMatrix* on the host with externally allocated data.

SetDataPtr functions have direct access to the raw data via pointers. Already allocated data can be set by passing their pointers.

Example

```
// Allocate a CSR matrix
int* csr_row_ptr   = new int[100 + 1];
int* csr_col_ind   = new int[345];
ValueType* csr_val = new ValueType[345];

// Fill the CSR matrix
// ...

// rocALUTION local matrix object
LocalMatrix<ValueType> mat;

// Set the CSR matrix data, csr_row_ptr, csr_col and csr_val pointers become
// invalid
mat.SetDataPtrCSR(&csr_row_ptr, &csr_col, &csr_val, "my_matrix", 345, 100,␣
↪100);
```

Note: Setting data pointers will leave the original pointers empty (set to NULL).

void **SetDataPtrMCSR**(int **row_offset, int **col, *ValueType* **val, std::string name, int64_t nnz, int64_t nrow, int64_t ncol)

Initialize a *LocalMatrix* on the host with externally allocated data.

SetDataPtr functions have direct access to the raw data via pointers. Already allocated data can be set by passing their pointers.

Example

```
// Allocate a CSR matrix
int* csr_row_ptr   = new int[100 + 1];
int* csr_col_ind   = new int[345];
ValueType* csr_val = new ValueType[345];

// Fill the CSR matrix
// ...

// rocALUTION local matrix object
LocalMatrix<ValueType> mat;

// Set the CSR matrix data, csr_row_ptr, csr_col and csr_val pointers become
// invalid
mat.SetDataPtrCSR(&csr_row_ptr, &csr_col, &csr_val, "my_matrix", 345, 100,␣
↪100);
```

Note: Setting data pointers will leave the original pointers empty (set to NULL).

void **SetDataPtrELL**(int **col, *ValueType* **val, std::string name, int64_t nnz, int64_t nrow, int64_t ncol, int max_row)

Initialize a *LocalMatrix* on the host with externally allocated data.

SetDataPtr functions have direct access to the raw data via pointers. Already allocated data can be set by passing their pointers.

Example

```
// Allocate a CSR matrix
int* csr_row_ptr   = new int[100 + 1];
int* csr_col_ind   = new int[345];
ValueType* csr_val = new ValueType[345];

// Fill the CSR matrix
// ...

// rocALUTION local matrix object
LocalMatrix<ValueType> mat;

// Set the CSR matrix data, csr_row_ptr, csr_col and csr_val pointers become
// invalid
mat.SetDataPtrCSR(&csr_row_ptr, &csr_col, &csr_val, "my_matrix", 345, 100,␣
↪100);
```

Note: Setting data pointers will leave the original pointers empty (set to NULL).

void **SetDataPtrDIA**(int **offset, *ValueType* **val, std::string name, int64_t nnz, int64_t nrow, int64_t ncol, int num_diag)

Initialize a *LocalMatrix* on the host with externally allocated data.

SetDataPtr functions have direct access to the raw data via pointers. Already allocated data can be set by passing their pointers.

Example

```
// Allocate a CSR matrix
int* csr_row_ptr   = new int[100 + 1];
int* csr_col_ind   = new int[345];
ValueType* csr_val = new ValueType[345];

// Fill the CSR matrix
// ...

// rocALUTION local matrix object
LocalMatrix<ValueType> mat;

// Set the CSR matrix data, csr_row_ptr, csr_col and csr_val pointers become
// invalid
mat.SetDataPtrCSR(&csr_row_ptr, &csr_col, &csr_val, "my_matrix", 345, 100,␣
↪100);
```

Note: Setting data pointers will leave the original pointers empty (set to NULL).

void **SetDataPtrDENSE**(*ValueType* **val, std::string name, int64_t nrow, int64_t ncol)

Initialize a *LocalMatrix* on the host with externally allocated data.

`SetDataPtr` functions have direct access to the raw data via pointers. Already allocated data can be set by passing their pointers.

Example

```
// Allocate a CSR matrix
int* csr_row_ptr   = new int[100 + 1];
int* csr_col_ind   = new int[345];
ValueType* csr_val = new ValueType[345];

// Fill the CSR matrix
// ...

// rocALUTION local matrix object
LocalMatrix<ValueType> mat;

// Set the CSR matrix data, csr_row_ptr, csr_col and csr_val pointers become
// invalid
mat.SetDataPtrCSR(&csr_row_ptr, &csr_col, &csr_val, "my_matrix", 345, 100,␣
↪100);
```

Note: Setting data pointers will leave the original pointers empty (set to `NULL`).

Unnamed Group

void **LeaveDataPtrCOO**(int **row, int **col, *ValueType* **val)

Leave a *LocalMatrix* to host pointers.

`LeaveDataPtr` functions have direct access to the raw data via pointers. A *LocalMatrix* object can leave its raw data to host pointers. This will leave the *LocalMatrix* empty.

Example

```
// rocALUTION CSR matrix object
LocalMatrix<ValueType> mat;

// Allocate the CSR matrix
mat.AllocateCSR("my_matrix", 345, 100, 100);

// Fill CSR matrix
// ...

int* csr_row_ptr   = NULL;
int* csr_col_ind   = NULL;
ValueType* csr_val = NULL;
```

(continues on next page)

(continued from previous page)

```
// Get (steal) the data from the matrix, this will leave the local matrix
// object empty
mat.LeaveDataPtrCSR(&csr_row_ptr, &csr_col_ind, &csr_val);
```

void **LeaveDataPtrCSR**(PtrType **row_offset, int **col, *ValueType* **val)

Leave a *LocalMatrix* to host pointers.

`LeaveDataPtr` functions have direct access to the raw data via pointers. A *LocalMatrix* object can leave its raw data to host pointers. This will leave the *LocalMatrix* empty.

Example

```
// rocALUTION CSR matrix object
LocalMatrix<ValueType> mat;

// Allocate the CSR matrix
mat.AllocateCSR("my_matrix", 345, 100, 100);

// Fill CSR matrix
// ...

int* csr_row_ptr   = NULL;
int* csr_col_ind   = NULL;
ValueType* csr_val = NULL;

// Get (steal) the data from the matrix, this will leave the local matrix
// object empty
mat.LeaveDataPtrCSR(&csr_row_ptr, &csr_col_ind, &csr_val);
```

void **LeaveDataPtrBCSR**(int **row_offset, int **col, *ValueType* **val, int &blockdim)

Leave a *LocalMatrix* to host pointers.

`LeaveDataPtr` functions have direct access to the raw data via pointers. A *LocalMatrix* object can leave its raw data to host pointers. This will leave the *LocalMatrix* empty.

Example

```
// rocALUTION CSR matrix object
LocalMatrix<ValueType> mat;

// Allocate the CSR matrix
mat.AllocateCSR("my_matrix", 345, 100, 100);

// Fill CSR matrix
// ...

int* csr_row_ptr   = NULL;
int* csr_col_ind   = NULL;
ValueType* csr_val = NULL;

// Get (steal) the data from the matrix, this will leave the local matrix
```

(continues on next page)

(continued from previous page)

```
// object empty
mat.LeaveDataPtrCSR(&csr_row_ptr, &csr_col_ind, &csr_val);
```

void **LeaveDataPtrMCSR**(int **row_offset, int **col, *ValueType* **val)

Leave a *LocalMatrix* to host pointers.

`LeaveDataPtr` functions have direct access to the raw data via pointers. A *LocalMatrix* object can leave its raw data to host pointers. This will leave the *LocalMatrix* empty.

Example

```
// rocALUTION CSR matrix object
LocalMatrix<ValueType> mat;

// Allocate the CSR matrix
mat.AllocateCSR("my_matrix", 345, 100, 100);

// Fill CSR matrix
// ...

int* csr_row_ptr   = NULL;
int* csr_col_ind   = NULL;
ValueType* csr_val = NULL;

// Get (steal) the data from the matrix, this will leave the local matrix
// object empty
mat.LeaveDataPtrCSR(&csr_row_ptr, &csr_col_ind, &csr_val);
```

void **LeaveDataPtrELL**(int **col, *ValueType* **val, int &max_row)

Leave a *LocalMatrix* to host pointers.

`LeaveDataPtr` functions have direct access to the raw data via pointers. A *LocalMatrix* object can leave its raw data to host pointers. This will leave the *LocalMatrix* empty.

Example

```
// rocALUTION CSR matrix object
LocalMatrix<ValueType> mat;

// Allocate the CSR matrix
mat.AllocateCSR("my_matrix", 345, 100, 100);

// Fill CSR matrix
// ...

int* csr_row_ptr   = NULL;
int* csr_col_ind   = NULL;
ValueType* csr_val = NULL;

// Get (steal) the data from the matrix, this will leave the local matrix
```

(continues on next page)

(continued from previous page)

```
// object empty
mat.LeaveDataPtrCSR(&csr_row_ptr, &csr_col_ind, &csr_val);
```

void **LeaveDataPtrDIA**(int **offset, *ValueType* **val, int &num_diag)

Leave a *LocalMatrix* to host pointers.

`LeaveDataPtr` functions have direct access to the raw data via pointers. A *LocalMatrix* object can leave its raw data to host pointers. This will leave the *LocalMatrix* empty.

Example

```
// rocALUTION CSR matrix object
LocalMatrix<ValueType> mat;

// Allocate the CSR matrix
mat.AllocateCSR("my_matrix", 345, 100, 100);

// Fill CSR matrix
// ...

int* csr_row_ptr   = NULL;
int* csr_col_ind   = NULL;
ValueType* csr_val = NULL;

// Get (steal) the data from the matrix, this will leave the local matrix
// object empty
mat.LeaveDataPtrCSR(&csr_row_ptr, &csr_col_ind, &csr_val);
```

void **LeaveDataPtrDENSE**(*ValueType* **val)

Leave a *LocalMatrix* to host pointers.

`LeaveDataPtr` functions have direct access to the raw data via pointers. A *LocalMatrix* object can leave its raw data to host pointers. This will leave the *LocalMatrix* empty.

Example

```
// rocALUTION CSR matrix object
LocalMatrix<ValueType> mat;

// Allocate the CSR matrix
mat.AllocateCSR("my_matrix", 345, 100, 100);

// Fill CSR matrix
// ...

int* csr_row_ptr   = NULL;
int* csr_col_ind   = NULL;
ValueType* csr_val = NULL;

// Get (steal) the data from the matrix, this will leave the local matrix
```

(continues on next page)

(continued from previous page)

```
// object empty
mat.LeaveDataPtrCSR(&csr_row_ptr, &csr_col_ind, &csr_val);
```

Public Functions

virtual void **Info**(void) const

Print object information.

`Info` can print object information about any rocALUTION object. This information consists of object properties and backend data.

Example

```
mat.Info();
vec.Info();
```

unsigned int **GetFormat**(void) const

Return the matrix format id (see matrix_formats.hpp)

int **GetBlockDimension**(void) const

Return the matrix block dimension.

virtual int64_t **GetM**(void) const

Return the number of rows in the matrix/stencil.

virtual int64_t **GetN**(void) const

Return the number of columns in the matrix/stencil.

virtual int64_t **GetNnz**(void) const

Return the number of non-zeros in the matrix/stencil.

bool **Check**(void) const

Perform a sanity check of the matrix.

Checks, if the matrix contains valid data, i.e. if the values are not infinity and not NaN (not a number) and if the structure of the matrix is correct (e.g. indices cannot be negative, CSR and COO matrices have to be sorted, etc.).

Return values

- **true** – if the matrix is ok (empty matrix is also ok).
- **false** – if there is something wrong with the structure or values.

virtual void **Clear**(void)

Clear (free all data) the object.

void **Zeros**(void)

Set all matrix values to zero.

void **Scale**(*ValueType* alpha)

Scale all values in the matrix.

void **ScaleDiagonal**(*ValueType* alpha)

Scale the diagonal entries of the matrix with alpha, all diagonal elements must exist.

void **ScaleOffDiagonal**(*ValueType* alpha)

Scale the off-diagonal entries of the matrix with alpha, all diagonal elements must exist.

void **AddScalar**(*ValueType* alpha)

Add a scalar to all matrix values.

void **AddScalarDiagonal**(*ValueType* alpha)

Add alpha to the diagonal entries of the matrix, all diagonal elements must exist.

void **AddScalarOffDiagonal**(*ValueType* alpha)

Add alpha to the off-diagonal entries of the matrix, all diagonal elements must exist.

void **ExtractSubMatrix**(int64_t row_offset, int64_t col_offset, int64_t row_size, int64_t col_size, *LocalMatrix*<*ValueType*> *mat) const

Extract a sub-matrix with row/col_offset and row/col_size.

void **ExtractSubMatrices**(int row_num_blocks, int col_num_blocks, const int *row_offset, const int *col_offset, *LocalMatrix*<*ValueType*> ***mat) const

Extract array of non-overlapping sub-matrices (row/col_num_blocks define the blocks for rows/columns; row/col_offset have sizes col/row_num_blocks+1, where [i+1]-[i] defines the i-th size of the sub-matrix)

void **ExtractDiagonal**(*LocalVector*<*ValueType*> *vec_diag) const

Extract the diagonal values of the matrix into a *LocalVector*.

void **ExtractInverseDiagonal**(*LocalVector*<*ValueType*> *vec_inv_diag) const

Extract the inverse (reciprocal) diagonal values of the matrix into a *LocalVector*.

void **ExtractU**(*LocalMatrix*<*ValueType*> *U, bool diag) const

Extract the upper triangular matrix.

void **ExtractL**(*LocalMatrix*<*ValueType*> *L, bool diag) const

Extract the lower triangular matrix.

void **Permute**(const *LocalVector*<int> &permutation)

Perform (forward) permutation of the matrix.

void **PermuteBackward**(const *LocalVector*<int> &permutation)

Perform (backward) permutation of the matrix.

void **CMK**(*LocalVector*<int> *permutation) const

Create permutation vector for CMK reordering of the matrix.

The Cuthill-McKee ordering minimize the bandwidth of a given sparse matrix.

Example

```
LocalVector<int> cmk;

mat.CMK(&cmk);
mat.Permute(cmk);
```

Parameters

permutation – **[out]** permutation vector for CMK reordering

void **RCMK**(*LocalVector*<int> *permutation) const

Create permutation vector for reverse CMK reordering of the matrix.

The Reverse Cuthill-McKee ordering minimize the bandwidth of a given sparse matrix.

Example

```
LocalVector<int> rcmk;

mat.RCMK(&rcmk);
mat.Permute(rcmk);
```

Parameters

permutation – **[out]** permutation vector for reverse CMK reordering

void **ConnectivityOrder**(*LocalVector*<int> *permutation) const

Create permutation vector for connectivity reordering of the matrix.

Connectivity ordering returns a permutation, that sorts the matrix by non-zero entries per row.

Example

```
LocalVector<int> conn;

mat.ConnectivityOrder(&conn);
mat.Permute(conn);
```

Parameters

permutation – **[out]** permutation vector for connectivity reordering

void **MultiColoring**(int &num_colors, int **size_colors, *LocalVector*<int> *permutation) const

Perform multi-coloring decomposition of the matrix.

The Multi-Coloring algorithm builds a permutation (coloring of the matrix) in a way such that no two adjacent nodes in the sparse matrix have the same color.

Example

```
LocalVector<int> mc;
int num_colors;
int* block_colors = NULL;

mat.MultiColoring(num_colors, &block_colors, &mc);
mat.Permute(mc);
```

Parameters

- **num_colors** – **[out]** number of colors
- **size_colors** – **[out]** pointer to array that holds the number of nodes for each color
- **permutation** – **[out]** permutation vector for multi-coloring reordering

void **MaximalIndependentSet**(int &size, *LocalVector*<int> *permutation) const

Perform maximal independent set decomposition of the matrix.

The Maximal Independent Set algorithm finds a set with maximal size, that contains elements that do not depend on other elements in this set.

Example

```
LocalVector<int> mis;
int size;

mat.MaximalIndependentSet(size, &mis);
mat.Permute(mis);
```

Parameters

- **size** – **[out]** number of independent sets
- **permutation** – **[out]** permutation vector for maximal independent set reordering

void **ZeroBlockPermutation**(int &size, *LocalVector*<int> *permutation) const

Return a permutation for saddle-point problems (zero diagonal entries)

For Saddle-Point problems, (i.e. matrices with zero diagonal entries), the Zero Block Permutation maps all zero-diagonal elements to the last block of the matrix.

Example

```
LocalVector<int> zbp;
int size;

mat.ZeroBlockPermutation(size, &zbp);
mat.Permute(zbp);
```

Parameters

- **size** – **[out]**
- **permutation** – **[out]** permutation vector for zero block permutation

void **ILU0Factorize**(void)

Perform ILU(0) factorization.

void **LUFactorize**(void)

Perform *LU* factorization.

void **ILUTFactorize**(double t, int maxrow)

Perform ILU(t,m) factorization based on threshold and maximum number of elements per row.

void **ILUpFactorize**(int p, bool level = true)

Perform ILU(p) factorization based on power.

void **LUAnalyse**(void)

Analyse the structure (level-scheduling)

void **LUAnalyseClear**(void)

Delete the analysed data (see LUAnalyse)

void **LUSolve**(const *LocalVector*<*ValueType*> &in, *LocalVector*<*ValueType*> *out) const

Solve *LU* out = in; if level-scheduling algorithm is provided then the graph traversing is performed in parallel.

void **ICFactorize**(*LocalVector*<*ValueType*> *inv_diag)

Perform IC(0) factorization.

void **LLAnalyse**(void)

Analyse the structure (level-scheduling)

void **LLAnalyseClear**(void)

Delete the analysed data (see LLAnalyse)

void **LLSolve**(const *LocalVector*<*ValueType*> &in, *LocalVector*<*ValueType*> *out) const

Solve LL^T out = in; if level-scheduling algorithm is provided then the graph traversing is performed in parallel.

void **LLSolve**(const *LocalVector*<*ValueType*> &in, const *LocalVector*<*ValueType*> &inv_diag, *LocalVector*<*ValueType*> *out) const

Solve LL^T out = in; if level-scheduling algorithm is provided then the graph traversing is performed in parallel.

void **LAnalyse**(bool diag_unit = false)

Analyse the structure (level-scheduling) L-part.

- diag_unit == true the diag is 1;
- diag_unit == false the diag is 0;

void **LAnalyseClear**(void)

Delete the analysed data (see LAnalyse) L-part.

void **LSolve**(const *LocalVector*<*ValueType*> &in, *LocalVector*<*ValueType*> *out) const

Solve L out = in; if level-scheduling algorithm is provided then the graph traversing is performed in parallel.

void **UAnalyse**(bool diag_unit = false)

Analyse the structure (level-scheduling) U-part;.

- diag_unit == true the diag is 1;
- diag_unit == false the diag is 0;

void **UAnalyseClear**(void)

Delete the analysed data (see UAnalyse) U-part.

void **USolve**(const *LocalVector*<*ValueType*> &in, *LocalVector*<*ValueType*> *out) const

Solve U out = in; if level-scheduling algorithm is provided then the graph traversing is performed in parallel.

void **Householder**(int idx, *ValueType* &beta, *LocalVector*<*ValueType*> *vec) const

Compute Householder vector.

void **QRDecompose**(void)

QR Decomposition.

void **QRSolve**(const *LocalVector*<*ValueType*> &in, *LocalVector*<*ValueType*> *out) const

Solve *QR* out = in.

void **Invert**(void)

Matrix inversion using *QR* decomposition.

void **ReadFileMTX**(const std::string &filename)

Read matrix from MTX (Matrix Market Format) file.

Read a matrix from Matrix Market Format file.

Example

```
LocalMatrix<ValueType> mat;
mat.ReadFileMTX("my_matrix.mtx");
```

Parameters
filename – **[in]** name of the file containing the MTX data.

void **WriteFileMTX**(const std::string &filename) const

Write matrix to MTX (Matrix Market Format) file.

Write a matrix to Matrix Market Format file.

Example

```
LocalMatrix<ValueType> mat;

// Allocate and fill mat
// ...

mat.WriteFileMTX("my_matrix.mtx");
```

Parameters
filename – **[in]** name of the file to write the MTX data to.

void **ReadFileCSR**(const std::string &filename)

Read matrix from CSR (rocALUTION binary format) file.

Read a CSR matrix from binary file. For details on the format, see *WriteFileCSR()*.

Example

```
LocalMatrix<ValueType> mat;
mat.ReadFileCSR("my_matrix.csr");
```

Parameters
filename – **[in]** name of the file containing the data.

void **WriteFileCSR**(const std::string &filename) const

Write CSR matrix to binary file.

Write a CSR matrix to binary file.

The binary format contains a header, the rocALUTION version and the matrix data as follows

```
// Header
out << "#rocALUTION binary csr file" << std::endl;

// rocALUTION version
out.write((char*)&version, sizeof(int));

// CSR matrix data
out.write((char*)&m, sizeof(int));
out.write((char*)&n, sizeof(int));
out.write((char*)&nnz, sizeof(int64_t));
out.write((char*)csr_row_ptr, (m + 1) * sizeof(int));
out.write((char*)csr_col_ind, nnz * sizeof(int));
out.write((char*)csr_val, nnz * sizeof(double));
```

Example

```
LocalMatrix<ValueType> mat;

// Allocate and fill mat
// ...

mat.WriteFileCSR("my_matrix.csr");
```

Note: *Vector* values array is always stored in double precision (e.g. double or std::complex<double>).

Parameters

filename – **[in]** name of the file to write the data to.

virtual void **MoveToAccelerator**(void)

Move the object to the accelerator backend.

virtual void **MoveToAcceleratorAsync**(void)

Move the object to the accelerator backend with async move.

virtual void **MoveToHost**(void)

Move the object to the host backend.

virtual void **MoveToHostAsync**(void)

Move the object to the host backend with async move.

virtual void **Sync**(void)

Sync (the async move)

void **CopyFrom**(const *LocalMatrix*<*ValueType*> &src)

Copy matrix from another *LocalMatrix*.

`CopyFrom` copies values and structure from another local matrix. Source and destination matrix should be in the same format.

Example

```
LocalMatrix<ValueType> mat1, mat2;

// Allocate and initialize mat1 and mat2
// ...

// Move mat1 to accelerator
// mat1.MoveToAccelerator();

// Now, mat1 is on the accelerator (if available)
// and mat2 is on the host

// Copy mat1 to mat2 (or vice versa) will move data between host and
// accelerator backend
mat1.CopyFrom(mat2);
```

Note: This function allows cross platform copying. One of the objects could be allocated on the accelerator backend.

Parameters

src – **[in]** Local matrix where values and structure should be copied from.

void **CopyFromAsync**(const *LocalMatrix*<*ValueType*> &src)

Async copy matrix (values and structure) from another *LocalMatrix*.

void **CloneFrom**(const *LocalMatrix*<*ValueType*> &src)

Clone the matrix.

`CloneFrom` clones the entire matrix, including values, structure and backend descriptor from another *LocalMatrix*.

Example

```
LocalMatrix<ValueType> mat;

// Allocate and initialize mat (host or accelerator)
// ...

LocalMatrix<ValueType> tmp;

// By cloning mat, tmp will have identical values and structure and will be␣
↪on
// the same backend as mat
tmp.CloneFrom(mat);
```

Parameters

src – **[in]** *LocalMatrix* to clone from.

void **UpdateValuesCSR**(*ValueType* *val)

Update CSR matrix entries only, structure will remain the same.

void **CopyFromCSR**(const PtrType *row_offsets, const int *col, const *ValueType* *val)

Copy (import) CSR matrix described in three arrays (offsets, columns, values). The object data has to be allocated (call AllocateCSR first)

void **CopyToCSR**(PtrType *row_offsets, int *col, *ValueType* *val) const

Copy (export) CSR matrix described in three arrays (offsets, columns, values). The output arrays have to be allocated.

void **CopyFromCOO**(const int *row, const int *col, const *ValueType* *val)

Copy (import) COO matrix described in three arrays (rows, columns, values). The object data has to be allocated (call AllocateCOO first)

void **CopyToCOO**(int *row, int *col, *ValueType* *val) const

Copy (export) COO matrix described in three arrays (rows, columns, values). The output arrays have to be allocated.

void **CopyFromHostCSR**(const PtrType *row_offset, const int *col, const *ValueType* *val, const std::string &name, int64_t nnz, int64_t nrow, int64_t ncol)

Allocates and copies (imports) a host CSR matrix.

If the CSR matrix data pointers are only accessible as constant, the user can create a *LocalMatrix* object and pass const CSR host pointers. The *LocalMatrix* will then be allocated and the data will be copied to the corresponding backend, where the original object was located at.

Parameters

- **row_offset** – **[in]** CSR matrix row offset pointers.
- **col** – **[in]** CSR matrix column indices.
- **val** – **[in]** CSR matrix values array.
- **name** – **[in]** Matrix object name.
- **nnz** – **[in]** Number of non-zero elements.
- **nrow** – **[in]** Number of rows.
- **ncol** – **[in]** Number of columns.

void **CreateFromMap**(const *LocalVector*<int> &map, int64_t n, int64_t m)

Create a restriction matrix operator based on an int vector map.

void **CreateFromMap**(const *LocalVector*<int> &map, int64_t n, int64_t m, *LocalMatrix*<*ValueType*> *pro)

Create a restriction and prolongation matrix operator based on an int vector map.

void **ConvertToCSR**(void)

Convert the matrix to CSR structure.

void **ConvertToMCSR**(void)

Convert the matrix to MCSR structure.

void **ConvertToBCSR**(int blockdim)

Convert the matrix to BCSR structure.

void **ConvertToCOO**(void)

Convert the matrix to COO structure.

void **ConvertToELL**(void)

Convert the matrix to ELL structure.

void **ConvertToDIA**(void)

Convert the matrix to DIA structure.

void **ConvertToHYB**(void)

Convert the matrix to HYB structure.

void **ConvertToDENSE**(void)

Convert the matrix to DENSE structure.

void **ConvertTo**(unsigned int matrix_format, int blockdim = 1)

Convert the matrix to specified matrix ID format.

virtual void **Apply**(const *LocalVector*<*ValueType*> &in, *LocalVector*<*ValueType*> *out) const

Apply the operator, out = Operator(in), where in and out are local vectors.

virtual void **ApplyAdd**(const *LocalVector*<*ValueType*> &in, *ValueType* scalar, *LocalVector*<*ValueType*> *out) const

Apply and add the operator, out += scalar * Operator(in), where in and out are local vectors.

void **SymbolicPower**(int p)

Perform symbolic computation (structure only) of $|this|^p$.

void **MatrixAdd**(const *LocalMatrix*<*ValueType*> &mat, *ValueType* alpha = static_cast<*ValueType*>(1), *ValueType* beta = static_cast<*ValueType*>(1), bool structure = false)

Perform matrix addition, this = alpha*this + beta*mat;.

- if structure==false the sparsity pattern of the matrix is not changed;
- if structure==true a new sparsity pattern is computed

void **MatrixMult**(const *LocalMatrix*<*ValueType*> &A, const *LocalMatrix*<*ValueType*> &B)

Multiply two matrices, this = A * B.

void **DiagonalMatrixMult**(const *LocalVector*<*ValueType*> &diag)

Multiply the matrix with diagonal matrix (stored in *LocalVector*), as *DiagonalMatrixMultR()*

void **DiagonalMatrixMultL**(const *LocalVector*<*ValueType*> &diag)

Multiply the matrix with diagonal matrix (stored in *LocalVector*), this=diag*this.

void **DiagonalMatrixMultR**(const *LocalVector*<*ValueType*> &diag)

Multiply the matrix with diagonal matrix (stored in *LocalVector*), this=this*diag.

void **TripleMatrixProduct**(const *LocalMatrix*<*ValueType*> &R, const *LocalMatrix*<*ValueType*> &A, const *LocalMatrix*<*ValueType*> &P)

Triple matrix product C=RAP.

void **Gershgorin**(*ValueType* &lambda_min, *ValueType* &lambda_max) const

Compute the spectrum approximation with Gershgorin circles theorem.

void **Compress**(double drop_off)

Delete all entries in the matrix which abs(a_ij) <= drop_off; the diagonal elements are never deleted.

virtual void **Transpose**(void)

Transpose the matrix.

void **Transpose**(*LocalMatrix*<*ValueType*> *T) const

Transpose the matrix.

void **Sort**(void)

Sort the matrix indices.

Sorts the matrix by indices.

- For CSR matrices, column values are sorted.
- For COO matrices, row indices are sorted.

void **Key**(long int &row_key, long int &col_key, long int &val_key) const

Compute a unique hash key for the matrix arrays.

Typically, it is hard to compare if two matrices have the same structure (and values). To do so, rocALUTION provides a keying function, that generates three keys, for the row index, column index and values array.

Parameters

- **row_key** – **[out]** row index array key
- **col_key** – **[out]** column index array key
- **val_key** – **[out]** values array key

void **ReplaceColumnVector**(int idx, const *LocalVector*<*ValueType*> &vec)

Replace a column vector of a matrix.

void **ReplaceRowVector**(int idx, const *LocalVector*<*ValueType*> &vec)

Replace a row vector of a matrix.

void **ExtractColumnVector**(int idx, *LocalVector*<*ValueType*> *vec) const

Extract values from a column of a matrix to a vector.

void **ExtractRowVector**(int idx, *LocalVector*<*ValueType*> *vec) const

Extract values from a row of a matrix to a vector.

void **AMGConnect**(*ValueType* eps, *LocalVector*<int> *connections) const

Strong couplings for aggregation-based AMG.

void **AMGAggregate**(const *LocalVector*<int> &connections, *LocalVector*<int> *aggregates) const

Plain aggregation - Modification of a greedy aggregation scheme from Vanek (1996)

void **AMGPMISAggregate**(const *LocalVector*<int> &connections, *LocalVector*<int> *aggregates) const

Parallel aggregation - Parallel maximal independent set aggregation scheme from Bell, Dalton, & Olsen (2012)

void **AMGSmoothedAggregation**(*ValueType* relax, const *LocalVector*<int> &aggregates, const *LocalVector*<int> &connections, *LocalMatrix*<*ValueType*> *prolong, int lumping_strat = 0) const

Interpolation scheme based on smoothed aggregation from Vanek (1996)

void **AMGAggregation**(const *LocalVector*<int> &aggregates, *LocalMatrix*<*ValueType*> *prolong) const

Aggregation-based interpolation scheme.

void **RSCoarsening**(float eps, *LocalVector*<int> *CFmap, *LocalVector*<bool> *S) const

Ruge Stueben coarsening.

void **RSPMISCoarsening**(float eps, *LocalVector*<int> *CFmap, *LocalVector*<bool> *S) const

Parallel maximal independent set coarsening for RS AMG.

void **RSDirectInterpolation**(const *LocalVector*<int> &CFmap, const *LocalVector*<bool> &S, *LocalMatrix*<*ValueType*> *prolong) const

Ruge Stueben Direct Interpolation.

void **RSExtPIInterpolation**(const *LocalVector*<int> &CFmap, const *LocalVector*<bool> &S, bool FF1, *LocalMatrix*<*ValueType*> *prolong) const

Ruge Stueben Ext+i Interpolation.

void **FSAI**(int power, const *LocalMatrix*<*ValueType*> *pattern)

Factorized Sparse Approximate Inverse assembly for given system matrix power pattern or external sparsity pattern.

void **SPAI**(void)

SParse Approximate Inverse assembly for given system matrix pattern.

void **InitialPairwiseAggregation**(*ValueType* beta, int &nc, *LocalVector*<int> *G, int &Gsize, int **rG, int &rGsize, int ordering) const

Initial Pairwise Aggregation scheme.

void **InitialPairwiseAggregation**(const *LocalMatrix*<*ValueType*> &mat, *ValueType* beta, int &nc, *LocalVector*<int> *G, int &Gsize, int **rG, int &rGsize, int ordering) const

Initial Pairwise Aggregation scheme for split matrices.

void **FurtherPairwiseAggregation**(*ValueType* beta, int &nc, *LocalVector*<int> *G, int &Gsize, int **rG, int &rGsize, int ordering) const

Further Pairwise Aggregation scheme.

void **FurtherPairwiseAggregation**(const *LocalMatrix*<*ValueType*> &mat, *ValueType* beta, int &nc, *LocalVector*<int> *G, int &Gsize, int **rG, int &rGsize, int ordering) const

Further Pairwise Aggregation scheme for split matrices.

void **CoarsenOperator**(*LocalMatrix*<*ValueType*> *Ac, int nrow, int ncol, const *LocalVector*<int> &G, int Gsize, const int *rG, int rGsize) const

Build coarse operator for pairwise aggregation scheme.

3.7 Local Stencil

template<typename **ValueType**>

class **LocalStencil** : public rocalution::*Operator*<*ValueType*>

LocalStencil class.

A *LocalStencil* is called local, because it will always stay on a single system. The system can contain several CPUs via UMA or NUMA memory system or it can contain an accelerator.

Template Parameters

ValueType – - can be int, float, double, std::complex<float> and std::complex<double>

Public Functions

LocalStencil(unsigned int type)

Initialize a local stencil with a type.

virtual void **Info**() const

Print object information.

`Info` can print object information about any rocALUTION object. This information consists of object properties and backend data.

Example

```
mat.Info();
vec.Info();
```

int64_t **GetNDim**(void) const

Return the dimension of the stencil.

virtual int64_t **GetM**(void) const

Return the number of rows in the matrix/stencil.

virtual int64_t **GetN**(void) const

Return the number of columns in the matrix/stencil.

virtual int64_t **GetNnz**(void) const

Return the number of non-zeros in the matrix/stencil.

void **SetGrid**(int size)

Set the stencil grid size.

virtual void **Clear**()

Clear (free all data) the object.

virtual void **Apply**(const *LocalVector*<*ValueType*> &in, *LocalVector*<*ValueType*> *out) const

Apply the operator, out = Operator(in), where in and out are local vectors.

virtual void **ApplyAdd**(const *LocalVector*<*ValueType*> &in, *ValueType* scalar, *LocalVector*<*ValueType*> *out) const

Apply and add the operator, out += scalar * Operator(in), where in and out are local vectors.

virtual void **MoveToAccelerator**(void)

Move the object to the accelerator backend.

virtual void **MoveToHost**(void)

Move the object to the host backend.

3.8 Global Matrix

template<typename **ValueType**>

class **GlobalMatrix** : public rocalution::*Operator*<*ValueType*>

GlobalMatrix class.

A *GlobalMatrix* is called global, because it can stay on a single or on multiple nodes in a network. For this type of communication, MPI is used.

A number of matrix formats are supported. These are CSR, BCSR, MCSR, COO, DIA, ELL, HYB, and DENSE.

Note: For CSR type matrices, the column indices must be sorted in increasing order. For COO matrices, the row indices must be sorted in increasing order. The function `Check` can be used to check whether a matrix contains valid data. For CSR and COO matrices, the function `Sort` can be used to sort the row or column indices respectively.

Template Parameters

ValueType – - can be int, float, double, std::complex<float> and std::complex<double>

Public Functions

explicit **GlobalMatrix**(const *ParallelManager* &pm)

Initialize a global matrix with a parallel manager.

virtual int64_t **GetM**(void) const

Return the number of rows in the matrix/stencil.

virtual int64_t **GetN**(void) const

Return the number of columns in the matrix/stencil.

virtual int64_t **GetNnz**(void) const

Return the number of non-zeros in the matrix/stencil.

virtual int64_t **GetLocalM**(void) const

Return the number of rows in the local matrix/stencil.

virtual int64_t **GetLocalN**(void) const

Return the number of columns in the local matrix/stencil.

virtual int64_t **GetLocalNnz**(void) const

Return the number of non-zeros in the local matrix/stencil.

virtual int64_t **GetGhostM**(void) const

Return the number of rows in the ghost matrix/stencil.

virtual int64_t **GetGhostN**(void) const

Return the number of columns in the ghost matrix/stencil.

virtual int64_t **GetGhostNnz**(void) const

Return the number of non-zeros in the ghost matrix/stencil.

virtual void **MoveToAccelerator**(void)

Move the object to the accelerator backend.

virtual void **MoveToHost**(void)

Move the object to the host backend.

virtual void **Info**(void) const

Print object information.

`Info` can print object information about any rocALUTION object. This information consists of object properties and backend data.

Example

```
mat.Info();
vec.Info();
```

virtual bool **Check**(void) const

Perform a sanity check of the matrix.

Checks, if the matrix contains valid data, i.e. if the values are not infinity and not NaN (not a number) and if the structure of the matrix is correct (e.g. indices cannot be negative, CSR and COO matrices have to be sorted, etc.).

Return values

- **true** – if the matrix is ok (empty matrix is also ok).
- **false** – if there is something wrong with the structure or values.

void **AllocateCSR**(const std::string &name, int64_t local_nnz, int64_t ghost_nnz)

Allocate CSR Matrix.

void **AllocateCOO**(const std::string &name, int64_t local_nnz, int64_t ghost_nnz)

Allocate COO Matrix.

virtual void **Clear**(void)

Clear (free all data) the object.

void **SetParallelManager**(const *ParallelManager* &pm)

Set the parallel manager of a global matrix.

void **SetDataPtrCSR**(PtrType **local_row_offset, int **local_col, *ValueType* **local_val, PtrType **ghost_row_offset, int **ghost_col, *ValueType* **ghost_val, std::string name, int64_t local_nnz, int64_t ghost_nnz)

Initialize a CSR matrix on the host with externally allocated data.

void **SetDataPtrCOO**(int **local_row, int **local_col, *ValueType* **local_val, int **ghost_row, int **ghost_col, *ValueType* **ghost_val, std::string name, int64_t local_nnz, int64_t ghost_nnz)

Initialize a COO matrix on the host with externally allocated data.

void **SetLocalDataPtrCSR**(PtrType **row_offset, int **col, *ValueType* **val, std::string name, int64_t nnz)

Initialize a CSR matrix on the host with externally allocated local data.

void **SetLocalDataPtrCOO**(int **row, int **col, *ValueType* **val, std::string name, int64_t nnz)

Initialize a COO matrix on the host with externally allocated local data.

void **SetGhostDataPtrCSR**(PtrType **row_offset, int **col, *ValueType* **val, std::string name, int64_t nnz)

Initialize a CSR matrix on the host with externally allocated ghost data.

void **SetGhostDataPtrCOO**(int **row, int **col, *ValueType* **val, std::string name, int64_t nnz)

Initialize a COO matrix on the host with externally allocated ghost data.

void **LeaveDataPtrCSR**(PtrType **local_row_offset, int **local_col, *ValueType* **local_val, PtrType **ghost_row_offset, int **ghost_col, *ValueType* **ghost_val)

Leave a CSR matrix to host pointers.

void **LeaveDataPtrCOO**(int **local_row, int **local_col, *ValueType* **local_val, int **ghost_row, int **ghost_col, *ValueType* **ghost_val)

Leave a COO matrix to host pointers.

void **LeaveLocalDataPtrCSR**(PtrType **row_offset, int **col, *ValueType* **val)

Leave a local CSR matrix to host pointers.

void **LeaveLocalDataPtrCOO**(int **row, int **col, *ValueType* **val)

Leave a local COO matrix to host pointers.

void **LeaveGhostDataPtrCSR**(PtrType **row_offset, int **col, *ValueType* **val)

Leave a CSR ghost matrix to host pointers.

void **LeaveGhostDataPtrCOO**(int **row, int **col, *ValueType* **val)

Leave a COO ghost matrix to host pointers.

void **CloneFrom**(const *GlobalMatrix*<*ValueType*> &src)

Clone the entire matrix (values,structure+backend descr) from another *GlobalMatrix*.

void **CopyFrom**(const *GlobalMatrix*<*ValueType*> &src)

Copy matrix (values and structure) from another *GlobalMatrix*.

void **ConvertToCSR**(void)

Convert the matrix to CSR structure.

void **ConvertToMCSR**(void)

Convert the matrix to MCSR structure.

void **ConvertToBCSR**(int blockdim)

Convert the matrix to BCSR structure.

void **ConvertToCOO**(void)

Convert the matrix to COO structure.

void **ConvertToELL**(void)

Convert the matrix to ELL structure.

void **ConvertToDIA**(void)

Convert the matrix to DIA structure.

void **ConvertToHYB**(void)

Convert the matrix to HYB structure.

void **ConvertToDENSE**(void)

Convert the matrix to DENSE structure.

void **ConvertTo**(unsigned int matrix_format, int blockdim = 1)

Convert the matrix to specified matrix ID format.

virtual void **Apply**(const *GlobalVector*<*ValueType*> &in, *GlobalVector*<*ValueType*> *out) const

Apply the operator, out = Operator(in), where in and out are global vectors.

virtual void **ApplyAdd**(const *GlobalVector*<*ValueType*> &in, *ValueType* scalar, *GlobalVector*<*ValueType*> *out) const

Apply and add the operator, out += scalar * Operator(in), where in and out are global vectors.

virtual void **Transpose**(void)

Transpose the matrix.

void **Transpose**(*GlobalMatrix*<*ValueType*> *T) const

Transpose the matrix.

void **TripleMatrixProduct**(const *GlobalMatrix*<*ValueType*> &R, const *GlobalMatrix*<*ValueType*> &A, const *GlobalMatrix*<*ValueType*> &P)

Triple matrix product C=RAP.

void **ReadFileMTX**(const std::string &filename)

Read matrix from MTX (Matrix Market Format) file.

void **WriteFileMTX**(const std::string &filename) const

Write matrix to MTX (Matrix Market Format) file.

void **ReadFileCSR**(const std::string &filename)

Read matrix from CSR (ROCALUTION binary format) file.

void **WriteFileCSR**(const std::string &filename) const

Write matrix to CSR (ROCALUTION binary format) file.

void **Sort**(void)

Sort the matrix indices.

Sorts the matrix by indices.

- For CSR matrices, column values are sorted.
- For COO matrices, row indices are sorted.

void **ExtractInverseDiagonal**(*GlobalVector*<*ValueType*> *vec_inv_diag) const

Extract the inverse (reciprocal) diagonal values of the matrix into a *GlobalVector*.

void **Scale**(*ValueType* alpha)

Scale all the values in the matrix.

void **InitialPairwiseAggregation**(*ValueType* beta, int &nc, *LocalVector*<int> *G, int &Gsize, int **rG, int &rGsize, int ordering) const

Initial Pairwise Aggregation scheme.

void **FurtherPairwiseAggregation**(*ValueType* beta, int &nc, *LocalVector*<int> *G, int &Gsize, int **rG, int &rGsize, int ordering) const

Further Pairwise Aggregation scheme.

void **CoarsenOperator**(*GlobalMatrix*<*ValueType*> *Ac, int nrow, int ncol, const *LocalVector*<int> &G, int Gsize, const int *rG, int rGsize) const

Build coarse operator for pairwise aggregation scheme.

void **CreateFromMap**(const *LocalVector*<int> &map, int64_t n, int64_t m, *GlobalMatrix*<*ValueType*> *pro)

Create a restriction and prolongation matrix operator based on an int vector map.

3.9 Local Vector

template<typename **ValueType**>

class **LocalVector** : public rocalution::*Vector*<*ValueType*>

LocalVector class.

A *LocalVector* is called local, because it will always stay on a single system. The system can contain several CPUs via UMA or NUMA memory system or it can contain an accelerator.

Template Parameters
ValueType – - can be int, float, double, std::complex<float> and std::complex<double>

Unnamed Group

ValueType &**operator[]**(int64_t i)

Access operator (only for host data)

The elements in the vector can be accessed via [] operators, when the vector is allocated on the host.

Example

```
// rocALUTION local vector object
LocalVector<ValueType> vec;

// Allocate vector
vec.Allocate("my_vector", 100);

// Initialize vector with 1
vec.Ones();

// Set even elements to -1
for(int64_t i = 0; i < vec.GetSize(); i += 2)
{
  vec[i] = -1;
}
```

Parameters
i – **[in]** access data at index **i**

Returns
value at index **i**

const *ValueType* &**operator[]**(int64_t i) const

Access operator (only for host data)

The elements in the vector can be accessed via [] operators, when the vector is allocated on the host.

Example

```
// rocALUTION local vector object
LocalVector<ValueType> vec;

// Allocate vector
vec.Allocate("my_vector", 100);

// Initialize vector with 1
vec.Ones();

// Set even elements to -1
for(int64_t i = 0; i < vec.GetSize(); i += 2)
{
  vec[i] = -1;
}
```

Parameters
i – **[in]** access data at index i

Returns
value at index i

Public Functions

virtual void **MoveToAccelerator**(void)

Move the object to the accelerator backend.

virtual void **MoveToAcceleratorAsync**(void)

Move the object to the accelerator backend with async move.

virtual void **MoveToHost**(void)

Move the object to the host backend.

virtual void **MoveToHostAsync**(void)

Move the object to the host backend with async move.

virtual void **Sync**(void)

Sync (the async move)

virtual void **Info**(void) const

Print object information.

Info can print object information about any rocALUTION object. This information consists of object properties and backend data.

Example

```
mat.Info();
vec.Info();
```

virtual int64_t **GetSize**(void) const

Return the size of the vector.

virtual bool **Check**(void) const

Perform a sanity check of the vector.

Checks, if the vector contains valid data, i.e. if the values are not infinity and not NaN (not a number).

Return values

- **true** – if the vector is ok (empty vector is also ok).
- **false** – if there is something wrong with the values.

void **Allocate**(std::string name, int64_t size)

Allocate a local vector with name and size.

The local vector allocation function requires a name of the object (this is only for information purposes) and corresponding size description for vector objects.

Example

```
LocalVector<ValueType> vec;

vec.Allocate("my vector", 100);
vec.Clear();
```

Parameters

- **name** – **[in]** object name
- **size** – **[in]** number of elements in the vector

void **SetDataPtr**(*ValueType* **ptr, std::string name, int64_t size)

Initialize a *LocalVector* on the host with externally allocated data.

SetDataPtr has direct access to the raw data via pointers. Already allocated data can be set by passing the pointer.

Example

```
// Allocate vector
ValueType* ptr_vec = new ValueType[200];

// Fill vector
// ...

// rocALUTION local vector object
LocalVector<ValueType> vec;

// Set the vector data, ptr_vec will become invalid
vec.SetDataPtr(&ptr_vec, "my_vector", 200);
```

Note: Setting data pointer will leave the original pointer empty (set to NULL).

void **LeaveDataPtr**(*ValueType* **ptr)

Leave a *LocalVector* to host pointers.

LeaveDataPtr has direct access to the raw data via pointers. A *LocalVector* object can leave its raw data to a host pointer. This will leave the *LocalVector* empty.

Example

```
// rocALUTION local vector object
LocalVector<ValueType> vec;

// Allocate the vector
vec.Allocate("my_vector", 100);

// Fill vector
// ...

ValueType* ptr_vec = NULL;

// Get (steal) the data from the vector, this will leave the local vector␣
↪object empty
vec.LeaveDataPtr(&ptr_vec);
```

virtual void **Clear**()

Clear (free all data) the object.

virtual void **Zeros**()

Set all values of the vector to 0.

virtual void **Ones**()

Set all values of the vector to 1.

virtual void **SetValues**(*ValueType* val)

Set all values of the vector to given argument.

virtual void **SetRandomUniform**(unsigned long long seed, *ValueType* a = static_cast<*ValueType*>(-1), *ValueType* b = static_cast<*ValueType*>(1))

Fill the vector with random values from interval [a,b].

virtual void **SetRandomNormal**(unsigned long long seed, *ValueType* mean = static_cast<*ValueType*>(0), *ValueType* var = static_cast<*ValueType*>(1))

Fill the vector with random values from normal distribution.

virtual void **ReadFileASCII**(const std::string &filename)

Read vector from ASCII file.

Read a vector from ASCII file.

Example

```
LocalVector<ValueType> vec;
vec.ReadFileASCII("my_vector.dat");
```

Parameters
filename – **[in]** name of the file containing the ASCII data.

virtual void **WriteFileASCII**(const std::string &filename) const

Write vector to ASCII file.

Write a vector to ASCII file.

Example

```
LocalVector<ValueType> vec;

// Allocate and fill vec
// ...

vec.WriteFileASCII("my_vector.dat");
```

Parameters
filename – **[in]** name of the file to write the ASCII data to.

virtual void **ReadFileBinary**(const std::string &filename)

Read vector from binary file.

Read a vector from binary file. For details on the format, see *WriteFileBinary()*.

Example

```
LocalVector<ValueType> vec;
vec.ReadFileBinary("my_vector.bin");
```

Parameters
filename – **[in]** name of the file containing the data.

virtual void **WriteFileBinary**(const std::string &filename) const

Write vector to binary file.

Write a vector to binary file.

The binary format contains a header, the rocALUTION version and the vector data as follows

```
// Header
out << "#rocALUTION binary vector file" << std::endl;

// rocALUTION version
out.write((char*)&version, sizeof(int));

// Vector data
out.write((char*)&size, sizeof(int));
out.write((char*)vec_val, size * sizeof(double));
```

Example

```
LocalVector<ValueType> vec;

// Allocate and fill vec
// ...

vec.WriteFileBinary("my_vector.bin");
```

Note: *Vector* values array is always stored in double precision (e.g. double or std::complex<double>).

Parameters
filename – **[in]** name of the file to write the data to.

virtual void **CopyFrom**(const *LocalVector*<*ValueType*> &src)

Copy vector from another vector.

`CopyFrom` copies values from another vector.

Example

```
LocalVector<ValueType> vec1, vec2;

// Allocate and initialize vec1 and vec2
// ...

// Move vec1 to accelerator
// vec1.MoveToAccelerator();

// Now, vec1 is on the accelerator (if available)
// and vec2 is on the host

// Copy vec1 to vec2 (or vice versa) will move data between host and
// accelerator backend
vec1.CopyFrom(vec2);
```

Note: This function allows cross platform copying. One of the objects could be allocated on the accelerator backend.

Parameters
src – **[in]** *Vector*, where values should be copied from.

virtual void **CopyFromAsync**(const *LocalVector*<*ValueType*> &src)

Async copy from another local vector.

virtual void **CopyFromFloat**(const *LocalVector*<float> &src)

Copy values from another local float vector.

virtual void **CopyFromDouble**(const *LocalVector*<double> &src)

Copy values from another local double vector.

virtual void **CopyFrom**(const *LocalVector*<*ValueType*> &src, int64_t src_offset, int64_t dst_offset, int64_t size)

Copy vector from another vector with offsets and size.

`CopyFrom` copies values with specific source and destination offsets and sizes from another vector.

Note: This function allows cross platform copying. One of the objects could be allocated on the accelerator backend.

Parameters

- **src** – **[in]** *Vector*, where values should be copied from.
- **src_offset** – **[in]** source offset.
- **dst_offset** – **[in]** destination offset.
- **size** – **[in]** number of entries to be copied.

void **CopyFromPermute**(const *LocalVector*<*ValueType*> &src, const *LocalVector*<int> &permutation)

Copy a vector under permutation (forward permutation)

void **CopyFromPermuteBackward**(const *LocalVector*<*ValueType*> &src, const *LocalVector*<int> &permutation)

Copy a vector under permutation (backward permutation)

virtual void **CloneFrom**(const *LocalVector*<*ValueType*> &src)

Clone the vector.

`CloneFrom` clones the entire vector, with data and backend descriptor from another *Vector*.

Example

```
LocalVector<ValueType> vec;

// Allocate and initialize vec (host or accelerator)
// ...

LocalVector<ValueType> tmp;

// By cloning vec, tmp will have identical values and will be on the same
// backend as vec
tmp.CloneFrom(vec);
```

Parameters

src – **[in]** *Vector* to clone from.

void **CopyFromData**(const *ValueType* *data)

Copy (import) vector.

Copy (import) vector data that is described in one array (values). The object data has to be allocated with *Allocate()*, using the corresponding size of the data, first.

Parameters

data – **[in]** data to be imported.

void **CopyFromHostData**(const *ValueType* *data)

Copy (import) vector from host data.

Copy (import) vector data that is described in one host array (values). The object data has to be allocated with *Allocate()*, using the corresponding size of the data, first.

Parameters
data – **[in]** data to be imported from host.

void **CopyToData**(*ValueType* *data) const

Copy (export) vector.

Copy (export) vector data that is described in one array (values). The output array has to be allocated, using the corresponding size of the data, first. Size can be obtain by *GetSize()*.

Parameters
data – **[out]** exported data.

void **Permute**(const *LocalVector*<int> &permutation)

Perform in-place permutation (forward) of the vector.

void **PermuteBackward**(const *LocalVector*<int> &permutation)

Perform in-place permutation (backward) of the vector.

void **Restriction**(const *LocalVector*<*ValueType*> &vec_fine, const *LocalVector*<int> &map)

Restriction operator based on restriction mapping vector.

void **Prolongation**(const *LocalVector*<*ValueType*> &vec_coarse, const *LocalVector*<int> &map)

Prolongation operator based on restriction mapping vector.

virtual void **AddScale**(const *LocalVector*<*ValueType*> &x, *ValueType* alpha)

Perform vector update of type this = this + alpha * x.

virtual void **ScaleAdd**(*ValueType* alpha, const *LocalVector*<*ValueType*> &x)

Perform vector update of type this = alpha * this + x.

virtual void **ScaleAddScale**(*ValueType* alpha, const *LocalVector*<*ValueType*> &x, *ValueType* beta)

Perform vector update of type this = alpha * this + x * beta.

virtual void **ScaleAddScale**(*ValueType* alpha, const *LocalVector*<*ValueType*> &x, *ValueType* beta, int64_t src_offset, int64_t dst_offset, int64_t size)

Perform vector update of type this = alpha * this + x * beta with offsets.

virtual void **ScaleAdd2**(*ValueType* alpha, const *LocalVector*<*ValueType*> &x, *ValueType* beta, const *LocalVector*<*ValueType*> &y, *ValueType* gamma)

Perform vector update of type this = alpha * this + x * beta + y * gamma.

virtual void **Scale**(*ValueType* alpha)

Perform vector scaling this = alpha * this.

virtual *ValueType* **Dot**(const *LocalVector*<*ValueType*> &x) const

Compute dot (scalar) product, return this^T y.

virtual *ValueType* **DotNonConj**(const *LocalVector*<*ValueType*> &x) const

Compute non-conjugate dot (scalar) product, return this^T y.

virtual *ValueType* **Norm**(void) const

Compute L_2 norm of the vector, return = srqt(this^T this)

virtual *ValueType* **Reduce**(void) const

Reduce the vector.

virtual *ValueType* **InclusiveSum**(void)

Compute Inclusive sum.

virtual *ValueType* **ExclusiveSum**(void)

Compute exclusive sum.

virtual *ValueType* **Asum**(void) const

Compute the sum of absolute values of the vector, return = sum(|this|)

virtual int64_t **Amax**(*ValueType* &value) const

Compute the absolute max of the vector, return = index(max(|this|))

virtual void **PointWiseMult**(const *LocalVector*<*ValueType*> &x)

Perform point-wise multiplication (element-wise) of this = this * x.

virtual void **PointWiseMult**(const *LocalVector*<*ValueType*> &x, const *LocalVector*<*ValueType*> &y)

Perform point-wise multiplication (element-wise) of this = x * y.

virtual void **Power**(double power)

Perform power operation to a vector.

void **GetIndexValues**(const *LocalVector*<int> &index, *LocalVector*<*ValueType*> *values) const

Get indexed values.

void **SetIndexValues**(const *LocalVector*<int> &index, const *LocalVector*<*ValueType*> &values)

Set indexed values.

void **GetContinuousValues**(int64_t start, int64_t end, *ValueType* *values) const

Get continuous indexed values.

void **SetContinuousValues**(int64_t start, int64_t end, const *ValueType* *values)

Set continuous indexed values.

void **ExtractCoarseMapping**(int64_t start, int64_t end, const int *index, int nc, int *size, int *map) const

Extract coarse boundary mapping.

void **ExtractCoarseBoundary**(int64_t start, int64_t end, const int *index, int nc, int *size, int *boundary) const

Extract coarse boundary index.

3.10 Global Vector

template<typename **ValueType**>

class **GlobalVector** : public rocalution::*Vector*<*ValueType*>

GlobalVector class.

A *GlobalVector* is called global, because it can stay on a single or on multiple nodes in a network. For this type of communication, MPI is used.

Template Parameters

ValueType – - can be int, float, double, std::complex<float> and std::complex<double>

Public Functions

explicit **GlobalVector**(const *ParallelManager* &pm)

Initialize a global vector with a parallel manager.

virtual void **MoveToAccelerator**(void)

Move the object to the accelerator backend.

virtual void **MoveToHost**(void)

Move the object to the host backend.

virtual void **Info**(void) const

Print object information.

`Info` can print object information about any rocALUTION object. This information consists of object properties and backend data.

Example

```
mat.Info();
vec.Info();
```

virtual bool **Check**(void) const

Perform a sanity check of the vector.

Checks, if the vector contains valid data, i.e. if the values are not infinity and not NaN (not a number).

Return values

- **true** – if the vector is ok (empty vector is also ok).
- **false** – if there is something wrong with the values.

virtual int64_t **GetSize**(void) const

Return the size of the vector.

virtual int64_t **GetLocalSize**(void) const

Return the size of the local vector.

virtual void **Allocate**(std::string name, int64_t size)

Allocate a global vector with name and size.

virtual void **Clear**(void)

Clear (free all data) the object.

void **SetParallelManager**(const *ParallelManager* &pm)

Set the parallel manager of a global vector.

virtual void **Zeros**(void)

Set all values of the vector to 0.

virtual void **Ones**(void)

Set all values of the vector to 1.

virtual void **SetValues**(*ValueType* val)

Set all values of the vector to given argument.

virtual void **SetRandomUniform**(unsigned long long seed, *ValueType* a = static_cast<*ValueType*>(-1), *ValueType* b = static_cast<*ValueType*>(1))

Fill the vector with random values from interval [a,b].

virtual void **SetRandomNormal**(unsigned long long seed, *ValueType* mean = static_cast<*ValueType*>(0), *ValueType* var = static_cast<*ValueType*>(1))

Fill the vector with random values from normal distribution.

virtual void **CloneFrom**(const *GlobalVector*<*ValueType*> &src)

Clone the vector.

`CloneFrom` clones the entire vector, with data and backend descriptor from another *Vector*.

Example

```
LocalVector<ValueType> vec;

// Allocate and initialize vec (host or accelerator)
// ...

LocalVector<ValueType> tmp;

// By cloning vec, tmp will have identical values and will be on the same
// backend as vec
tmp.CloneFrom(vec);
```

Parameters
src – **[in]** *Vector* to clone from.

ValueType &**operator[]**(int64_t i)

Access operator (only for host data)

const *ValueType* &**operator[]**(int64_t i) const

Access operator (only for host data)

void **SetDataPtr**(*ValueType* **ptr, std::string name, int64_t size)

Initialize the local part of a global vector with externally allocated data.

void **LeaveDataPtr**(*ValueType* **ptr)

Get a pointer to the data from the local part of a global vector and free the global vector object.

virtual void **CopyFrom**(const *GlobalVector*<*ValueType*> &src)

Copy vector from another vector.

`CopyFrom` copies values from another vector.

Example

```
LocalVector<ValueType> vec1, vec2;

// Allocate and initialize vec1 and vec2
// ...
```

(continues on next page)

(continued from previous page)

```
// Move vec1 to accelerator
// vec1.MoveToAccelerator();

// Now, vec1 is on the accelerator (if available)
// and vec2 is on the host

// Copy vec1 to vec2 (or vice versa) will move data between host and
// accelerator backend
vec1.CopyFrom(vec2);
```

Note: This function allows cross platform copying. One of the objects could be allocated on the accelerator backend.

Parameters

src – **[in]** *Vector*, where values should be copied from.

virtual void **ReadFileASCII**(const std::string &filename)

Read vector from ASCII file.

Read a vector from ASCII file.

Example

```
LocalVector<ValueType> vec;
vec.ReadFileASCII("my_vector.dat");
```

Parameters

filename – **[in]** name of the file containing the ASCII data.

virtual void **WriteFileASCII**(const std::string &filename) const

Write vector to ASCII file.

Write a vector to ASCII file.

Example

```
LocalVector<ValueType> vec;

// Allocate and fill vec
// ...

vec.WriteFileASCII("my_vector.dat");
```

Parameters

filename – **[in]** name of the file to write the ASCII data to.

virtual void **ReadFileBinary**(const std::string &filename)

Read vector from binary file.

Read a vector from binary file. For details on the format, see *WriteFileBinary()*.

Example

```
LocalVector<ValueType> vec;
vec.ReadFileBinary("my_vector.bin");
```

Parameters

filename – **[in]** name of the file containing the data.

virtual void **WriteFileBinary**(const std::string &filename) const

Write vector to binary file.

Write a vector to binary file.

The binary format contains a header, the rocALUTION version and the vector data as follows

```
// Header
out << "#rocALUTION binary vector file" << std::endl;

// rocALUTION version
out.write((char*)&version, sizeof(int));

// Vector data
out.write((char*)&size, sizeof(int));
out.write((char*)vec_val, size * sizeof(double));
```

Example

```
LocalVector<ValueType> vec;

// Allocate and fill vec
// ...

vec.WriteFileBinary("my_vector.bin");
```

Note: *Vector* values array is always stored in double precision (e.g. double or std::complex<double>).

Parameters

filename – **[in]** name of the file to write the data to.

virtual void **AddScale**(const *GlobalVector*<*ValueType*> &x, *ValueType* alpha)

Perform vector update of type this = this + alpha * x.

virtual void **ScaleAdd**(*ValueType* alpha, const *GlobalVector*<*ValueType*> &x)

Perform vector update of type this = alpha * this + x.

virtual void **ScaleAdd2**(*ValueType* alpha, const *GlobalVector*<*ValueType*> &x, *ValueType* beta, const *GlobalVector*<*ValueType*> &y, *ValueType* gamma)

Perform vector update of type this = alpha * this + x * beta + y * gamma.

virtual void **ScaleAddScale**(*ValueType* alpha, const *GlobalVector*<*ValueType*> &x, *ValueType* beta)

Perform vector update of type this = alpha * this + x * beta.

virtual void **Scale**(*ValueType* alpha)

Perform vector scaling this = alpha * this.

virtual *ValueType* **Dot**(const *GlobalVector*<*ValueType*> &x) const

Compute dot (scalar) product, return this^T y.

virtual *ValueType* **DotNonConj**(const *GlobalVector*<*ValueType*> &x) const

Compute non-conjugate dot (scalar) product, return this^T y.

virtual *ValueType* **Norm**(void) const

Compute L_2 norm of the vector, return = srqt(this^T this)

virtual *ValueType* **Reduce**(void) const

Reduce the vector.

virtual *ValueType* **InclusiveSum**(void)

Compute Inclusive sum.

virtual *ValueType* **ExclusiveSum**(void)

Compute exclusive sum.

virtual *ValueType* **Asum**(void) const

Compute the sum of absolute values of the vector, return = sum(|this|)

virtual int64_t **Amax**(*ValueType* &value) const

Compute the absolute max of the vector, return = index(max(|this|))

virtual void **PointWiseMult**(const *GlobalVector*<*ValueType*> &x)

Perform point-wise multiplication (element-wise) of this = this * x.

virtual void **PointWiseMult**(const *GlobalVector*<*ValueType*> &x, const *GlobalVector*<*ValueType*> &y)

Perform point-wise multiplication (element-wise) of this = x * y.

virtual void **Power**(double power)

Perform power operation to a vector.

void **Restriction**(const *GlobalVector*<*ValueType*> &vec_fine, const *LocalVector*<int> &map)

Restriction operator based on restriction mapping vector.

void **Prolongation**(const *GlobalVector*<*ValueType*> &vec_coarse, const *LocalVector*<int> &map)

Prolongation operator based on restriction mapping vector.

3.11 Base Classes

template<typename **ValueType**>

class **BaseMatrix**

template<typename **ValueType**>

class **BaseStencil**

template<typename **ValueType**>

class **BaseVector**

template<typename **ValueType**>

class **HostMatrix**

template<typename **ValueType**>

class **HostStencil**

template<typename **ValueType**>

class **HostVector**

template<typename **ValueType**>

class **AcceleratorMatrix**

template<typename **ValueType**>

class **AcceleratorStencil**

template<typename **ValueType**>

class **AcceleratorVector**

3.12 Parallel Manager

class **ParallelManager** : public rocalution::RocalutionObj

Parallel Manager class.

The parallel manager class handles the communication and the mapping of the global operators. Each global operator and vector need to be initialized with a valid parallel manager in order to perform any operation. For many distributed simulations, the underlying operator is already distributed. This information need to be passed to the parallel manager.

Public Functions

void **SetMPICommunicator**(const void *comm)
Set the MPI communicator.

void **Clear**(void)
Clear all allocated resources.

inline const void ***GetComm**(void) const
Return communicator.

inline int **GetRank**(void) const
Return rank.

int64_t **GetGlobalNrow**(void) const
Return the global number of rows.

int64_t **GetGlobalNcol**(void) const
Return the global number of columns.

int64_t **GetLocalNrow**(void) const
Return the local number of rows.

int64_t **GetLocalNcol**(void) const
Return the local number of columns.

int **GetNumReceivers**(void) const
Return the number of receivers.

int **GetNumSenders**(void) const
Return the number of senders.

int **GetNumProcs**(void) const
Return the number of involved processes.

void **SetGlobalNrow**(int64_t nrow)
Initialize the global number of rows.

void **SetGlobalNcol**(int64_t ncol)
Initialize the global number of columns.

void **SetLocalNrow**(int64_t nrow)
Initialize the local number of rows.

void **SetLocalNcol**(int64_t ncol)
Initialize the local number of columns.

void **SetBoundaryIndex**(int size, const int *index)
Set all boundary indices of this ranks process.

const int ***GetBoundaryIndex**(void) const
Get all boundary indices of this ranks process.

void **SetReceivers**(int nrecv, const int *recvs, const int *recv_offset)
Number of processes, the current process is receiving data from, array of the processes, the current process is receiving data from and offsets, where the boundary for process 'receiver' starts.

void **SetSenders**(int nsend, const int *sends, const int *send_offset)

Number of processes, the current process is sending data to, array of the processes, the current process is sending data to and offsets where the ghost part for process 'sender' starts.

void **LocalToGlobal**(int proc, int local, int &global)

Mapping local to global.

void **GlobalToLocal**(int global, int &proc, int &local)

Mapping global to local.

bool **Status**(void) const

Check sanity status of parallel manager.

void **ReadFileASCII**(const std::string &filename)

Read file that contains all relevant parallel manager data.

void **WriteFileASCII**(const std::string &filename) const

Write file that contains all relevant parallel manager data.

3.13 Solvers

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **Solver** : public rocalution::RocalutionObj

Base class for all solvers and preconditioners.

Most of the solvers can be performed on linear operators *LocalMatrix*, *LocalStencil* and *GlobalMatrix* - i.e. the solvers can be performed locally (on a shared memory system) or in a distributed manner (on a cluster) via MPI. The only exception is the AMG (Algebraic Multigrid) solver which has two versions (one for *LocalMatrix* and one for *GlobalMatrix* class). The only pure local solvers (which do not support global/MPI operations) are the mixed-precision defect-correction solver and all direct solvers.

All solvers need three template parameters - Operators, Vectors and Scalar type.

The *Solver* class is purely virtual and provides an interface for

- *SetOperator()* to set the operator A, i.e. the user can pass the matrix here.
- *Build()* to build the solver (including preconditioners, sub-solvers, etc.). The user need to specify the operator first before calling *Build()*.
- *Solve()* to solve the system $Ax = b$. The user need to pass a right-hand-side b and a vector x, where the solution will be obtained.
- *Print()* to show solver information.
- *ReBuildNumeric()* to only re-build the solver numerically (if possible).
- *MoveToHost()* and *MoveToAccelerator()* to offload the solver (including preconditioners and sub-solvers) to the host/accelerator.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::DirectLinearSolver< OperatorType, VectorType, ValueType >*, *rocalution::IterativeLinearSolver< OperatorType, VectorType, ValueType >*, *rocalution::Preconditioner< OperatorType, VectorType, ValueType >*

Public Functions

void **SetOperator**(const *OperatorType* &op)

Set the *Operator* of the solver.

virtual void **ResetOperator**(const *OperatorType* &op)

Reset the operator; see *ReBuildNumeric()*

virtual void **Print**(void) const = 0

Print information about the solver.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x) = 0

Solve *Operator* x = rhs.

virtual void **SolveZeroSol**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs, setting initial x = 0.

virtual void **Clear**(void)

Clear (free all local data) the solver.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **BuildMoveToAcceleratorAsync**(void)

Build the solver and move it to the accelerator asynchronously.

virtual void **Sync**(void)

Synchronize the solver.

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

virtual void **MoveToHost**(void)

Move all data (i.e. move the solver) to the host.

virtual void **MoveToAccelerator**(void)

Move all data (i.e. move the solver) to the accelerator.

virtual void **Verbose**(int verb = 1)

Provide verbose output of the solver.

- verb = 0 -> no output
- verb = 1 -> print info about the solver (start, end);
- verb = 2 -> print (iter, residual) via iteration control;

3.13.1 Iterative Linear Solvers

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **IterativeLinearSolver** : public rocalution::*Solver*<*OperatorType*, *VectorType*, *ValueType*>

Base class for all linear iterative solvers.

The iterative solvers are controlled by an iteration control object, which monitors the convergence properties of the solver, i.e. maximum number of iteration, relative tolerance, absolute tolerance and divergence tolerance. The iteration control can also record the residual history and store it in an ASCII file.

- *Init()*, *InitMinIter()*, *InitMaxIter()* and *InitTol()* initialize the solver and set the stopping criteria.
- *RecordResidualHistory()* and *RecordHistory()* start the recording of the residual and write it into a file.
- *Verbose()* sets the level of verbose output of the solver (0 - no output, 2 - detailed output, including residual and iteration information).
- *SetPreconditioner()* sets the preconditioning.

All iterative solvers are controlled based on

- Absolute stopping criteria, when $|r_k|_{L_p} < \epsilon_{abs}$
- Relative stopping criteria, when $|r_k|_{L_p}/|r_1|_{L_p} \leq \epsilon_{rel}$
- Divergence stopping criteria, when $|r_k|_{L_p}/|r_1|_{L_p} \geq \epsilon_{div}$
- Maximum number of iteration N, when $k = N$

where k is the current iteration, r_k the residual for the current iteration k (i.e. $r_k = b - Ax_k$) and r_1 the starting residual (i.e. $r_1 = b - Ax_{init}$). In addition, the minimum number of iterations M can be specified. In this case, the solver will not stop to iterate, before $k \geq M$.

The L_p norm is used for the computation, where p could be 1, 2 and ∞. The norm computation can be set with *SetResidualNorm()* with 1 for L_1, 2 for L_2 and 3 for L_∞. For the computation with L_∞, the index of the maximum value can be obtained with *GetAmaxResidualIndex()*. If this function is called and L_∞ was not selected, this function will return -1.

The reached criteria can be obtained with *GetSolverStatus()*, returning

- 0, if no criteria has been reached yet
- 1, if absolute tolerance has been reached
- 2, if relative tolerance has been reached
- 3, if divergence tolerance has been reached
- 4, if maximum number of iteration has been reached

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::BaseMultiGrid< OperatorType, VectorType, ValueType >*, *rocalution::BiCGStab< OperatorType, VectorType, ValueType >*, *rocalution::BiCGStabl< OperatorType, VectorType, ValueType*

>, *rocalution::CG< OperatorType, VectorType, ValueType >*, *rocalution::Chebyshev< OperatorType, VectorType, ValueType >*, *rocalution::CR< OperatorType, VectorType, ValueType >*, *rocalution::FCG< OperatorType, VectorType, ValueType >*, *rocalution::FGMRES< OperatorType, VectorType, ValueType >*, *rocalution::FixedPoint< OperatorType, VectorType, ValueType >*, *rocalution::GMRES< OperatorType, VectorType, ValueType >*, *rocalution::IDR< OperatorType, VectorType, ValueType >*, *rocalution::QMRCGStab< OperatorType, VectorType, ValueType >*

Public Functions

void **Init**(double abs_tol, double rel_tol, double div_tol, int max_iter)

Initialize the solver with absolute/relative/divergence tolerance and maximum number of iterations.

void **Init**(double abs_tol, double rel_tol, double div_tol, int min_iter, int max_iter)

Initialize the solver with absolute/relative/divergence tolerance and minimum/maximum number of iterations.

void **InitMinIter**(int min_iter)

Set the minimum number of iterations.

void **InitMaxIter**(int max_iter)

Set the maximum number of iterations.

void **InitTol**(double abs, double rel, double div)

Set the absolute/relative/divergence tolerance.

void **SetResidualNorm**(int resnorm)

Set the residual norm to L_1, L_2 or L_∞ norm.

- resnorm = 1 -> L_1 norm
- resnorm = 2 -> L_2 norm
- resnorm = 3 -> L_∞ norm

void **RecordResidualHistory**(void)

Record the residual history.

void **RecordHistory**(const std::string &filename) const

Write the history to file.

virtual void **Verbose**(int verb = 1)

Set the solver verbosity output.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

virtual void **SetPreconditioner**(*Solver*<*OperatorType*, *VectorType*, *ValueType*> &precond)

Set a preconditioner of the linear solver.

virtual int **GetIterationCount**(void)

Return the iteration count.

virtual double **GetCurrentResidual**(void)

Return the current residual.

virtual int **GetSolverStatus**(void)

Return the current status.

virtual int64_t **GetAmaxResidualIndex**(void)

Return absolute maximum index of residual vector when using L_∞ norm.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **FixedPoint** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Fixed-Point Iteration Scheme.

The Fixed-Point iteration scheme is based on additive splitting of the matrix $A = M + N$. The scheme reads

$$x_{k+1} = M^{-1}(b - Nx_k).$$

It can also be reformulated as a weighted defect correction scheme

$$x_{k+1} = x_k - \omega M^{-1}(Ax_k - b).$$

The inversion of M can be performed by preconditioners (*Jacobi*, Gauss-Seidel, *ILU*, etc.) or by any type of solvers.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

void **SetRelaxation**(*ValueType* omega)

Set relaxation parameter ω.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

virtual void **SolveZeroSol**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs, setting initial x = 0.

template<class **OperatorTypeH**, class **VectorTypeH**, typename **ValueTypeH**, class **OperatorTypeL**, class **VectorTypeL**, typename **ValueTypeL**>
class **MixedPrecisionDC** : public rocalution::*IterativeLinearSolver*<*OperatorTypeH*, *VectorTypeH*, *ValueTypeH*>

Mixed-Precision Defect Correction Scheme.

The Mixed-Precision solver is based on a defect-correction scheme. The current implementation of the library is using host based correction in double precision and accelerator computation in single precision. The solver is implemeting the scheme

$$x_{k+1} = x_k + A^{-1}r_k,$$

where the computation of the residual $r_k = b - Ax_k$ and the update $x_{k+1} = x_k + d_k$ are performed on the host in double precision. The computation of the residual system $Ad_k = r_k$ is performed on the accelerator in single precision. In addition to the setup functions of the iterative solver, the user need to specify the inner ($Ad_k = r_k$) solver.

Template Parameters

- **OperatorTypeH** – - can be *LocalMatrix*
- **VectorTypeH** – - can be *LocalVector*
- **ValueTypeH** – - can be double
- **OperatorTypeL** – - can be *LocalMatrix*
- **VectorTypeL** – - can be *LocalVector*
- **ValueTypeL** – - can be float

Public Functions

virtual void **Print**(void) const

Print information about the solver.

void **Set**(*Solver*<*OperatorTypeL*, *VectorTypeL*, *ValueTypeL*> &Solver_L)

Set the inner solver for $Ad_k = r_k$.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **Chebyshev** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Chebyshev Iteration Scheme.

The *Chebyshev* Iteration scheme (also known as acceleration scheme) is similar to the *CG* method but requires minimum and maximum eigenvalues of the operator. templates

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

void **Set**(*ValueType* lambda_min, *ValueType* lambda_max)

Set the minimum and maximum eigenvalues of the operator.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

3.13.1.1 Krylov Subspace Solvers

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **BiCGStab** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Bi-Conjugate Gradient Stabilized Method.

The Bi-Conjugate Gradient Stabilized method is a variation of CGS and solves sparse (non) symmetric linear systems $Ax = b$. SAAD

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **BiCGStabl** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Bi-Conjugate Gradient Stabilized (l) Method.

The Bi-Conjugate Gradient Stabilized (l) method is a generalization of *BiCGStab* for solving sparse (non) symmetric linear systems $Ax = b$. It minimizes residuals over l-dimensional Krylov subspaces. The degree l can be set with *SetOrder()*. bicgstabl

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

virtual void **SetOrder**(int l)

Set the order.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **CG** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Conjugate Gradient Method.

The Conjugate Gradient method is the best known iterative method for solving sparse symmetric positive definite (SPD) linear systems $Ax = b$. It is based on orthogonal projection onto the Krylov subspace $\mathcal{K}_m(r_0, A)$, where r_0 is the initial residual. The method can be preconditioned, where the approximation should also be SPD. SAAD

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **BuildMoveToAcceleratorAsync**(void)

Build the solver and move it to the accelerator asynchronously.

virtual void **Sync**(void)

Synchronize the solver.

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **CR** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Conjugate Residual Method.

The Conjugate Residual method is an iterative method for solving sparse symmetric semi-positive definite linear systems $Ax = b$. It is a Krylov subspace method and differs from the much more popular Conjugate Gradient method that the system matrix is not required to be positive definite. The method can be preconditioned where the approximation should also be SPD or semi-positive definite. SAAD

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **FCG** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Flexible Conjugate Gradient Method.

The Flexible Conjugate Gradient method is an iterative method for solving sparse symmetric positive definite linear systems $Ax = b$. It is similar to the Conjugate Gradient method with the only difference, that it allows the preconditioner M^{-1} to be not a constant operator. This can be especially helpful if the operation $M^{-1}x$ is the result of another iterative process and not a constant operator. fcg

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **GMRES** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Generalized Minimum Residual Method.

The Generalized Minimum Residual method (*GMRES*) is a projection method for solving sparse (non) symmetric linear systems $Ax = b$, based on restarting technique. The solution is approximated in a Krylov subspace $\mathcal{K} = \mathcal{K}_m$ and $\mathcal{L} = A\mathcal{K}_m$ with minimal residual, where $\mathcal{K}_m$ is the m-th Krylov subspace with $v_1 = r_0/||r_0||_2$. SAAD

The Krylov subspace basis size can be set using *SetBasisSize()*. The default size is 30.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

virtual void **SetBasisSize**(int size_basis)

Set the size of the Krylov subspace basis.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **FGMRES** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Flexible Generalized Minimum Residual Method.

The Flexible Generalized Minimum Residual method (*FGMRES*) is a projection method for solving sparse (non) symmetric linear systems $Ax = b$. It is similar to the *GMRES* method with the only difference, the *FGMRES* is based on a window shifting of the Krylov subspace and thus allows the preconditioner M^{-1} to be not a constant

operator. This can be especially helpful if the operation $M^{-1}x$ is the result of another iterative process and not a constant operator. SAAD

The Krylov subspace basis size can be set using *SetBasisSize()*. The default size is 30.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

virtual void **SetBasisSize**(int size_basis)

Set the size of the Krylov subspace basis.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **IDR** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Induced Dimension Reduction Method.

The Induced Dimension Reduction method is a Krylov subspace method for solving sparse (non) symmetric linear systems $Ax = b$. IDR(s) generates residuals in a sequence of nested subspaces. IDR1 IDR2

The dimension of the shadow space can be set by *SetShadowSpace()*. The default size of the shadow space is 4.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*, *GlobalMatrix* or *LocalStencil*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

void **SetShadowSpace**(int s)

Set the size of the Shadow Space.

void **SetRandomSeed**(unsigned long long seed)

Set random seed for ONB creation (seed must be greater than 0)

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **QMRCGStab** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Quasi-Minimal Residual Conjugate Gradient Stabilized Method.

The Quasi-Minimal Residual Conjugate Gradient Stabilized method is a variant of the Krylov subspace *BiCGStab* method for solving sparse (non) symmetric linear systems $Ax = b$. qmrcgstab

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

3.13.1.2 MultiGrid Solvers

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **BaseMultiGrid** : public rocalution::*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Base class for all multigrid solvers Trottenberg2003.

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::BaseAMG< OperatorType, VectorType, ValueType >*, *rocalution::MultiGrid< OperatorType, VectorType, ValueType >*

Public Functions

virtual void **Print**(void) const

Print information about the solver.

void **SetSolver**(*Solver*<*OperatorType*, *VectorType*, *ValueType*> &solver)

Set the coarse grid solver.

void **SetSmoother**(*IterativeLinearSolver*<*OperatorType*, *VectorType*, *ValueType*> **smoother)

Set the smoother for each level.

void **SetSmootherPreIter**(int iter)

Set the number of pre-smoothing steps.

void **SetSmootherPostIter**(int iter)

Set the number of post-smoothing steps.

virtual void **SetRestrictOperator**(*OperatorType* **op) = 0

Set the restriction operator for each level.

virtual void **SetProlongOperator**(*OperatorType* **op) = 0

Set the prolongation operator for each level.

virtual void **SetOperatorHierarchy**(*OperatorType* **op) = 0

Set the operator for each level.

void **SetScaling**(bool scaling)

Enable/disable scaling of intergrid transfers.

void **SetHostLevels**(int levels)

Force computation of coarser levels on the host backend.

void **SetCycle**(unsigned int cycle)

Set the *MultiGrid* Cycle (default: Vcycle)

void **SetKcycleFull**(bool kcycle_full)

Set the *MultiGrid* Kcycle on all levels or only on finest level.

void **InitLevels**(int levels)

Set the depth of the multigrid solver.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **MultiGrid** : public rocalution::*BaseMultiGrid*<*OperatorType*, *VectorType*, *ValueType*>

MultiGrid Method.

The *MultiGrid* method can be used with external data, such as externally computed restriction, prolongation and operator hierarchy. The user need to pass all this information for each level and for its construction. This includes smoothing step, prolongation/restriction, grid traversing and coarse grid solver. This data need to be passed to the solver. Trottenberg2003

- Restriction and prolongation operations can be performed in two ways, based on Restriction() and Prolongation() of the *LocalVector* class, or by matrix-vector multiplication. This is configured by a set function.
- Smoothers can be of any iterative linear solver. Valid options are *Jacobi*, Gauss-Seidel, *ILU*, etc. using a *FixedPoint* iteration scheme with pre-defined number of iterations. The smoothers could also be a solver such as *CG*, *BiCGStab*, etc.
- Coarse grid solver could be of any iterative linear solver type. The class also provides mechanisms to specify, where the coarse grid solver has to be performed, on the host or on the accelerator. The coarse grid solver can be preconditioned.
- Grid scaling based on a L_2 norm ratio.
- *Operator* matrices need to be passed on each grid level.

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **SetRestrictOperator**(*OperatorType* **op)

Set the restriction operator for each level.

virtual void **SetProlongOperator**(*OperatorType* **op)

Set the prolongation operator for each level.

virtual void **SetOperatorHierarchy**(*OperatorType* **op)

Set the operator for each level.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **BaseAMG** : public rocalution::*BaseMultiGrid*<*OperatorType*, *VectorType*, *ValueType*>

Base class for all algebraic multigrid solvers.

The Algebraic *MultiGrid* solver is based on the *BaseMultiGrid* class. The coarsening is obtained by different aggregation techniques. The smoothers can be constructed inside or outside of the class.

All parameters in the Algebraic *MultiGrid* class can be set externally, including smoothers and coarse grid solver.

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::PairwiseAMG< OperatorType, VectorType, ValueType >*, *rocalution::RugeStuebenAMG< OperatorType, VectorType, ValueType >*, *rocalution::SAAMG< OperatorType, VectorType, ValueType >*, *rocalution::UAAMG< OperatorType, VectorType, ValueType >*

Public Functions

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

virtual void **ClearLocal**(void)

Clear all local data.

virtual void **BuildHierarchy**(void)

Create AMG hierarchy.

virtual void **BuildSmoothers**(void)

Create AMG smoothers.

void **SetCoarsestLevel**(int coarse_size)

Set coarsest level for hierarchy creation.

void **SetManualSmoothers**(bool sm_manual)

Set flag to pass smoothers manually for each level.

void **SetManualSolver**(bool s_manual)

Set flag to pass coarse grid solver manually.

void **SetDefaultSmootherFormat**(unsigned int op_format)

Set the smoother operator format.

void **SetOperatorFormat**(unsigned int op_format, int op_blockdim)

Set the operator format.

int **GetNumLevels**(void)

Returns the number of levels in hierarchy.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **UAAMG** : public rocalution::*BaseAMG*<*OperatorType*, *VectorType*, *ValueType*>

Unsmoothed Aggregation Algebraic *MultiGrid* Method.

The Unsmoothed Aggregation Algebraic *MultiGrid* method is based on unsmoothed aggregation based interpolation scheme. stueben

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

void **SetCouplingStrength**(*ValueType* eps)

Set coupling strength.

void **SetOverInterp**(*ValueType* overInterp)

Set over-interpolation parameter for aggregation.

void **SetCoarseningStrategy**(CoarseningStrategy strat)

Set Coarsening strategy.

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **SAAMG** : public rocalution::*BaseAMG*<*OperatorType*, *VectorType*, *ValueType*>

Smoothed Aggregation Algebraic *MultiGrid* Method.

The Smoothed Aggregation Algebraic *MultiGrid* method is based on smoothed aggregation based interpolation scheme. vanek

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

void **SetCouplingStrength**(*ValueType* eps)

Set coupling strength.

void **SetInterpRelax**(*ValueType* relax)

Set the relaxation parameter.

void **SetCoarseningStrategy**(CoarseningStrategy strat)

Set Coarsening strategy.

void **SetLumpingStrategy**(LumpingStrategy lumping_strat)

Set lumping strategy.

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **RugeStuebenAMG** : public rocalution::*BaseAMG*<*OperatorType*, *VectorType*, *ValueType*>

Ruge-Stueben Algebraic *MultiGrid* Method.

The Ruge-Stueben Algebraic *MultiGrid* method is based on the classic Ruge-Stueben coarsening with direct interpolation. The solver provides high-efficiency in terms of complexity of the solver (i.e. number of iterations). However, most of the time it has a higher building step and requires higher memory usage. stueben

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

void **SetStrengthThreshold**(float eps)

Set strength threshold.

void **SetCoarseningStrategy**(CoarseningStrategy strat)

Set Coarsening strategy.

void **SetInterpolationType**(InterpolationType type)

Set Interpolation type.

void **SetInterpolationFF1Limit**(bool FF1)

Enable FF1 interpolation limitation.

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **PairwiseAMG** : public rocalution::*BaseAMG*<*OperatorType*, *VectorType*, *ValueType*>

Pairwise Aggregation Algebraic *MultiGrid* Method.

The Pairwise Aggregation Algebraic *MultiGrid* method is based on a pairwise aggregation matching scheme. It delivers very efficient building phase which is suitable for Poisson-like equation. Most of the time it requires K-cycle for the solving phase to provide low number of iterations. This version has multi-node support. pairwiseamg

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **ClearLocal**(void)

Clear all local data.

void **SetBeta**(*ValueType* beta)

Set beta for pairwise aggregation.

void **SetOrdering**(unsigned int ordering)

Set re-ordering for aggregation.

void **SetCoarseningFactor**(double factor)

Set target coarsening factor.

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

3.13.2 Direct Solvers

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **DirectLinearSolver** : public rocalution::*Solver*<*OperatorType*, *VectorType*, *ValueType*>

Base class for all direct linear solvers.

The library provides three direct methods - *LU*, *QR* and *Inversion* (based on *QR* decomposition). The user can pass a sparse matrix, internally it will be converted to dense and then the selected method will be applied. These methods are not very optimal and due to the fact that the matrix is converted to a dense format, these methods should be used only for very small matrices.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::Inversion< OperatorType, VectorType, ValueType >*, *rocalution::LU< OperatorType, VectorType, ValueType >*, *rocalution::QR< OperatorType, VectorType, ValueType >*

Public Functions

virtual void **Verbose**(int verb = 1)

Provide verbose output of the solver.

- verb = 0 -> no output
- verb = 1 -> print info about the solver (start, end);
- verb = 2 -> print (iter, residual) via iteration control;

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **Inversion** : public rocalution::*DirectLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

Matrix *Inversion*.

Full matrix inversion based on *QR* decomposition.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **LU** : public rocalution::*DirectLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

LU Decomposition.

Lower-Upper Decomposition factors a given square matrix into lower and upper triangular matrix, such that $A = LU$.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **QR** : public rocalution::*DirectLinearSolver*<*OperatorType*, *VectorType*, *ValueType*>

QR Decomposition.

The *QR* Decomposition decomposes a given matrix into $A = QR$, such that Q is an orthogonal matrix and R an upper triangular matrix.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

3.14 Preconditioners

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **Preconditioner** : public rocalution::*Solver*<*OperatorType*, *VectorType*, *ValueType*>

Base class for all preconditioners.

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::AIChebyshev< OperatorType, VectorType, ValueType >*, *rocalution::AS< OperatorType, VectorType, ValueType >*, *rocalution::BlockJacobi< OperatorType, VectorType, ValueType >*, *rocalution::BlockPreconditioner< OperatorType, VectorType, ValueType >*, *rocalution::DiagJacobiSaddlePointPrecond< OperatorType, VectorType, ValueType >*, *rocalution::FSAI< OperatorType, VectorType, ValueType >*, *rocalution::GS< OperatorType, VectorType, ValueType >*, *rocalution::IC< OperatorType, VectorType, ValueType >*, *rocalution::ILU< OperatorType, VectorType, ValueType >*, *rocalution::ILUT< OperatorType, VectorType, ValueType >*, *rocalution::Jacobi< OperatorType, VectorType, ValueType >*, *rocalution::MultiColored< OperatorType, VectorType, ValueType >*, *rocalution::MultiElimination< OperatorType, VectorType, ValueType >*, *rocalution::SGS< OperatorType, VectorType, ValueType >*, *rocalution::SPAI< OperatorType, VectorType, ValueType >*, *rocalution::TNS< OperatorType, VectorType, ValueType >*, *rocalution::VariablePreconditioner< OperatorType, VectorType, ValueType >*

Public Functions

virtual void **SolveZeroSol**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs, setting initial x = 0.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **AIChebyshev** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Approximate Inverse - *Chebyshev Preconditioner*.

The Approximate Inverse - *Chebyshev Preconditioner* is an inverse matrix preconditioner with values from a linear combination of matrix-valued *Chebyshev* polynomials. chebpoly

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

void **Set**(int p, *ValueType* lambda_min, *ValueType* lambda_max)

Set order, min and max eigenvalues.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **FSAI** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Factorized Approximate Inverse *Preconditioner*.

The Factorized Sparse Approximate Inverse preconditioner computes a direct approximation of M^{-1} by minimizing the Frobenius norm $||I - GL||_F$, where L denotes the exact lower triangular part of A and $G := M^{-1}$. The *FSAI* preconditioner is initialized by q, based on the sparsity pattern of $|A^q|$. However, it is also possible to supply external sparsity patterns in form of the *LocalMatrix* class. kolotilina

Note: The *FSAI* preconditioner is only suited for symmetric positive definite matrices.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

void **Set**(int power)

Set the power of the system matrix sparsity pattern.

void **Set**(const *OperatorType* &pattern)

Set an external sparsity pattern.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

void **SetPrecondMatrixFormat**(unsigned int mat_format, int blockdim = 1)

Set the matrix format of the preconditioner.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **SPAI** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

SParse Approximate Inverse *Preconditioner*.

The SParse Approximate Inverse algorithm is an explicitly computed preconditioner for general sparse linear systems. In its current implementation, only the sparsity pattern of the system matrix is supported. The *SPAI* computation is based on the minimization of the Frobenius norm $||AM - I||_F$. grote

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

void **SetPrecondMatrixFormat**(unsigned int mat_format, int blockdim = 1)

Set the matrix format of the preconditioner.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **TNS** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Truncated Neumann Series *Preconditioner*.

The Truncated Neumann Series (*TNS*) preconditioner is based on $M^{-1} = K^T D^{-1} K$, where $K = (I - LD^{-1} + (LD^{-1})^2)$, with the diagonal D of A and the strictly lower triangular part L of A. The preconditioner can be computed in two forms - explicitly and implicitly. In the explicit form, the full construction of M is performed via matrix-matrix operations, whereas in the implicit from, the application of the preconditioner is based on matrix-vector operations only. The matrix format for the stored matrices can be specified.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

void **Set**(bool imp)

Set implicit (true) or explicit (false) computation.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

void **SetPrecondMatrixFormat**(unsigned int mat_format, int blockdim = 1)

Set the matrix format of the preconditioner.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **AS** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Additive Schwarz *Preconditioner*.

The Additive Schwarz preconditioner relies on a preconditioning technique, where the linear system $Ax = b$ can be decomposed into small sub-problems based on $A_i = R_i^T A R_i$, where R_i are restriction operators. Those restriction operators produce sub-matrices wich overlap. This leads to contributions from two preconditioners on the overlapped area which are scaled by $1/2$. RAS

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::RAS< OperatorType, VectorType, ValueType >*

Public Functions

virtual void **Print**(void) const

Print information about the solver.

void **Set**(int nb, int overlap, *Solver*<*OperatorType*, *VectorType*, *ValueType*> **preconds)

Set number of blocks, overlap and array of preconditioners.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **RAS** : public rocalution::*AS*<*OperatorType*, *VectorType*, *ValueType*>

Restricted Additive Schwarz *Preconditioner*.

The Restricted Additive Schwarz preconditioner relies on a preconditioning technique, where the linear system $Ax = b$ can be decomposed into small sub-problems based on $A_i = R_i^T A R_i$, where R_i are restriction operators. The *RAS* method is a mixture of block *Jacobi* and the *AS* scheme. In this case, the sub-matrices contain overlapped areas from other blocks, too. RAS

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **BlockJacobi** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Block-Jacobi *Preconditioner*.

The Block-Jacobi preconditioner is designed to wrap any local preconditioner and apply it in a global block fashion locally on each interior matrix.

Template Parameters

- **OperatorType** – - can be *GlobalMatrix*
- **VectorType** – - can be *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

void **Set**(*Solver*<*LocalMatrix*<*ValueType*>, *LocalVector*<*ValueType*>, *ValueType*> &precond)

Set local preconditioner.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

virtual void **SolveZeroSol**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs, setting initial x = 0.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **BlockPreconditioner** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Block-Preconditioner.

When handling vector fields, typically one can try to use different preconditioners and/or solvers for the different blocks. For such problems, the library provides a block-type preconditioner. This preconditioner builds the following block-type matrix

$$P = \begin{pmatrix} A_d & 0 & . & 0 \\ B_1 & B_d & . & 0 \\ . & . & . & . \\ Z_1 & Z_2 & . & Z_d \end{pmatrix}$$

The solution of P can be performed in two ways. It can be solved by block-lower-triangular sweeps with inversion of the blocks $A_d \dots Z_d$ and with a multiplication of the corresponding blocks. This is set by *SetLSolver()* (which is the default solution scheme). Alternatively, it can be used only with an inverse of the diagonal $A_d \dots Z_d$ (Block-Jacobi type) by using *SetDiagonalSolver()*.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Clear**(void)

Clear (free all local data) the solver.

void **Set**(int n, const int *size, *Solver*<*OperatorType*, *VectorType*, *ValueType*> **D_solver)

Set number, size and diagonal solver.

void **SetDiagonalSolver**(void)

Set diagonal solver mode.

void **SetLSolver**(void)

Set lower triangular sweep mode.

void **SetExternalLastMatrix**(const *OperatorType* &mat)

Set external last block matrix.

virtual void **SetPermutation**(const *LocalVector*<int> &perm)

Set permutation vector.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **Jacobi** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Jacobi Method.

The *Jacobi* method is for solving a diagonally dominant system of linear equations $Ax = b$. It solves for each diagonal element iteratively until convergence, such that

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i}^{n} a_{ij}x_j^{(k)}\right)$$

Template Parameters

- **OperatorType** – - can be *LocalMatrix* or *GlobalMatrix*
- **VectorType** – - can be *LocalVector* or *GlobalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

virtual void **ResetOperator**(const *OperatorType* &op)

Reset the operator; see *ReBuildNumeric()*

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **GS** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Gauss-Seidel / Successive Over-Relaxation Method.

The Gauss-Seidel / SOR method is for solving system of linear equations $Ax = b$. It approximates the solution iteratively with

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i}^{n} a_{ij}x_j^{(k)}\right),$$

with $\omega \in (0, 2)$.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

virtual void **ResetOperator**(const *OperatorType* &op)

Reset the operator; see *ReBuildNumeric()*

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **SGS** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Symmetric Gauss-Seidel / Symmetric Successive Over-Relaxation Method.

The Symmetric Gauss-Seidel / SSOR method is for solving system of linear equations $Ax = b$. It approximates the solution iteratively.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*

- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

virtual void **ResetOperator**(const *OperatorType* &op)

Reset the operator; see *ReBuildNumeric()*

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **ILU** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Incomplete *LU* Factorization based on levels.

The Incomplete *LU* Factorization based on levels computes a sparse lower and sparse upper triangular matrix such that $A = LU - R$.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

virtual void **Set**(int p, bool level = true)

Initialize ILU(p) factorization.

Initialize ILU(p) factorization based on power. SAAD

- level = true build the structure based on levels
- level = false build the structure only based on the power(p+1)

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **ILUT** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Incomplete *LU* Factorization based on threshold.

The Incomplete *LU* Factorization based on threshold computes a sparse lower and sparse upper triangular matrix such that $A = LU - R$. Fill-in values are dropped depending on a threshold and number of maximal fill-ins per row. SAAD

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

virtual void **Set**(double t)

Set drop-off threshold.

virtual void **Set**(double t, int maxrow)

Set drop-off threshold and maximum fill-ins per row.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **IC** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Incomplete Cholesky Factorization without fill-ins.

The Incomplete Cholesky Factorization computes a sparse lower triangular matrix such that $A = LL^T - R$. Additional fill-ins are dropped and the sparsity pattern of the original matrix is preserved.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **VariablePreconditioner** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Variable *Preconditioner*.

The Variable *Preconditioner* can hold a selection of preconditioners. Thus, any type of preconditioners can be combined. As example, the variable preconditioner can combine *Jacobi*, *GS* and *ILU* - then, the first iteration of the iterative solver will apply *Jacobi*, the second iteration will apply *GS* and the third iteration will apply *ILU*. After that, the solver will start again with *Jacobi*, *GS*, *ILU*.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Clear**(void)

Clear (free all local data) the solver.

virtual void **SetPreconditioner**(int n, *Solver*<*OperatorType*, *VectorType*, *ValueType*> **precond)

Set the preconditioner sequence.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **MultiColored** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Base class for all multi-colored preconditioners.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*

- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::MultiColoredILU< OperatorType, VectorType, ValueType >*, *rocalution::MultiColoredSGS< OperatorType, VectorType, ValueType >*

Public Functions

virtual void **Clear**(void)

Clear (free all local data) the solver.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

void **SetPrecondMatrixFormat**(unsigned int mat_format, int blockdim = 1)

Set a specific matrix type of the decomposed block matrices.

void **SetDecomposition**(bool decomp)

Set if the preconditioner should be decomposed or not.

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **MultiColoredSGS** : public rocalution::*MultiColored*<*OperatorType*, *VectorType*, *ValueType*>

Multi-Colored Symmetric Gauss-Seidel / SSOR *Preconditioner*.

The Multi-Colored Symmetric Gauss-Seidel / SSOR preconditioner is based on the splitting of the original matrix. Higher parallelism in solving the forward and backward substitution is obtained by performing a multi-colored decomposition. Details on the Symmetric Gauss-Seidel / SSOR algorithm can be found in the *SGS* preconditioner.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Subclassed by *rocalution::MultiColoredGS< OperatorType, VectorType, ValueType >*

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

void **SetRelaxation**(*ValueType* omega)

Set the relaxation parameter for the SOR/SSOR scheme.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **MultiColoredGS** : public rocalution::*MultiColoredSGS*<*OperatorType*, *VectorType*, *ValueType*>

Multi-Colored Gauss-Seidel / SOR *Preconditioner*.

The Multi-Colored Symmetric Gauss-Seidel / SOR preconditioner is based on the splitting of the original matrix. Higher parallelism in solving the forward substitution is obtained by performing a multi-colored decomposition. Details on the Gauss-Seidel / SOR algorithm can be found in the *GS* preconditioner.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **MultiColoredILU** : public rocalution::*MultiColored*<*OperatorType*, *VectorType*, *ValueType*>

Multi-Colored Incomplete *LU* Factorization *Preconditioner*.

Multi-Colored Incomplete *LU* Factorization based on the ILU(p) factorization with a power(q)-pattern method. This method provides a higher degree of parallelism of forward and backward substitution compared to the standard ILU(p) preconditioner. Lukarski2012

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **ReBuildNumeric**(void)

Rebuild the solver only with numerical computation (no allocation or data structure computation)

void **Set**(int p)

Initialize a multi-colored *ILU*(p, p+1) preconditioner.

void **Set**(int p, int q, bool level = true)

Initialize a multi-colored ILU(p, q) preconditioner.

level = true will perform the factorization with levels

level = false will perform the factorization only on the power(q)-pattern

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **MultiElimination** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Multi-Elimination Incomplete *LU* Factorization *Preconditioner*.

The Multi-Elimination Incomplete *LU* preconditioner is based on the following decomposition

$$A = \begin{pmatrix} D & F \\ E & C \end{pmatrix} = \begin{pmatrix} I & 0 \\ ED^{-1} & I \end{pmatrix} \times \begin{pmatrix} D & F \\ 0 & \hat{A} \end{pmatrix},$$

where $\hat{A} = C - ED^{-1}F$. To make the inversion of D easier, we permute the preconditioning before the factorization with a permutation P to obtain only diagonal elements in D. The permutation here is based on a maximal independent set. This procedure can be applied to the block matrix $\hat{A}$, in this way we can perform the factorization recursively. In the last level of the recursion, we need to provide a solution procedure. By the design of the library, this can be any kind of solver. SAAD

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

inline int **GetSizeDiagBlock**(void) const

Returns the size of the first (diagonal) block of the preconditioner.

inline int **GetLevel**(void) const

Return the depth of the current level.

virtual void **Print**(void) const

Print information about the solver.

virtual void **Clear**(void)

Clear (free all local data) the solver.

void **Set**(*Solver*<*OperatorType*, *VectorType*, *ValueType*> &AA_Solver, int level, double drop_off = 0.0)

Initialize (recursively) ME-ILU with level (depth of recursion)

AA_Solvers - defines the last-block solver

drop_off - defines drop-off tolerance

void **SetPrecondMatrixFormat**(unsigned int mat_format, int blockdim = 1)

Set a specific matrix type of the decomposed block matrices.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

template<class **OperatorType**, class **VectorType**, typename **ValueType**>

class **DiagJacobiSaddlePointPrecond** : public rocalution::*Preconditioner*<*OperatorType*, *VectorType*, *ValueType*>

Diagonal *Preconditioner* for Saddle-Point Problems.

Consider the following saddle-point problem

$$A = \begin{pmatrix} K & F \\ E & 0 \end{pmatrix}.$$

For such problems we can construct a diagonal Jacobi-type preconditioner of type

$$P = \begin{pmatrix} K & 0 \\ 0 & S \end{pmatrix},$$

with $S = ED^{-1}F$, where D are the diagonal elements of K. The matrix S is fully constructed (via sparse matrix-matrix multiplication). The preconditioner needs to be initialized with two external solvers/preconditioners - one for the matrix K and one for the matrix S.

Template Parameters

- **OperatorType** – - can be *LocalMatrix*
- **VectorType** – - can be *LocalVector*
- **ValueType** – - can be float, double, std::complex<float> or std::complex<double>

Public Functions

virtual void **Print**(void) const

Print information about the solver.

virtual void **Clear**(void)

Clear (free all local data) the solver.

void **Set**(*Solver*<*OperatorType*, *VectorType*, *ValueType*> &K_Solver, *Solver*<*OperatorType*, *VectorType*, *ValueType*> &S_Solver)

Initialize solver for K and S.

virtual void **Build**(void)

Build the solver (data allocation, structure and numerical computation)

virtual void **Solve**(const *VectorType* &rhs, *VectorType* *x)

Solve *Operator* x = rhs.

INDEX

R