Htool-DDM

Release 1.0.0

P. Marchand & P.-H. Tournier

Mar 25, 2025

OVERVIEW

Htool-DDM is a lightweight header-only C++14 library that provides an easy-to-use interface for parallel iterative solvers and a default matrix compression via in-house hierarchical matrix implementation. Its goal is to provide modern iterative solvers for dense/compressed linear systems.

It is also an extensible framework which contains several customization points. For example, one can provide its own compression algorithm, or customize the default hierarchical compression. Via its interface with HPDDM, it is also a flexible tool to test various iterative solvers and preconditioners.

The project is hosted on GitHub, under the permissive MIT license.

Projects including Htool-DDM

Htool-DDM provides distributed solvers and black-box hierarchical compression. It can be used directly in C++, via its Python interface, or in the following projects:

- FreeFEM to compress matrices stemming from the discretisation of boundary integral equations and iterative solvers,
- PETSc for black-box compression and iterative solvers.

License

Authors

If you need help or have questions regarding Htool, feel free to contact the main developers or to leave a report on our GitHub issue tracker!

Developers:

- Pierre Marchand
- Pierre-Henri Tournier

Contributors/Collaborators:

- Xavier Claeys
- Pierre Jolivet
- Frédéric Nataf

Acknowledgements

- Centre Inria de Saclay - Île-de-France, France
- ANR NonlocalDD, (grant ANR-15-CE23-0017-01), France
- Centre Inria de Paris, France
- Laboratoire Jacques-Louis Lions,Paris, France

OVERVIEW

CHAPTER

ONE

WHY USING HTOOL-DDM?

Htool-DDM aims to provide iterative solvers with preconditioners stemming from **domain decomposition methods** (DDM). It uses matrix compression via **hierarchical matrices** by default, and in particular, it provides

- parallel matrix-vector and matrix-matrix product using MPI and OpenMP,
- iterative solvers via HPDDM,
- preconditioning techniques using domain decomposition methods.

Why?

The storage and cost of assembly for dense matrices are both quadratic with respect to their size, while the cost of using linear solvers is cubic for direct solvers. For iterative solvers, the matrix-vector product has quadratic complexity, which is to be multiplied by the number of iterations. To reduce these costs, several compression techniques have been developed, which gives approximated representation of matrix-vector product and other operations. Htool-DDM provides default compression via sec_hierarchical_matrices, and emphasize has been put on

- parallelisation for **high-performance computing**,
- a **black box** interface, to tackle a great variety of problems.

Applications

Hierarchical matrices are generally used to compress matrices stemming from the discretisation of asymptotically smooth kernels $\kappa(x, y)$, i.e, for two cluster of geometric points X and Y,

$$|\partial_x^\alpha \partial_y^\beta \kappa(x,y)| \leq C_{\mathrm{as}} |x-y|^{-|\alpha|-|\beta|-s}.$$

with $x \in X$, $y \in Y$, $x \neq y$, $\alpha, \beta \in \mathbb{N}_0^d$ and $\alpha + \beta \neq 0$.

Such matrices arise in the context of

- discretisation of boundary integral equations[1],
- solving Lyapunov and Riccati equations[1],
- discretization of the integral Fractional laplacian[2],
- kernel-based scattered data interpolation[3].

[1] Steffen Börm, Lars Grasedyck, and Wolfgang Hackbusch. Introduction to hierarchical matrices with applications. *Engineering Analysis with Boundary Elements*, 27(5):405–422, 2003.

[2] Mark Ainsworth and Christian Glusa. Towards an efficient finite element method for the integral fractional laplacian on polygonal domains. In *Contemporary Computational Mathematics - A Celebration of the 80th Birthday of Ian Sloan*, pages 17–57. Springer International Publishing, 2018. doi:10.1007/978-3-319-72456-0_2.

[3] Armin Iske, Sabine Le Borne, and Michael Wende. Hierarchical matrix approximation for kernel-based scattered data interpolation. *SIAM Journal on Scientific Computing*, 39(5):A2287–A2316, jan 2017. doi:10.1137/16m1101167.

CHAPTER

TWO

HIERARCHICAL MATRICES

We give here a quick overview on hierarchical matrices, and we refer to Hackbusch[1], Bebendorf[2], Börm, Grasedyck, and Hackbusch[3] for a detailed presentation. The presentation and figures are taken from Marchand[4].

2.1 Low-rank compression

2.1.1 Low-rank matrix

Let $\mathbf{B} \in \mathbb{C}^{M\times N}$ be a low-rank matrix, i.e, there exist $\mathbf{u}_j \in \mathbb{C}^M$, $\mathbf{v}_j \in \mathbb{C}^N$ with $1 \leq j \leq r \leq \min(M, N)$ such that

$$\mathbf{B} = \sum_{j=1}^{r} \mathbf{u}_j \mathbf{v}_j^T .$$

Note

Remark that, using the previous expression of $\mathbf{B}$, storage cost and complexity of matrix vector product is $(M+N)r$, which is lower than for the usual dense format as soon as $r < \frac{MN}{M+N}$.

2.1.2 Low-rank approximation

Usually, matrices of interest are not low-rank, but they may be well-approximated by low-rank matrices. To build such approximation, one can use a *truncated Singular Value Decomposition* (SVD):

$$\mathbf{B}^{(r)} = \sum_{j=1}^{r} \sigma_j \mathbf{u}_j \mathbf{v}_j^T ,$$

where $(\sigma_j)_{j=1}^r$ are the singular values of $\mathbf{B}$ in decreasing order. Then, the approximation is

$$\|\mathbf{B} - \mathbf{B}^{(r)}\|_2^2 = \sigma_{r+1}^2 \quad \text{and} \quad \|\mathbf{B} - \mathbf{B}^{(r)}\|_{\mathrm{F}}^2 = \sum_{j=r+1}^{n} \sigma_j^2 .$$

In conclusion, a matrix with sufficiently decreasing singular values can be well-approximated by a low-rank approximation.

[1] Wolfgang Hackbusch. *Hierarchical Matrices: Algorithms and Analysis*. Volume 49 of Springer Series in Computational Mathematics. Springer-Verlag, Berlin, 2015. doi:10.1007/978-3-662-47324-5.

[2] Mario Bebendorf. *Hierarchical matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems*. Volume 63 of Lecture Notes in Computational Science and Engineering. Springer-Verlag, Berlin, 2008.

[3] Steffen Börm, Lars Grasedyck, and Wolfgang Hackbusch. Introduction to hierarchical matrices with applications. *Engineering Analysis with Boundary Elements*, 27(5):405–422, 2003.

[4] Pierre Marchand. *Schwarz methods and boundary integral equations*. Theses, Sorbonne Université, January 2020. URL: https://hal.archives-ouvertes.fr/tel-02922455.

Note

A truncated SVD is actually the best low-rank approximation possible (see Eckart-Young-Mirsky theorem).

Warning

Computing a SVD requires the whole matrix, which would require storing all the coefficients, and this is exactly what we want to avoid. There exist several techniques to approximate a truncated SVD computing only some coefficients of the initial matrix, for example, Adaptive Cross Approximation (ACA) or randomized SVD.

2.2 Hierarchical clustering

Generally, matrices of interest are not low-rank matrices, and do not have rapidly decreasing singular values. But they can have sub-blocks with rapidly decreasing singular values. In particular, this is the case of matrices stemming from the discretisation of asymptotically smooth kernel, $\kappa(x, y)$, i.e, for two clusters of geometric points X and Y,

$$|\partial_x^\alpha \partial_y^\beta \kappa(x, y)| \leq C_{\text{as}} |x - y|^{-|\alpha|-|\beta|-s}.$$

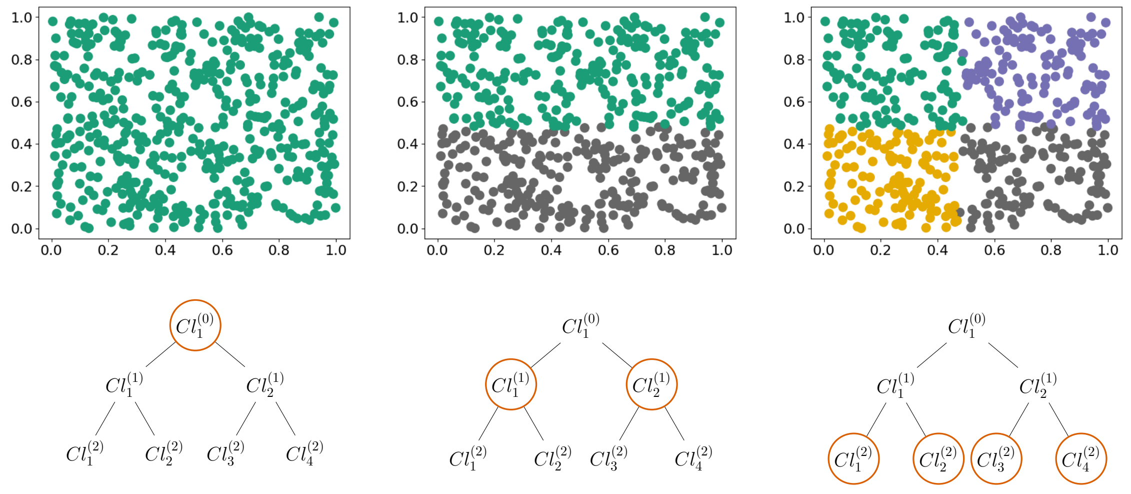
In this case, sub-blocks corresponding to the interaction between two clusters X and Y satisfying an *admissibility condition*,

$$\max(\operatorname{diam}(X), \operatorname{diam}(Y)) \leq \eta \operatorname{dist}(X, Y),$$

have exponentially decreasing singular values. Thus, they can be well-approximated by low-rank matrices.

2.2.1 Geometric clustering

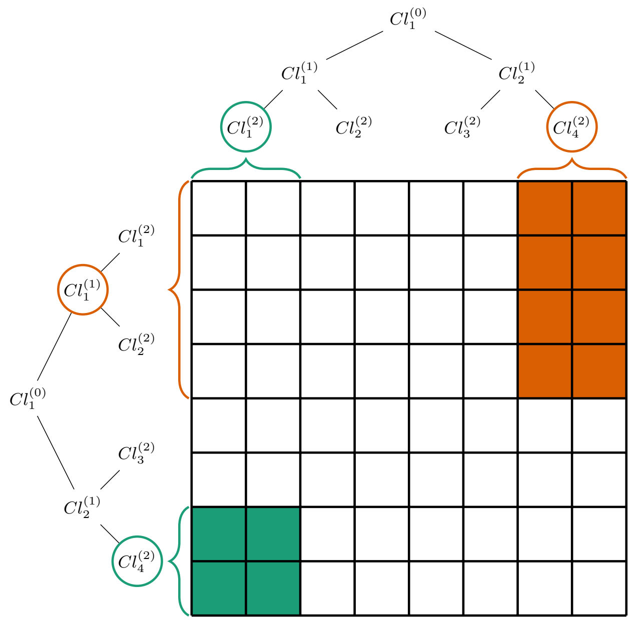
To identify sub-blocks satisfying the admissibility condition, a hierarchical partition of the geometry is introduced via a *cluster tree*. Several strategies can be adopted to produce such partitions. A simple example is to partition the geometry by dividing recursively the set of points into two, for example,



where Cl_i^j is the i th cluster at level j, and each figure shows one level of the cluster tree.

2.2.2 Block cluster tree

The geometric clustering described previously defines a block cluster tree as described in the following figure.



Then, hierarchical matrices are built traversing the block cluster tree starting from its root, and

- If the current block satisfies the admissibility condition, we approximate it using a low-rank matrix.
- If the current block does not satisfy the admissibility condition
  - If it is leaf, we compute the block as a dense matrix,
  - If it is not a leaf, we check the children of the current block.

CHAPTER

THREE

DOMAIN DECOMPOSITION SOLVERS

This is a quick introduction to domain decomposition solvers for dense/compressed systems, and how they are implemented in Htool-DDM. These techniques have been introduced in Hebeker[1] and analysed in papers like Stephan and Tran[2], Heuer[3] for application to boundary integral equations. One goal of this library is to provide DD solvers with a GenEO coarse space Marchand *et al.*[4], Marchand[5].

3.1 Iterative solvers

For a given linear system $\mathbf{A} \in \mathbb{C}^{n\times n}$, we want to solve

$$\mathbf{A}\mathbf{x} = \mathbf{b},$$

where

- $\mathbf{b}$ is a given right-hand side stemming from the data of the underlying problem, and,
- $\mathbf{x}$ is the unknown to be computed.

There are mainly two strategies to solve a linear system:

- A *direct* linear solver, which typically relies on a factorisation of $\mathbf{A}$.
- An *iterative* linear solver, which requires multiples matrix-vector products for a given precision.

This library focuses on the second approach because it is easier to adapt such algorithms in a distributed-memory context. The issue is that the number of matrix-vector products can be large, making the method less efficient. Thus, a preconditioner $\mathbf{P} \in \mathbb{C}^{n\times n}$ is often used to solve the alternative linear system

$$\mathbf{P}\mathbf{A}\mathbf{x} = \mathbf{P}\mathbf{b},$$

with the aim of making the preconditioned linear system $\mathbf{P}\mathbf{A}$ more suitable for iterative resolution.

[1] Friedrich K. Hebeker. On a parallel schwarz algorithm for symmetric strongly elliptic integral equations. In Roland Glowinski, Yuri A. Kuznetsov, Gérard Meurant, Jacques Périaux, and Olaf B. Widlund, editors, *Domain Decomposition Methods for Partial Differential Equations*, 382–393. SIAM, 1990.

[2] Ersnt P. Stephan and Thanh Tran. Domain decomposition algorithms for indefinite hypersingular integral equations: the h and p versions. *SIAM Journal on Scientific Computing*, 19(4):1139–1153, 1998.

[3] Norbert Heuer. Efficient algorithms for the \$p\$-version of the boundary element method. *Journal of Integral Equations and Applications*, 8(3):337–360, September 1996. doi:10.1216/jiea/1181075956.

[4] Pierre Marchand, Xavier Claeys, Pierre Jolivet, Frédéric Nataf, and Pierre-Henri Tournier. Two-level preconditioning for h-version boundary element approximation of hypersingular operator with GenEO. *Numerische Mathematik*, 146(3):597–628, sep 2020. doi:10.1007/s00211-020-01149-5.

[5] Pierre Marchand. *Schwarz methods and boundary integral equations*. Theses, Sorbonne Université, January 2020. URL: https://hal.archives-ouvertes.fr/tel-02922455.

3.2 Schwarz preconditioners

A specific class of preconditionners stemming from Domain Decomposition Methods (DDM) are *Schwarz preconditioners*. They rely on a decomposition with overlap of our set of unknowns in N subdomains. Each subdomain defines a local numbering with $\sigma_p : \{1, \dots, n_p\} \to \{1, ..., n\}$, where n_p is the number of unknows in the p^{th} subdomain. Thus, the restriction matrix can be defined as $\mathbf{R}_p \in \mathbb{R}^{n_p \times n}$

$$(\mathbf{R})_{j,k} = \begin{cases} 1 & \text{if } k = \sigma_p(j), \\ 0 & \text{otherwise.} \end{cases}$$

Its transpose $\mathbf{R}_p^T$ defines the extension by zero, and $\mathbf{R}_p\mathbf{A}\mathbf{R}_p^T$ is the diagonal block of the original linear system associated with the p^{th} subdomain. An example of one-level Schwarz preconditioner can then be defined as

$$\mathbf{P}_{\text{ASM}} = \sum_{p=1}^{N} \mathbf{R}_p^T(\mathbf{R}_p\mathbf{A}\mathbf{R}_p^T)^{-1}\mathbf{R}_p,$$

which is called a *Additive Schwarz Method* (ASM) preconditionner.

Note

It lends itself well in a distributed-memory context because it relies on solving local problem independently. Typically, a subdomain will be associated with one MPI process.

3.3 GenEO coarse space

If Schwarz preconditioners, and DD preconditioners in general, improve iterative solving of linear systems (by lowering the number of required matrix-vector products), their efficiency decrease when the number of subdomains (and thus MPI processes) increases.

To make those preconditionners more robust when N increases, a recurring technique is to add a small *coarse space*. It defines the columns of $\mathbf{R}_0^T$ so that $\mathbf{R}_0\mathbf{A}\mathbf{R}_0^T$ is a global matrix of small size. Then, a coarse space can be used additively with $\mathbf{P}_{\text{ASM}}$:

$$\mathbf{P}_{\text{ASM},2} = \mathbf{R}_0^T(\mathbf{R}_0\mathbf{A}\mathbf{R}_0^T)^{-1}\mathbf{R}_0 + \sum_{p=1}^{N} \mathbf{R}_p^T(\mathbf{R}_p\mathbf{A}\mathbf{R}_p^T)^{-1}\mathbf{R}_p.$$

It is also called a two-level Schwarz preconditioner because we solve

1. local problems to the subdomains: $\mathbf{R}_p\mathbf{A}\mathbf{R}_p^T$ and,
2. a global problem $\mathbf{R}_0\mathbf{A}\mathbf{R}_0^T$ (but hopefully of small size).

Various definitions of coarse spaces have been introduced, they usually rely on a coarser discretization of the underlying problem, or on well-chosen local eigenproblems. This library focuses on the latter with the so-called "Generalized Eigenproblems in the Overlap" (GenEO) coarse space, where local eigenproblems are of the form

$$\mathbf{D}_p\mathbf{R}_p\mathbf{A}\mathbf{R}_p^T\mathbf{D}_p\mathbf{v}_p = \lambda_p\mathbf{B}_p\mathbf{v}_p,$$

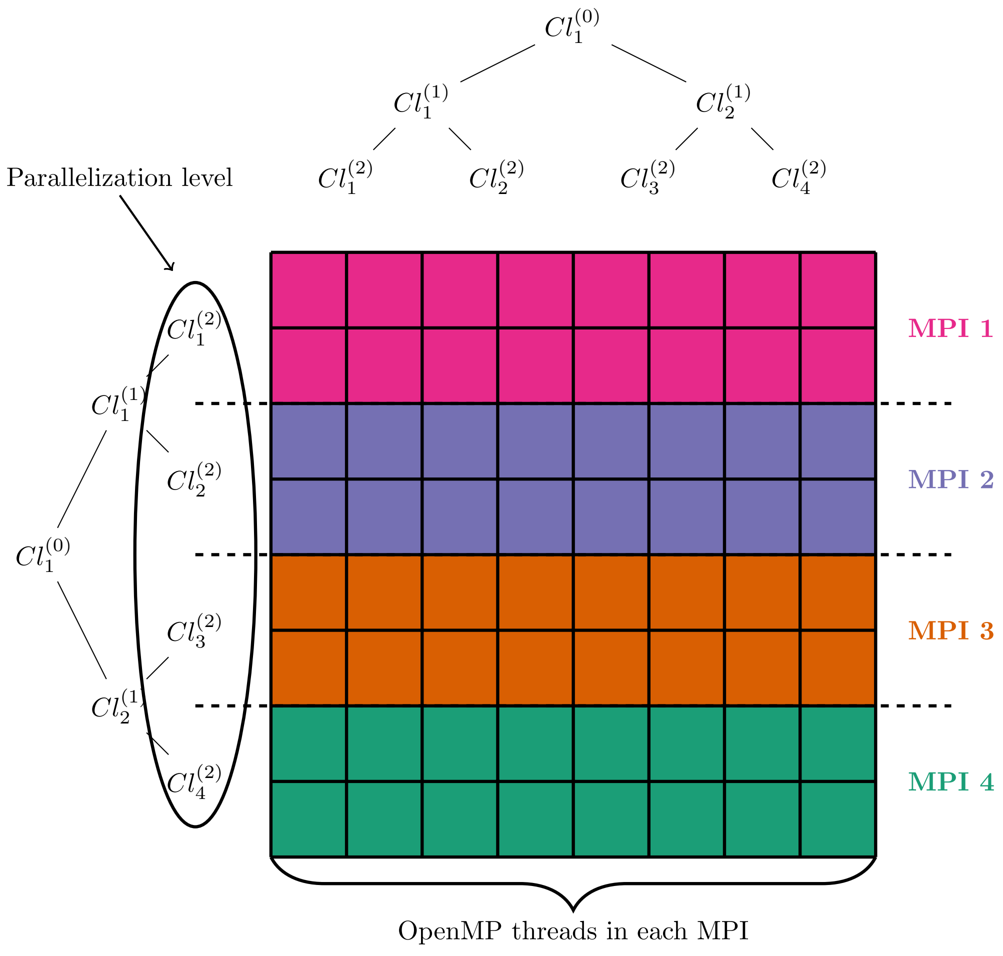
where

- $\mathbf{D}_p$ defines a partition of the unity and,
- $\mathbf{B}_p$ is well-chosen matrix depending on the underlying problem.

We refer Marchand *et al.*[Page 11, 4], Marchand[Page 11, 5] where such coarse space is analysed in the context of boundary integral equations.

3.4 Row-wise distributed operator

We focused so far on the preconditionner $\mathbf{P}$, but the actual linear system $\mathbf{A}$ we want to solve also needs to be parallelized. We use a simple row-wise data-layout so that applying the preconditionner is relatively easy:



The distribution of the linear system is defined via a partition induced by a level of the target cluster tree.

CHAPTER

FOUR

GETTING STARTED

Htool can be used directly via

- its *C++ API*
- its Python API.

When used via another library or software, we refer to its own documentation and examples:

Library/Software	Version	Documentation	Examples
FreeFEM	≥ 4.5	documentation	2D and 3D BEM with PETSc and without
PETSc	≥ 3.16	documentation	ex82

CHAPTER FIVE

QUICKSTART

Htool-DDM can be used for different use cases:

- *Geometric clustering*
- *Hierarchical compression*
- *DDM Solver*

5.1 Dependencies

Htool-DDM is a C++ header-only library, it requires:

- C++ compiler with standard 14.
- BLAS, to perform algebraic dense operations.

And

- For DDM solvers: a MPI implementation, HPDDM and LAPACK.
- For shared memory parallelism when using in-house $\mathcal{H}$-matrix algebra: OpenMP
- for SVD (re)compression: LAPACK
- CMake to generate a build-system for the examples and tests.

5.2 Installation

It is sufficient to include the `include` folder of this repository in your library. If you prefer, the following command copies the `include` folder your OS-specific include directory to make it more widely available: in the root of this repository on your system,

```
cmake -B build
cmake --build build --config Release --target install -- -j $(nproc)
```

Note

The *install* prefix can be modified using `cmake .. -DCMAKE_INSTALL_PREFIX:PATH=your/install/path` instead of the first line.

Once installed, you can also use Htool-DDM as a cmake imported target as follows in your `CMakeLists.txt`:

```
find_package(Htool REQUIRED)
find_package(MPI REQUIRED)
find_package(BLAS REQUIRED)
# other optional packages

add_executable(your_target your_file.cpp)
target_link_libraries(your_target Htool::htool)
```

5.3 Geometric clustering

The hierarchical clustering of a geometry is contained in *htool::Cluster* and can be created using a *htool::ClusterTreeBuilder*. For example,

```
int number_points = 10000;
int spatial_dimension = 3;
int number_of_partitions = 2;
int number_of_children = 2;
std::vector<double> coordinates(number_points*spatial_dimension);
// coordinates = {...}
htool::ClusterTreeBuilder<double> cluster_tree_builder;
htool::Cluster<double> cluster = cluster_tree_builder.create_cluster_tree(number_points,␣
↪spatial_dimension, coordinates.data(), number_of_children, number_of_partitions);
```

where

- `number_of_partitions` defines the number of children at a level of the cluster tree called "partition", which can be used to distribute data in a MPI context.
- `number_of_children` defines the number of children for the other nodes that are not leaves.

5.4 Hierarchical compression

To use the in-house hierarchical compression *htool::HMatrix*, Htool-DDM will need two inputs:

1. The underlying geometry of the kernel to be compressed for the *Geometric clustering*.
2. A function to generate any coefficient of the matrix to be compressed.

5.4.1 Coefficient generator

A coefficient generator is defined by following the interface *htool::VirtualGenerator*, and in particular *htool::VirtualGenerator::copy_submatrix()*. For example:

```
class UserOperator: public htool::VirtualGenerator<double>{
    virtual void copy_submatrix(int M, int N, const int *rows, const int *cols, T *ptr)␣
↪const override {
        for (int i = 0; i < M; i++) {
            for (int j = 0; j < N; j++) {
                ptr[i + M * j] = ...// A(rows[i], cols[j]) where A is the kernel to be␣
↪compressed;
            }
        }
    }
};
```

5.4.2 Build a hierarchical matrix

To build a *htool::HMatrix*, a *htool::HMatrixBuilder* can be used:

- Its constructor *htool::HMatrixBuilder::HMatrixBuilder()* takes at least one geometry.
- *htool::HMatrixBuilder::build()* generates a *htool::HMatrix* from a *Coefficient generator*, and a *htool::HMatrixTreeBuilder* object containing all the parameters related to compression.

```
int number_points = 10000;
int spatial_dimension = 3;
double epsilon = 1e-3;
double eta = 10;
char symmetry = 'N';
char uplo = 'N';
std::vector<double> coordinates(number_points*spatial_dimension);
// coordinates = {...}
htool::HMatrixBuilder<double> hmatrix_builder(number_points, spatial_dimension,␣
↪coordinates.data());
htool::HMatrix<double> hmatrix = hmatrix_builder.build(UserOperator(),
↪htool::HMatrixTreeBuilder<double>(epsilon, eta, symmetry, uplo));
```

Note

The geometric clustering is done within the constructor of *htool::HMatrixBuilder*. You can still access the resulting target and source clusters as public members of `hmatrix_builder`.

5.4.3 Use a hierarchical matrix

Basic linear algebra is provided for *htool::HMatrix*. Shared-memory parallelism is also supported via execution policy traits.

Table 5.4.1: Supported linear algebra

Operations	C++ function	Supported execution policy
$\mathcal{H}$-matrix vector product	*htool::add_hmatrix_vector_product()*	std::execution::seq, std::execution::par
$\mathcal{H}$-matrix matrix product	*htool::add_hmatrix_matrix_product()*	std::execution::seq, std::execution::par
$\mathcal{H}$-LU factorisation	*htool::lu_factorization()*	None
$\mathcal{H}$-LU solve	*htool::lu_solve()*	None
$\mathcal{H}$-Cholesky factorisation	*htool::cholesky_factorization()*	None
$\mathcal{H}$-Cholesky solve	*htool::cholesky_solve()*	None

Note

We try to have a similar API to BLAS/LAPACK or <linalg>. In particular, parallel version of linear algebra functions are available using execution policy traits. If none is given, it defaults to sequential operation.

5.5 DDM Solver

5.5.1 Distributed operator

To assemble a distributed operator, first a *geometric clustering* needs to be applied. The partition defined by the target geometric cluster tree is then used to define a row-wise distributed operator (see *Row-wise distributed operator*). `htool::DefaultApproximationBuilder` builds a row-wise distributed operator `htool::DistributedOperator` where each block of rows is compressed using a `htool::HMatrix`, which can be accessed as a public member.

```
htool::HMatrixTreeBuilder<double> hmatrix_tree_builder(epsilon, eta, symmetry, uplo)
htool::DefaultApproximationBuilder<double> distributed_operator_builder(UserOperator(),␣
↪target_cluster, source_cluster, hmatrix_tree_builder,MPI_COMM_WORLD);
DistributedOperator<double> &distributed_operator = distributed_operator_builder.
↪distributed_operator;
```

A `htool::DistributedOperator` object provides distributed products with matrices and vectors.

5.5.2 Linear solver

The iterative linear solve will be done via a `htool::DDM` object, which can be built using `htool::DDMSolverBuilder`. We give here an example for a simple block-Jacobi solver, in particular we use `htool::DefaultApproximationBuilder::block_diagonal_hmatrix`, which is a pointer to the $\mathcal{H}$ matrix for the block diagonal associated with the local subdomain.

```
// Create DDM object
htool::DDMSolverBuilder<double> ddm_solver_build(distributed_operator,distributed_
↪operator_builder.block_diagonal_hmatrix);
htool::DDM<double> solver = ddm_solver_build.solver;

// Prepare solver
HPDDM::Option &opt = *HPDDM::Option::get();
opt.parse("-hpddm_schwarz_method asm ");
ddm_with_overlap.facto_one_level();

// Solve
int mu = 1 // number of right-hand side
std::vector<double> x = ...// unknowns
std::vector<double> b = ...// right-hand sides
ddm_with_overlap.solve(b.data(), x.data(), mu);
```

Note

We rely on the external library HPDDM for the implementation of efficient iterative solvers. We refer to its cheat-sheet listing the various possible options for the solver.

CHAPTER

SIX

BASIC USAGE

6.1 Geometric clustering

A geometric clustering is a hierarchical partition of the underlying geometry on which the kernel matrix is defined. See *htool::ClusterTreeBuilder::create_cluster_tree()* for more information about the available parameters to customize *htool::Cluster* construction.

6.1.1 Available customizations

To create geometric clustering with *htool::ClusterTreeBuilder*, two type of strategies allow customization of the algorithm defining n_{children} children of a cluster node:

1. Computation of the main directions for a given cluster node. Available strategies are:
    - *htool::ComputeLargestExtent*
    - *htool::ComputeBoundingBox*
2. Splitting position of a cluster node along a given direction. Available strategies are
    - *htool::RegularSplitting*
    - *htool::GeometricSplitting*

Strategies can be used to define a *htool::Partitioning* that can be given to *htool::ClusterTreeBuilder* via *htool::ClusterTreeBuilder::set_partitioning_strategy()*.

In any case, *htool::Partitioning* will compute the main direction of the current cluster, using the first strategy, and split the current cluster n_{children} times orthogonally to this direction using the second strategy. Except in the case where $n_{\text{children}} = 2^d$ where d is the spatial dimension, in which case it will split in two along each main directions.

See *here* for more advanced customization.

6.1.2 Visualisation

The geometric clustering can be exported to a file using *htool::save_clustered_geometry()*.

6.2 Hierarchical compression

6.2.1 Available customizations

Compression via *htool::HMatrixTreeBuilder* can be customized in the following aspects:

1. For admissible condition, i.e., the geometric a priori we have to define admissible bloc, see *Hierarchical clustering*, the current strategy is:
    - *htool::RjasanowSteinbach*

2. For low-rank compression, see *Low-rank compression*, the available strategies are:
    - `htool::SVD`
    - `htool::fullACA`
    - `htool::partialACA`
    - `htool::sympartialACA`
    - `htool::RecompressedLowRankGenerator`

Strategies are then given to `htool::HMatrixTreeBuilder` via its constructor or its function members

1. `htool::HMatrixTreeBuilder::set_admissibility_condition()`,
2. `htool::HMatrixTreeBuilder::set_low_rank_generator()`.

The constructor of `htool::HMatrixTreeBuilder` also takes the usual parameters for $\mathcal{H}$-matrix compression (tolerance for low-rank compression, symmetry, etc.), see its documentation.

6.2.2 Visualisation

Blocks from a `htool::HMatrix` can be exported to a file with `htool::save_leaves_with_rank()`.

Information about compression can also but accessed from a `htool::HMatrix` with

- `htool::get_tree_parameters()`,
- `htool::get_hmatrix_information()`,

or directly exported to an output stream with

- `htool::print_tree_parameters()`,
- `htool::print_hmatrix_information()`.

6.3 DDM solvers

6.3.1 Preconditioner with overlap

To use *Schwarz preconditioners* with overlap, there are two possibilities using `htool::DDMSolverBuilder`:

1. Reuse a previous `htool::HMatrix` associated with the local problem without overlap. Typically, it can be taken from a `htool::DistributedOperator` using $\mathcal{H}$-matrices for compression. See this `constructor`.
1. Assemble a `htool::HMatrix` associated with the local problem with overlap. See this `constructor`.

Note

In both cases, additional information about the partitioning need to be given to `htool::DDMSolverBuilder` constructors.

CHAPTER

SEVEN

ADVANCED USAGE

7.1 Geometric clustering

1. `htool::VirtualDirectionComputationStrategy` defines the interface for choosing the direction whose orthogonal plane will be the cutting plane.
2. `htool::VirtualSplittingStrategy` defines how where split along the previous direction.

7.2 Hierarchical compression

1. *`htool::VirtualAdmissibilityCondition`* which is the admissible condition, i.e., the geometric a priori we have to define admissible bloc, see *Hierarchical clustering*.
2. *`htool::VirtualLowRankGenerator`* which defines the low-rank compression, see *Low-rank compression*.

CHAPTER

EIGHT

DEVELOPER GUIDE

8.1 Build and run tests

CHAPTER

NINE

PUBLIC API

9.1 Generator

template<typename **`CoefficientPrecision`**>

class **`VirtualGenerator`**

Define the interface for the user to give Htool a function generating dense sub-blocks of the global matrix the user wants to compress. This is done by the user implementing *VirtualGenerator::copy_submatrix*.

Template Parameters

`CoefficientPrecision` – Precision of the coefficients (float, double,…)

Public Functions

virtual void **`copy_submatrix`**`(`int M, int N, const int *rows, const int *cols, *CoefficientPrecision* *ptr`)` const = 0

Generate a dense sub-block of the global matrix the user wants to compress. Note that sub-blocks queried by Htool are potentially non-contiguous in the user's numbering.

Parameters

- **`M`** – **[in]** specifies the number of columns of the queried block
- **`N`** – **[in]** specifies the number of rows of the queried block
- **`rows`** – **[in]** is an integer array of size M. It specifies the queried columns in the user's numbering
- **`cols`** – **[in]** is an integer array of size N. It specifies the queried rows in the user's numbering
- **`ptr`** – **[out]** is a `CoefficientPrecision` precision array of size $M \times N$. Htool already allocates and desallocates it internally, so it should **not** be allocated by the user.

inline **`VirtualGenerator`**`()`

`VirtualGenerator``(`const *VirtualGenerator*&`)` = default

VirtualGenerator &**`operator=`**`(`const *VirtualGenerator*&`)` = default

`VirtualGenerator``(`*VirtualGenerator*&&`)` = default

VirtualGenerator &**`operator=`**`(`*VirtualGenerator*&&`)` = default

inline virtual **`~VirtualGenerator`**`()`

9.2 Geometric clustering

9.2.1 Builder

template<typename **T**>

class **ClusterTreeBuilder**

a *ClusterTreeBuilder* object encapsulates the parameters and strategies to create *Cluster* objects.

Template Parameters

T –

Public Functions

inline void **set_maximal_leaf_size**(int maximal_leaf_size)

inline void **set_is_complete**(bool is_complete)

inline void **set_partitioning_strategy**(std::shared_ptr<*VirtualPartitioning*<*T*>> partitioning_strategy)

Cluster<*T*> **create_cluster_tree**(int number_of_points, int spatial_dimension, const *T* *coordinates, const *T* *radii, const *T* *weights, int number_of_children, int size_of_partition, const int *partition) const

It performs a hierarchical partitioning of the geometry defined with `coordinates` using the *VirtualPartitioning* strategy contained in the *ClusterTreeBuilder* object. Each geometric point is associated with a weight and a radius, which are used to compute the radius and center of each cluster node. A special level of the cluster tree, called level of partition, can be customized, either by its size or its content.

Parameters

- **number_of_points** – **[in]**
- **spatial_dimension** – **[in]**
- **coordinates** – **[in]** is a column-major `T` array of dimension `spatial_dimension` x `number_of_points`, corresponding to the geometric points.
- **radii** – **[in]** is a `T` array of dimension `number_of_points`
- **weights** – **[in]** is a `T` array of dimension `number_of_points`
- **number_of_children** – **[in]** specifies the number of children in the cluster tree, except for the parent of the level of partition.
- **size_of_partition** – **[in]** specifies the number of cluster node on the level of partition.
- **partition** – **[in]** is an integer array of size 2 x `size_of_partition`. Pairs of (offset,size) for each cluster node on the level of partition.

Returns

Cluster root of the cluster tree

inline *Cluster*<*T*> **create_cluster_tree**(int number_of_points, int spatial_dimension, const *T* *coordinates, int number_of_children, int size_of_partition) const

Same as *main constructor* but defaults weights and radiis and defines the partition automatically using the contained *VirtualPartitioning* strategy.

Parameters

- **number_of_points** –
- **spatial_dimension** –

- `coordinates` –
- `number_of_children` –
- `size_of_partition` –

Returns

Cluster root of the cluster tree

inline *Cluster*<*T*> `create_cluster_tree`(int number_of_points, int spatial_dimension, const *T* *coordinates, int number_of_children, int size_of_partition, const int *partition) const

Same as *main constructor* but defaults weights and radiis.

Parameters

- `number_of_points` –
- `spatial_dimension` –
- `coordinates` –
- `number_of_children` –
- `size_of_partition` –
- `partition` –

Returns

Cluster root of the cluster tree

9.2.2 Cluster

template<typename `CoordinatesPrecision`>

class `Cluster` : public htool::TreeNode<*Cluster*<*CoordinatesPrecision*>, ClusterTreeData<*CoordinatesPrecision*>>

Public Functions

inline `Cluster`(*CoordinatesPrecision* radius, std::vector<*CoordinatesPrecision*> &center, int rank, int offset, int size)

inline `Cluster`(const *Cluster* &parent, *CoordinatesPrecision* radius, std::vector<*CoordinatesPrecision*> &center, int rank, int offset, int size, int counter, bool is_on_partition)

`Cluster`(const *Cluster*&) = delete

Cluster &`operator=`(const *Cluster*&) = delete

`Cluster`(*Cluster* &&cluster) noexcept = default

Cluster &`operator=`(*Cluster* &&cluster) noexcept = default

virtual `~Cluster`() = default

inline const *CoordinatesPrecision* &`get_radius`() const

inline const std::vector<*CoordinatesPrecision*> &`get_center`() const

inline int `get_rank`() const

inline int **`get_offset`**() const

inline int **`get_size`**() const

inline int **`get_counter`**() const

inline bool **`is_permutation_local`**() const

inline unsigned int **`get_maximal_depth`**() const

inline unsigned int **`get_minimal_depth`**() const

inline unsigned int **`get_maximal_leaf_size`**() const

inline const std::vector<const *Cluster*<*CoordinatesPrecision*>*> &**`get_clusters_on_partition`**() const

inline const *Cluster*<*CoordinatesPrecision*> &**`get_cluster_on_partition`**(size_t index) const

inline const *Cluster*<*CoordinatesPrecision*> &**`get_root_cluster`**() const

inline const std::vector<int> &**`get_permutation`**() const

inline std::vector<int> &**`get_permutation`**()

inline void **`set_is_permutation_local`**(bool is_permutation_local)

inline void **`set_minimal_depth`**(unsigned int minimal_depth)

inline void **`set_maximal_depth`**(unsigned int maximal_depth)

inline void **`set_maximal_leaf_size`**(unsigned int maximal_leaf_size)

inline bool **`operator==`**(const *Cluster*<*CoordinatesPrecision*> &rhs) const

9.2.3 Partitioning interface

template<typename **`CoordinatePrecision`**>

class **`VirtualPartitioning`**

Subclassed by *htool::Partitioning< CoordinatePrecision, ComputationDirectionPolicy, SplittingPolicy >*

Public Functions

virtual std::vector<std::pair<int, int>> **`compute_partitioning`**(*Cluster*<*CoordinatePrecision*> &current_cluster, int spatial_dimension, const *CoordinatePrecision* *coordinates, const *CoordinatePrecision* *const radii, const *CoordinatePrecision* *const weights, int number_of_partitions) = 0

inline virtual **`~VirtualPartitioning`**()

template<typename **`CoordinatePrecision`**, typename **`ComputationDirectionPolicy`**, typename **`SplittingPolicy`**>

class **`Partitioning`** : public htool::*VirtualPartitioning*<*CoordinatePrecision*>

Strategy to define a new splitting for a given cluster. It is based on `ComputationDirectionPolicy` which defines a strategy to compute the main directions of a given cluster, and `SplittingPolicy` which defines a strategy to split the cluster using these directions.

Template Parameters

- `CoordinatePrecision` –
- `ComputationDirectionPolicy` –
- `SplittingPolicy` –

Public Functions

inline virtual std::vector<std::pair<int, int>> **`compute_partitioning`**(*Cluster*<*CoordinatePrecision*> &current_cluster, int spatial_dimension, const *CoordinatePrecision* *coordinates, const *CoordinatePrecision* *const radii, const *CoordinatePrecision* *const weights, int number_of_partitions) override

It returns `number_of_partitions` pairs of (offset,size) defining a new splitting of the current cluster, and it updates the permutation stored in `current_cluster` to make this new splitting contiguous. If `number_of_partitions` is equal to two to the `spatial_dimension` power, it will split in two using `SplittingPolicy` along the main directions computed with `ComputationDirectionPolicy`. Otherwise, it will only split `number_of_partition` along the main direction.

Parameters

- `current_cluster` –
- `spatial_dimension` –
- `coordinates` –
- `radii` –
- `weights` –
- `number_of_partitions` –

Returns

9.2.4 Direction computation strategies

template<typename **T**>

class **`ComputeLargestExtent`**

It computes the main directions of a cluster by computing the eigenvectors of the covariance matrix.

Template Parameters

`T` –

Public Static Functions

static inline Matrix<*T*> **`compute_direction`**(const *Cluster*<*T*> &cluster, int spatial_dimension, const *T* *const coordinates, const *T**const, const *T* *const weights)

template<typename **T**>

class **`ComputeBoundingBox`**

It computes the main directions of a cluster by choosing the x/y/z axis containing its largest extent in descreasing order.

Template Parameters
`T` –

Public Static Functions

static inline Matrix<*T*> **`compute_direction`**(const *Cluster*<*T*> &cluster, int spatial_dimension, const *T* *const coordinates, const *T**const, const *T**const)

9.2.5 Splitting strategies

template<typename `T`>

class **`GeometricSplitting`**

It splits in a given number of partitions of near-equal geometric size along a specific direction.

Template Parameters
`T` –

Public Static Functions

static inline std::vector<std::pair<int, int>> **`splitting`**(int offset, int size, int spatial_dimension, const *T* *const coordinates, const std::vector<int> &permutation, const std::vector<*T*> &direction, int number_of_partition)

template<typename `T`>

class **`RegularSplitting`**

It splits in a given number of partitions of near-equal number of elements.

Template Parameters
`T` –

Public Static Functions

static inline std::vector<std::pair<int, int>> **`splitting`**(int offset, int size, int, const *T**const, const std::vector<int>&, const std::vector<*T*>&, int number_of_partition)

9.2.6 Visualisation

template<typename `CoordinatesPrecision`>
void `htool::`**`save_clustered_geometry`**(const *Cluster*<*CoordinatesPrecision*> &cluster_tree, int spatial_dimension, const *CoordinatesPrecision* *x0, std::string filename, const std::vector<int> &depths)

9.3 Hierarchical matrix

9.3.1 Builder

template<typename `CoefficientsPrecision`, typename `CoordinatesPrecision` = htool::underlying_type<*CoefficientsPrecision*>>
class **`HMatrixBuilder`**

Public Functions

inline **`HMatrixBuilder`**(int target_number_of_points, int target_spatial_dimension, const *CoordinatesPrecision* *target_coordinates, const *ClusterTreeBuilder*<*CoordinatesPrecision*> *target_cluster_tree_builder, int source_number_of_points, int source_spatial_dimension, const *CoordinatesPrecision* *source_coordinates, const *ClusterTreeBuilder*<*CoordinatesPrecision*> *source_cluster_tree_builder)

Parameters

- **`target_number_of_points`** –
- **`target_spatial_dimension`** –
- **`target_coordinates`** –
- **`target_cluster_tree_builder`** –
- **`source_number_of_points`** –
- **`source_spatial_dimension`** –
- **`source_coordinates`** –
- **`source_cluster_tree_builder`** –

inline **`HMatrixBuilder`**(int target_number_of_points, int target_spatial_dimension, const *CoordinatesPrecision* *target_coordinates, int source_number_of_points, int source_spatial_dimension, const *CoordinatesPrecision* *source_coordinates)

Parameters

- **`target_number_of_points`** –
- **`target_spatial_dimension`** –
- **`target_coordinates`** –
- **`source_number_of_points`** –
- **`source_spatial_dimension`** –
- **`source_coordinates`** –

inline **`HMatrixBuilder`**(int number_of_points, int spatial_dimension, const *CoordinatesPrecision* *coordinates)

Parameters

- **`number_of_points`** –
- **`spatial_dimension`** –
- **`coordinates`** –

inline *HMatrix*<*CoefficientsPrecision*, *CoordinatesPrecision*> **`build`**(const *VirtualGenerator*<*CoefficientsPrecision*> &generator, const *HMatrixTreeBuilder*<*CoefficientsPrecision*, *CoordinatesPrecision*> &hmatrix_tree_builder)

Parameters

- **`generator`** –

- **`hmatrix_tree_builder`** –

Returns

Public Members

Cluster<*CoordinatesPrecision*> **`target_cluster`**

Cluster<*CoordinatesPrecision*> **`source_cluster`**

template<typename **`CoefficientPrecision`**, typename **`CoordinatePrecision`** = underlying_type<*CoefficientPrecision*>>
class **`HMatrixTreeBuilder`**

a *HMatrixTreeBuilder* object encapsulates the parameters and strategies to create *HMatrix* objects.

Template Parameters

- **`CoefficientPrecision`** –
- **`CoordinatePrecision`** –

Public Functions

inline explicit **`HMatrixTreeBuilder`**(underlying_type<*CoefficientPrecision*> epsilon, *CoordinatePrecision* eta, char symmetry, char UPLO, int reqrank, std::shared_ptr<VirtualInternalLowRankGenerator<*CoefficientPrecision*>> low_rank_strategy, std::shared_ptr<*VirtualAdmissibilityCondition*<*CoordinatePrecision*>> admissibility_condition_strategy = nullptr)

inline explicit **`HMatrixTreeBuilder`**(underlying_type<*CoefficientPrecision*> epsilon, *CoordinatePrecision* eta, char symmetry, char UPLO, int reqrank = -1, std::shared_ptr<*VirtualLowRankGenerator*<*CoefficientPrecision*>> low_rank_strategy = nullptr, std::shared_ptr<*VirtualAdmissibilityCondition*<*CoordinatePrecision*>> admissibility_condition_strategy = nullptr)

`HMatrixTreeBuilder`(const *HMatrixTreeBuilder*&) = delete

HMatrixTreeBuilder &**`operator=`**(const *HMatrixTreeBuilder*&) = delete

`HMatrixTreeBuilder`(*HMatrixTreeBuilder*&&) noexcept = default

HMatrixTreeBuilder &**`operator=`**(*HMatrixTreeBuilder*&&) noexcept = default

virtual **`~HMatrixTreeBuilder`**() = default

HMatrixType **`build`**(const VirtualInternalGenerator<*CoefficientPrecision*> &generator, const ClusterType &target_root_cluster_tree, const ClusterType &source_root_cluster_tree, int target_partition_number, int partition_number_for_symmetry) const

inline HMatrixType **`build`**(const VirtualInternalGenerator<*CoefficientPrecision*> &generator, const ClusterType &target_root_cluster_tree, const ClusterType &source_root_cluster_tree, int target_partition_number) const inline HMatrixType **build**(const VirtualInternalGenerator<*CoefficientPrecision*> &generator, const ClusterType &target_root_cluster_tree, const ClusterType &source_root_cluster_tree) const

inline HMatrixType **build**(const *VirtualGenerator*<*CoefficientPrecision*> &generator, const ClusterType &target_root_cluster_tree, const ClusterType &source_root_cluster_tree, int target_partition_number, int partition_number_for_symmetry) const

inline HMatrixType **build**(const *VirtualGenerator*<*CoefficientPrecision*> &generator, const ClusterType &target_root_cluster_tree, const ClusterType &source_root_cluster_tree, int target_partition_number) const

inline HMatrixType **build**(const *VirtualGenerator*<*CoefficientPrecision*> &generator, const ClusterType &target_root_cluster_tree, const ClusterType &source_root_cluster_tree) const

inline void **set_symmetry**(char symmetry_type)

inline void **set_UPLO**(char UPLO_type)

inline void **set_low_rank_generator**(std::shared_ptr<*VirtualLowRankGenerator*<*CoefficientPrecision*>> ptr)

inline void **set_low_rank_generator**(std::shared_ptr<VirtualInternalLowRankGenerator<*CoefficientPrecision*>> ptr)

inline void **set_admissibility_condition**(std::shared_ptr<*VirtualAdmissibilityCondition*<*CoordinatePrecision*>> ptr)

inline void **set_minimal_source_depth**(int minimal_source_depth)

inline void **set_minimal_target_depth**(int minimal_target_depth)

inline void **set_dense_blocks_generator**(std::shared_ptr<VirtualDenseBlocksGenerator<*CoefficientPrecision*>> dense_blocks_generator)

inline void **set_block_tree_consistency**(bool consistency)

inline char **get_symmetry**() const

inline char **get_UPLO**() const

inline double **get_epsilon**() const

inline double **get_eta**() const

inline std::shared_ptr<VirtualInternalLowRankGenerator<*CoefficientPrecision*>> **get_internal_low_rank_generator**() cons

inline std::shared_ptr<*VirtualLowRankGenerator*<*CoefficientPrecision*>> **get_low_rank_generator**() const

9.3.2 HMatrix

template<typename **CoefficientPrecision**, typename **CoordinatePrecision** = underlying_type<*CoefficientPrecision*>>
class **HMatrix** : public htool::TreeNode<*HMatrix*<*CoefficientPrecision*, underlying_type<*CoefficientPrecision*>>, HMatrixTreeData<*CoefficientPrecision*, underlying_type<*CoefficientPrecision*>>>

Public Types

enum class **`StorageType`**

Values:

enumerator **`Dense`**

enumerator **`LowRank`**

enumerator **`Hierarchical`**

Public Functions

inline **`HMatrix`**(const *Cluster*<*CoordinatePrecision*> &target_cluster, const *Cluster*<*CoordinatePrecision*> &source_cluster)

inline **`HMatrix`**(const *HMatrix* &parent, const *Cluster*<*CoordinatePrecision*> *target_cluster, const *Cluster*<*CoordinatePrecision*> *source_cluster)

inline **`HMatrix`**(const *HMatrix* &rhs)

inline *HMatrix* &**`operator=`**(const *HMatrix* &rhs)

`HMatrix`(*HMatrix*&&) noexcept = default

HMatrix &**`operator=`**(*HMatrix*&&) noexcept = default

virtual **`~HMatrix`**() = default

inline const *Cluster*<*CoordinatePrecision*> &**`get_target_cluster`**() const

inline const *Cluster*<*CoordinatePrecision*> &**`get_source_cluster`**() const

inline int **`nb_cols`**() const

inline int **`nb_rows`**() const

inline htool::underlying_type<*CoefficientPrecision*> **`get_epsilon`**() const

inline *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> ***`get_child_or_this`**(const *Cluster*<*CoordinatePrecision*> &required_target_cluster, const *Cluster*<*CoordinatePrecision*> &required_source_cluster)

inline const *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> ***`get_child_or_this`**(const *Cluster*<*CoordinatePrecision*> &required_target_cluster, const *Cluster*<*CoordinatePrecision*> &required_source_cluster) const inline int **`get_rank`**() const

inline const Matrix<*CoefficientPrecision*> ***`get_dense_data`**() const

inline Matrix<*CoefficientPrecision*> ***`get_dense_data`**()

inline const LowRankMatrix<*CoefficientPrecision*> ***`get_low_rank_data`**() const

inline LowRankMatrix<*CoefficientPrecision*> ***`get_low_rank_data`**()

inline char **`get_symmetry`**() const

inline char **`get_UPLO`**() const

inline const HMatrixTreeData<*CoefficientPrecision*, *CoordinatePrecision*> ***`get_hmatrix_tree_data`**() const

inline const *HMatrix*<*CoefficientPrecision*> ***`get_sub_hmatrix`**(const *Cluster*<*CoordinatePrecision*> &target_cluster, const *Cluster*<*CoordinatePrecision*> &source_cluster) const

inline *HMatrix*<*CoefficientPrecision*> ***`get_sub_hmatrix`**(const *Cluster*<*CoordinatePrecision*> &target_cluster, const *Cluster*<*CoordinatePrecision*> &source_cluster)

inline *StorageType* **`get_storage_type`**() const

inline void **`set_symmetry`**(char symmetry)

inline void **`set_UPLO`**(char UPLO)

inline void **`set_symmetry_for_leaves`**(char symmetry)

inline void **`set_UPLO_for_leaves`**(char UPLO)

inline void **`set_target_cluster`**(const *Cluster*<*CoordinatePrecision*> *new_target_cluster)

inline bool **`is_dense`**() const

inline bool **`is_low_rank`**() const

inline bool **`is_hierarchical`**() const

inline void **`set_eta`**(*CoordinatePrecision* eta)

inline void **`set_epsilon`**(underlying_type<*CoefficientPrecision*> epsilon)

inline void **`set_low_rank_generator`**(std::shared_ptr<VirtualInternalLowRankGenerator<*CoefficientPrecision*>> ptr)

inline void **`set_admissibility_condition`**(std::shared_ptr<*VirtualAdmissibilityCondition*<*CoordinatePrecision*>> ptr)

inline void **`set_minimal_target_depth`**(unsigned int minimal_target_depth)

inline void **`set_minimal_source_depth`**(unsigned int minimal_source_depth) inline void **set_block_tree_consistency**(bool consistency)

inline char **get_symmetry_for_leaves**() const

inline char **get_UPLO_for_leaves**() const

inline bool **is_block_tree_consistent**() const

inline void **compute_dense_data**(const VirtualInternalGenerator<*CoefficientPrecision*> &generator)

inline bool **compute_low_rank_data**(const VirtualInternalLowRankGenerator<*CoefficientPrecision*> &low_rank_generator, int reqrank, underlying_type<*CoefficientPrecision*> epsilon)

inline void **clear_low_rank_data**()

inline void **set_dense_data**(std::unique_ptr<Matrix<*CoefficientPrecision*>> dense_matrix_ptr)

9.3.3 Admissibility conditions

template<typename **CoordinatePrecision**>

class **VirtualAdmissibilityCondition**

Subclassed by *htool::RjasanowSteinbach< CoordinatePrecision >*

Public Functions

virtual bool **ComputeAdmissibility**(const *Cluster*<*CoordinatePrecision*> &target, const *Cluster*<*CoordinatePrecision*> &source, double eta) const = 0

inline virtual **~VirtualAdmissibilityCondition**()

template<typename **CoordinatePrecision**>

class **RjasanowSteinbach** : public htool::*VirtualAdmissibilityCondition*<*CoordinatePrecision*>

Public Functions

inline virtual bool **ComputeAdmissibility**(const *Cluster*<*CoordinatePrecision*> &target, const *Cluster*<*CoordinatePrecision*> &source, double eta) const override

9.3.4 Low-rank compression

template<typename **CoefficientPrecision**, typename **CoordinatesPrecision** = underlying_type<*CoefficientPrecision*>>
class **VirtualLowRankGenerator**

Public Functions

inline **VirtualLowRankGenerator**()

virtual bool **copy_low_rank_approximation**(int M, int N, const int *rows, const int *cols, LowRankMatrix<*CoefficientPrecision*> &lrmat) const = 0 virtual bool **copy_low_rank_approximation**(int M, int N, const int *rows, const int *cols, int reqrank, LowRankMatrix<*CoefficientPrecision*> &lrmat) const = 0

inline virtual **~VirtualLowRankGenerator**()

template<typename **CoefficientPrecision**>

class **SVD** : public htool::VirtualInternalLowRankGenerator<*CoefficientPrecision*>

It provides low-rank approximation using truncated Singular Value Decomposition (*SVD*).

Template Parameters

CoefficientPrecision –

Public Functions

inline **SVD**(const VirtualInternalGenerator<*CoefficientPrecision*> &A)

inline **SVD**(const *VirtualGenerator*<*CoefficientPrecision*> &A)

inline virtual bool **copy_low_rank_approximation**(int M, int N, int row_offset, int col_offset, LowRankMatrix<*CoefficientPrecision*> &lrmat) const override

inline virtual bool **copy_low_rank_approximation**(int M, int N, int row_offset, int col_offset, int rank, LowRankMatrix<*CoefficientPrecision*> &lrmat) const override

template<typename **CoefficientPrecision**>

class **fullACA** : public htool::VirtualInternalLowRankGenerator<*CoefficientPrecision*>

It provides low-rank approximation using the full Adaptive Cross Approximation (ACA).

Template Parameters

CoefficientPrecision –

Public Functions

inline **fullACA**(const VirtualInternalGenerator<*CoefficientPrecision*> &A)

inline **fullACA**(const *VirtualGenerator*<*CoefficientPrecision*> &A)

inline virtual bool **copy_low_rank_approximation**(int M, int N, int row_offset, int col_offset, LowRankMatrix<*CoefficientPrecision*> &lrmat) const override

inline virtual bool **copy_low_rank_approximation**(int M, int N, int row_offset, int col_offset, int reqrank, LowRankMatrix<*CoefficientPrecision*> &lrmat) const override

template<typename **CoefficientPrecision**>

class **partialACA** : public htool::VirtualInternalLowRankGenerator<*CoefficientPrecision*>

It provides low-rank approximation using the partial Adaptive Cross Approximation (ACA).

Public Functions

inline **partialACA**(const VirtualInternalGenerator<*CoefficientPrecision*> &A)

inline **partialACA**(const *VirtualGenerator*<*CoefficientPrecision*> &A)

inline virtual bool **copy_low_rank_approximation**(int M, int N, int row_offset, int col_offset, LowRankMatrix<*CoefficientPrecision*> &lrmat) const override

inline virtual bool **copy_low_rank_approximation**(int M, int N, int row_offset, int col_offset, int reqrank, LowRankMatrix<*CoefficientPrecision*> &lrmat) const override

template<typename **CoefficientPrecision**>

class **sympartialACA** : public htool::VirtualInternalLowRankGenerator<*CoefficientPrecision*>

It provides low-rank approximation using a symmetric version of the partial Adaptive Cross Approximation (ACA).

Public Functions

inline **sympartialACA**(const VirtualInternalGenerator<*CoefficientPrecision*> &A)

inline **sympartialACA**(const *VirtualGenerator*<*CoefficientPrecision*> &A)

inline virtual bool **copy_low_rank_approximation**(int M, int N, int row_offset, int col_offset, LowRankMatrix<*CoefficientPrecision*> &lrmat) const override

inline virtual bool **copy_low_rank_approximation**(int M, int N, int row_offset, int col_offset, int reqrank, LowRankMatrix<*CoefficientPrecision*> &lrmat) const override

template<typename **CoefficientPrecision**>

class **RecompressedLowRankGenerator** : public htool::VirtualInternalLowRankGenerator<*CoefficientPrecision*>

It provides low-rank approximation using a given compression method with a QR-SVD recompression.

Template Parameters

CoefficientPrecision –

Public Functions

inline **RecompressedLowRankGenerator**(const VirtualInternalLowRankGenerator<*CoefficientPrecision*> &low_rank_generator, std::function<void(LowRankMatrix<*CoefficientPrecision*>&)> recompression)

inline virtual bool **copy_low_rank_approximation**(int M, int N, int row_offset, int col_offset, LowRankMatrix<*CoefficientPrecision*> &lrmat) const

inline virtual bool **copy_low_rank_approximation**(int M, int N, int row_offset, int col_offset, int reqrank, LowRankMatrix<*CoefficientPrecision*> &lrmat) const

9.3.5 Visualisation

template<typename **CoefficientPrecision**, typename **CoordinatePrecision** = underlying_type<*CoefficientPrecision*>>
void htool::**save_leaves_with_rank**(const *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> &hmatrix, std::string filename)

template<typename **CoefficientPrecision**, typename **CoordinatePrecision** = underlying_type<*CoefficientPrecision*>>
std::map<std::string, std::string> htool::**get_tree_parameters**(const *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> &hmatrix)

template<typename **CoefficientPrecision**, typename **CoordinatePrecision** = underlying_type<*CoefficientPrecision*>>
void htool::**print_tree_parameters**(const *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> &hmatrix, std::ostream &os)

template<typename **CoefficientPrecision**, typename **CoordinatePrecision** = underlying_type<*CoefficientPrecision*>>
std::map<std::string, std::string> htool::**get_hmatrix_information**(const *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> &hmatrix)

template<typename **CoefficientPrecision**, typename **CoordinatePrecision** = underlying_type<*CoefficientPrecision*>>
void htool::**print_hmatrix_information**(const *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> &hmatrix, std::ostream &os)

9.3.6 Linear algebra

template<typename **ExecutionPolicy**, typename **CoefficientPrecision**, typename **CoordinatePrecision** = underlying_type<*CoefficientPrecision*>>
void htool::**add_hmatrix_vector_product**(*ExecutionPolicy*&&, char trans, *CoefficientPrecision* alpha, const *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> &A, const *CoefficientPrecision* *in, *CoefficientPrecision* beta, *CoefficientPrecision* *out, *CoefficientPrecision* *buffer = nullptr)

Template Parameters

- **ExecutionPolicy** –
- **CoefficientPrecision** –
- **CoordinatePrecision** –

Parameters

- **trans** –
- **alpha** –
- **A** –
- **in** –
- **beta** –
- **out** –
- **buffer** –

template<typename **CoefficientPrecision**, typename **CoordinatePrecision** = underlying_type<*CoefficientPrecision*>>
void htool::**add_hmatrix_vector_product**(char trans, *CoefficientPrecision* alpha, const *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> &A, const *CoefficientPrecision* *in, *CoefficientPrecision* beta, *CoefficientPrecision* *out, *CoefficientPrecision* *buffer = nullptr)

Template Parameters

- **CoefficientPrecision** –
- **CoordinatePrecision** –

Parameters

- **trans** –
- **alpha** –
- **A** –
- **in** –
- **beta** –
- **out** –
- **buffer** –

template<typename **ExecutionPolicy**, typename **CoefficientPrecision**, typename **CoordinatePrecision** = underlying_type<*CoefficientPrecision*>>
void htool::**add_hmatrix_matrix_product**(*ExecutionPolicy* &&execution_policy, char transa, char transb, *CoefficientPrecision* alpha, const *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> &A, const Matrix<*CoefficientPrecision*> &B, *CoefficientPrecision* beta, Matrix<*CoefficientPrecision*> &C, *CoefficientPrecision* *buffer = nullptr)

test

Template Parameters

- **ExecutionPolicy** –
- **CoefficientPrecision** –
- **CoordinatePrecision** –

Parameters

- **execution_policy** –
- **transa** –
- **transb** –
- **alpha** –
- **A** –
- **B** –
- **beta** –
- **C** –
- **buffer** –

template<typename **CoefficientPrecision**, typename **CoordinatePrecision** = underlying_type<*CoefficientPrecision*>>
void htool::**add_hmatrix_matrix_product**(char transa, char transb, *CoefficientPrecision* alpha, const *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> &A, const Matrix<*CoefficientPrecision*> &B, *CoefficientPrecision* beta, Matrix<*CoefficientPrecision*> &C, *CoefficientPrecision* *buffer = nullptr)

Template Parameters

- **CoefficientPrecision** –
- **CoordinatePrecision** –

Parameters

- **transa** –
- **transb** –
- **alpha** –
- **A** –
- **B** –
- **beta** –
- **C** –
- **buffer** –

template<typename **CoefficientPrecision**, typename **CoordinatePrecision** = underlying_type<*CoefficientPrecision*>>
void htool::**lu_factorization**(*HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> &hmatrix)

Template Parameters

- **CoefficientPrecision** –
- **CoordinatePrecision** –

Parameters
hmatrix –

template<typename **CoefficientPrecision**, typename **CoordinatePrecision** = underlying_type<*CoefficientPrecision*>>
void htool::**lu_solve**(char trans, const *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> &A, Matrix<*CoefficientPrecision*> &X)

Template Parameters

- **CoefficientPrecision** –
- **CoordinatePrecision** –

Parameters

- **trans** –
- **A** –
- **X** –

template<typename **CoefficientPrecision**, typename **CoordinatePrecision** = underlying_type<*CoefficientPrecision*>>

void htool::**cholesky_factorization**(char UPLO, *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> &hmatrix)

Template Parameters

- **CoefficientPrecision** –
- **CoordinatePrecision** –

Parameters

- **UPLO** –
- **hmatrix** –

template<typename **CoefficientPrecision**, typename **CoordinatePrecision** = underlying_type<*CoefficientPrecision*>>
void htool::**cholesky_solve**(char UPLO, const *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> &A, Matrix<*CoefficientPrecision*> &X)

Template Parameters

- **CoefficientPrecision** –
- **CoordinatePrecision** –

Parameters

- **UPLO** –
- **A** –
- **X** –

9.4 Distributed operator

9.4.1 Builder

template<typename **CoefficientPrecision**, typename **CoordinatePrecision** = underlying_type<*CoefficientPrecision*>>
class **DefaultApproximationBuilder**

Public Functions

inline **DefaultApproximationBuilder**(const VirtualInternalGenerator<*CoefficientPrecision*> &generator, const *Cluster*<*CoordinatePrecision*> &target_cluster, const *Cluster*<*CoordinatePrecision*> &source_cluster, const *HMatrixTreeBuilder*<*CoefficientPrecision*, *CoordinatePrecision*> &hmatrix_tree_builder, MPI_Comm communicator)

inline **DefaultApproximationBuilder**(const *VirtualGenerator*<*CoefficientPrecision*> &generator, const *Cluster*<*CoordinatePrecision*> &target_cluster, const *Cluster*<*CoordinatePrecision*> &source_cluster, const *HMatrixTreeBuilder*<*CoefficientPrecision*, *CoordinatePrecision*> &hmatrix_tree_builder, MPI_Comm communicator)

Public Members

HMatrix<*CoefficientPrecision*, *CoordinatePrecision*> **`hmatrix`**

DistributedOperator<*CoefficientPrecision*> &**`distributed_operator`**

const *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> ***`block_diagonal_hmatrix`** = {nullptr}

template<typename **`CoefficientPrecision`**, typename **`CoordinatePrecision`** = underlying_type<*CoefficientPrecision*>>
class **`DefaultLocalApproximationBuilder`**

Public Functions

inline **`DefaultLocalApproximationBuilder`**(const VirtualInternalGenerator<*CoefficientPrecision*> &generator, const *Cluster*<*CoordinatePrecision*> &target_cluster, const *Cluster*<*CoordinatePrecision*> &source_cluster, const *HMatrixTreeBuilder*<*CoefficientPrecision*, *CoordinatePrecision*> &hmatrix_tree_builder, MPI_Comm communicator)

inline **`DefaultLocalApproximationBuilder`**(const *VirtualGenerator*<*CoefficientPrecision*> &generator, const *Cluster*<*CoordinatePrecision*> &target_cluster, const *Cluster*<*CoordinatePrecision*> &source_cluster, const *HMatrixTreeBuilder*<*CoefficientPrecision*, *CoordinatePrecision*> &hmatrix_tree_builder, MPI_Comm communicator)

Public Members

HMatrix<*CoefficientPrecision*, *CoordinatePrecision*> **`hmatrix`**

DistributedOperator<*CoefficientPrecision*> &**`distributed_operator`**

const *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> ***`block_diagonal_hmatrix`** = {nullptr}

template<typename **`CoefficientPrecision`**, typename **`CoordinatePrecision`** = underlying_type<*CoefficientPrecision*>>
class **`CustomApproximationBuilder`**

Public Functions

inline explicit **`CustomApproximationBuilder`**(const *Cluster*<*CoordinatePrecision*> &target_cluster, const *Cluster*<*CoordinatePrecision*> &source_cluster, char symmetry, char UPLO, MPI_Comm communicator, const VirtualLocalOperator<*CoefficientPrecision*> &local_operator)

Public Members

DistributedOperator<*CoefficientPrecision*> **distributed_operator**

9.4.2 DistributedOperator

template<typename **CoefficientPrecision**>

class **DistributedOperator**

Public Functions

DistributedOperator(const *DistributedOperator*&) = delete

DistributedOperator &**operator=**(const *DistributedOperator*&) = delete

DistributedOperator(*DistributedOperator* &&cluster) noexcept = default

DistributedOperator &**operator=**(*DistributedOperator* &&cluster) noexcept = default

virtual **~DistributedOperator**() = default

inline explicit **DistributedOperator**(const VirtualPartition<*CoefficientPrecision*> &target_partition, const VirtualPartition<*CoefficientPrecision*> &source_partition, char symmetry, char UPLO, MPI_Comm comm)

inline void **add_local_operator**(const VirtualLocalOperator<*CoefficientPrecision*> *local_operator)

void **vector_product_global_to_global**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out) const

void **matrix_product_global_to_global**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out, int mu) const

void **vector_product_transp_global_to_global**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out) const

void **matrix_product_transp_global_to_global**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out, int mu) const

void **vector_product_local_to_local**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out, *CoefficientPrecision* *work = nullptr) const

void **matrix_product_local_to_local**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out, int mu, *CoefficientPrecision* *work = nullptr) const

void **vector_product_transp_local_to_local**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out, *CoefficientPrecision* *work = nullptr) const

void **matrix_product_transp_local_to_local**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out, int mu, *CoefficientPrecision* *work = nullptr) const

void **internal_vector_product_global_to_local**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out) const void **internal_matrix_product_global_to_local**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out, int mu) const

void **internal_vector_product_transp_local_to_global**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out) const

void **internal_matrix_product_transp_local_to_global**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out, int mu) const

void **internal_vector_product_local_to_local**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out, *CoefficientPrecision* *work = nullptr) const

void **internal_matrix_product_local_to_local**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out, int mu, *CoefficientPrecision* *work = nullptr) const

void **internal_vector_product_transp_local_to_local**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out, *CoefficientPrecision* *work = nullptr) const

void **internal_matrix_product_transp_local_to_local**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out, int mu, *CoefficientPrecision* *work = nullptr) const

void **internal_sub_matrix_product_to_local**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out, int mu, int offset, int size) const

void **local_to_global_source**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out, const int &mu) const

void **local_to_global_target**(const *CoefficientPrecision* *const in, *CoefficientPrecision* *const out, const int &mu) const

inline bool &**use_permutation**()

inline const bool &**use_permutation**() const

inline char **get_symmetry_type**() const

inline char **get_storage_type**() const

inline MPI_Comm **get_comm**() const

inline const VirtualPartition<*CoefficientPrecision*> &**get_target_partition**() const

inline const VirtualPartition<*CoefficientPrecision*> &**get_source_partition**() const

9.5 DDM solvers

9.5.1 Builder

template<typename **CoefficientPrecision**, typename **CoordinatePrecision** = underlying_type<*CoefficientPrecision*>>

class `DDMSolverBuilder`

Builder that encapsulates all the implementation details for constructing *DDM* object.

Template Parameters

- `CoefficientPrecision` –
- `CoordinatePrecision` –

Public Functions

inline `DDMSolverBuilder`(*DistributedOperator*<*CoefficientPrecision*> &distributed_operator, *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> &block_diagonal_hmatrix)

Builder for *DDM* Block Jacobi solver (without overlap). The `block_diagonal_hmatrix` corresponding to the local subproblem is usually taken from the local diagonal block of the `distributed_operator`.

Parameters

- `distributed_operator` –
- `block_diagonal_hmatrix` –

inline `DDMSolverBuilder`(*DistributedOperator*<*CoefficientPrecision*> &distributed_operator, *HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*> &block_diagonal_hmatrix, const *VirtualGenerator*<*CoefficientPrecision*> &generator, const std::vector<int> &ovr_subdomain_to_global, const std::vector<int> &cluster_to_ovr_subdomain, const std::vector<int> &neighbors, const std::vector<std::vector<int>> &intersections)

Builder for *DDM* Schwarz solver (with overlap), where `block_diagonal_hmatrix` correspond to the problem for the subdomain without overlap, usually taken from the local diagonal block of the `distributed_operator`. The problem in the overlapping subdomain is then assembled as a 2x2 block matrix where one diagonal block is the previous *HMatrix*, and the other blocks are dense blocks stored in *blocks_in_overlap*.

Parameters

- `distributed_operator` –
- `block_diagonal_hmatrix` –
- `generator` –
- `ovr_subdomain_to_global` –
- `cluster_to_ovr_subdomain` –
- `neighbors` –
- `intersections` –

inline `DDMSolverBuilder`(*DistributedOperator*<*CoefficientPrecision*> &distributed_operator, const std::vector<int> &ovr_subdomain_to_global, const std::vector<int> &cluster_to_ovr_subdomain, const std::vector<int> &neighbors, const std::vector<std::vector<int>> &intersections, const *VirtualGenerator*<*CoefficientPrecision*> &generator, int spatial_dimension, const *CoordinatePrecision* *global_geometry, const *CoordinatePrecision* *radii, const *CoordinatePrecision* *weights, const *ClusterTreeBuilder*<*CoordinatePrecision*> &cluster_tree_builder, const *HMatrixTreeBuilder*<*CoefficientPrecision*, *CoordinatePrecision*> &local_hmatrix_builder)

Builder for *DDM* Schwarz solver (with overlap), where a *HMatrix* for the local subproblem with overlap is assembled and stored in *local_hmatrix*. It can be useful when the *DistributedOperator* is not using any *HMatrix* for local compression.

Parameters

- **`distributed_operator`** –
- **`ovr_subdomain_to_global`** –
- **`cluster_to_ovr_subdomain`** –
- **`neighbors`** –
- **`intersections`** –
- **`generator`** –
- **`spatial_dimension`** –
- **`global_geometry`** –
- **`radii`** –
- **`weights`** –
- **`cluster_tree_builder`** –
- **`local_hmatrix_builder`** –

inline `DDMSolverBuilder`(*DistributedOperator*<*CoefficientPrecision*> &distributed_operator, const std::vector<int> &ovr_subdomain_to_global, const std::vector<int> &cluster_to_ovr_subdomain, const std::vector<int> &neighbors, const std::vector<std::vector<int>> &intersections, const *VirtualGenerator*<*CoefficientPrecision*> &generator, int spatial_dimension, const *CoordinatePrecision* *global_geometry, const *ClusterTreeBuilder*<*CoordinatePrecision*> &cluster_tree_builder, const *HMatrixTreeBuilder*<*CoefficientPrecision*, *CoordinatePrecision*> &local_hmatrix_builder)

Public Members

const std::vector<int> &**`local_to_global_numbering`**

std::unique_ptr<*HMatrix*<*CoefficientPrecision*, *CoordinatePrecision*>> **`local_hmatrix`**

HMatrix associated with the local problem with overlap. Built only if not using a *HMatrix* associated h the local problem without overlap.

std::array<Matrix<*CoefficientPrecision*>, 3> **`blocks_in_overlap`**

Dense blocks related to the overlap when using a solver with overlap, and a *HMatrix* for the local problem without overlap.

DDM<*CoefficientPrecision*, HPDDMCustomLocalSolver> **`solver`**

Resulting solver from builder.

9.5.2 DDM

template<typename **CoefficientPrecision**, template<class> class **LocalSolver**>

class **DDM**

Public Functions

DDM(const *DDM*&) = delete

DDM &**operator=**(const *DDM*&) = delete

DDM(*DDM* &&cluster) noexcept = default

DDM &**operator=**(*DDM* &&cluster) noexcept = default

virtual **~DDM**() = default

inline void **clean**()

inline **DDM**(int size_w_overlap, const *DistributedOperator*<*CoefficientPrecision*> &distributed_operator, std::unique_ptr<HPDDMOperator<*CoefficientPrecision*, *LocalSolver*>> hpddm_op)

inline void **facto_one_level**()

inline void **build_coarse_space**(VirtualCoarseSpaceBuilder<*CoefficientPrecision*> &coarse_space_builder, VirtualCoarseOperatorBuilder<*CoefficientPrecision*> &coarse_operator_builder)

inline void **solve**(const *CoefficientPrecision* *const rhs, *CoefficientPrecision* *const x, const int &mu = 1)

inline void **print_infos**() const

inline void **save_infos**(const std::string &outputname, std::ios_base::openmode mode = std::ios_base::app, const std::string &sep = " = ") const

inline void **add_infos**(std::string key, std::string value) const

inline void **set_infos**(std::string key, std::string value) const

inline std::string **get_information**(const std::string &key) const

inline std::map<std::string, std::string> **get_information**() const

inline int **get_nevi**() const

inline int **get_local_size**() const

CHAPTER

TEN

QUICKSTART

10.1 Dependencies

Htool-DDM's Python interface is based on pybind11 and Htool-DDM *C++ header-only library*. It requires

- C++ compiler with standard 14,
- Python 3,
- MPI implementation for distributed parallelism,
- BLAS, to perform algebraic dense operations,
- pybind11 to define the python interface,
- cmake to build the python interface,
- HPDDM and its dependencies (BLAS, LAPACK) to use iterative solvers and DDM preconditioners,

Note

Htool-DDM *C++ header-only library*, HPDDM and pybind11 are git submodules of Htool-DDM python interface. Thus, users do not need to install them.

It also requires the following python packages

- mpi4py for using MPI via Python,
- numpy for basic linear algebra in Python.

And optionnally,

- matplotlib for visualisation.

10.2 Compilation

First, you need to clone this repository with its submodules:

```
git clone --recurse-submodules https://github.com/htool-ddm/htool_python.git && cd htool_
↪python
```

In the folder of this repository, do:

```
pip install .
```

In case you need to pass cmake variables, you can use

```
CMAKE_ARGS="-DCMAKE_VAR=VALUE1 -DCMAKE_VAR_2=VALUE2" pip install .
```

10.3 Hierarchical compression

10.3.1 Coefficient generator

10.3.2 Build a hierarchical matrix

10.3.3 Use a hierarchical matrix

10.4 DDM Solver

CHAPTER

ELEVEN

BASIC USAGE

11.1 Defining a geometry

11.2 Defining a IMatrix

11.3 Build a HMatrix

11.4 Use a HMatrix

11.5 Full example

11.6 Solvers

CHAPTER TWELVE

ADVANCED USAGE

12.1 Installation

Htool is a C++11 header library, you need to download it, using for example

```
git clone https://github.com/htool-ddm/htool
```

and you need to include the header files contained in include/htool. Note that Htool requires

- a MPI implementation, to perform communications on parallel computing architectures,
- a BLAS implementation, to perform algebraic operations (dense matrix-matrix or matrix-vector operations).

And optionally

- LAPACK, to perform SVD compression,
- HPDDM and its dependencies (BLAS, LAPACK) to use iterative solvers and DDM preconditioners.

12.2 Required inputs

At the very least, Htool requires a geometry and a function to generate the coefficients of the matrix you wish to compress. More precisely,

- the geometry is a matrix of size $d \times N$, where N is the number of points and d the geometric dimension, stored in a column-major array,
- the function generating the coefficients is passed to Htool using the interface *htool::VirtualGenerator*.

Here is an example where we define the interaction between to set of points `target_points` and `source_points` with $\kappa(x, y) = e^{-|x-y|}$:

```
class MyMatrix : public VirtualGenerator<double> {
 const vector<double> & target_points;
 const vector<double> & source_points;
 int dimension;

 public:
 MyMatrix(int dimension0, int nr, int nc, const vector<double> &p10, const  vector
↪<double> &p20) : VirtualGenerator(nr, nc), target_points (target_points_0), source_
↪points(source_points_0), dimension(dimension0) {}

 void copy_submatrix(int M, int N, const int *const rows, const int *const  cols, double␣
↪*ptr) const override {
  for (int j = 0; j < M; j++) {
```

(continues on next page)

(continued from previous page)

```
    for (int k = 0; k < N; k++) {
     double norm2=0;
     for (int p = 0; p < dimension; p++) {
      norm2+= (target_points[p+dimension*row[j]]-source_
↪points[p+dimension*cols[k]])*(target_points[p+dimension*row[j]]-source_
↪points[p+dimension*cols[k]]);
     }
     ptr[j + M * k] = exp(-sqrt(norm2));
    }
   }
  }
};
```

Note

Htool does not have any a priori information on the kernel, so it is up to the user to optimize the computations in *htool::VirtualGenerator::copy_submatrix*.

12.3 Build a HMatrix

12.4 Use a HMatrix

12.5 Full example

12.6 Solvers

CHAPTER

THIRTEEN

DEVELOPER GUIDE

13.1 Build and run tests

CHAPTER

FOURTEEN

BUILD DOCUMENTATION

At the root of the repository do:

```
pip install -r requirements.txt
```

It will install the necessary dependencies. The documentation can be built using a `<builder>`

```
cd docs & make <builder>
```

where `<builder>` can be any supported builder by Sphinx, e.g, `html` or `latex`.

INDEX