## Contents

1 Referencing 2

2 Contents:
   2.1 Algebraic Multigrid ........................................ 3
   2.2 Design Principles ........................................... 4
   2.3 Components .................................................. 8
   2.4 Tutorial ...................................................... 37
   2.5 Examples ..................................................... 114
   2.6 Benchmarks .................................................... 118
   2.7 Bibliography .................................................. 131

Bibliography 132

Index 135
AMGCL is a header-only C++ library for solving large sparse linear systems with algebraic multigrid (AMG) method. AMG is one of the most effective iterative methods for solution of equation systems arising, for example, from discretizing PDEs on unstructured grids [BrMH85], [Stue99], [TrOS01]. The method can be used as a black-box solver for various computational problems, since it does not require any information about the underlying geometry. AMG is often used not as a standalone solver but as a preconditioner within an iterative solver (e.g. Conjugate Gradients, BiCGStab, or GMRES).

The library has minimal dependencies, and provides both shared-memory and distributed memory (MPI) versions of the algorithms. The AMG hierarchy is constructed on a CPU and then is transferred into one of the provided backends. This allows for transparent acceleration of the solution phase with help of OpenCL, CUDA, or OpenMP technologies. Users may provide their own backends which enables tight integration between AMGCL and the user code.

The source code is available under liberal MIT license at https://github.com/ddemidov/amgcl.
CHAPTER 1

Referencing
2.1 Algebraic Multigrid

Here we outline the basic principles behind the Algebraic Multigrid (AMG) method [BrMH85], [Stue99]. Consider a system of linear algebraic equations in the form

$$Au = f$$

where $A$ is an $n \times n$ matrix. Multigrid methods are based on the recursive use of two-grid scheme, which combines

- *Relaxation*, or *smoothing iteration*: a simple iterative method such as Jacobi or Gauss-Seidel; and
- *Coarse grid correction*: solving residual equation on a coarser grid. Transfer between grids is described with transfer operators $P$ (prolongation or interpolation) and $R$ (restriction).

A setup phase of a generic algebraic multigrid (AMG) algorithm may be described as follows:

- Start with a system matrix $A_1 = A$.
- **While the matrix $A_i$ is too big to be solved directly:**
  1. Introduce prolongation operator $P_i$, and restriction operator $R_i$.
  2. Construct coarse system using Galerkin operator: $A_{i+1} = R_i A_i P_i$.
- Construct a direct solver for the coarsest system $A_L$.

Note that in order to construct the next level in the AMG hierarchy, we only need to define transfer operators $P$ and $R$. Also, the restriction operator is often chosen to be a transpose of the prolongation operator: $R = P^T$.

Having constructed the AMG hierarchy, we can use it to solve the system as follows:

- Start at the finest level with initial approximation $u_1 = u^0$.
- **Iterate until convergence (V-cycle):**
  - At each level of the grid hierarchy, finest-to-co coarsest:
    1. Apply a couple of smoothing iterations (pre-relaxation) to the current solution $u_i = S_i(A_i, f_i, u_i)$. 

2. Find residual \( e_i = f_i - A_i u_i \) and restrict it to the RHS on the coarser level: \( f_{i+1} = R_i e_i \).

- Solve the corasest system directly: \( u_L = A_L^{-1} f_L \).

- At each level of the grid hierarchy, coarsest-to-finest:
  1. Update the current solution with the interpolated solution from the coarser level: \( u_i = u_i + P_i u_{i+1} \).
  2. Apply a couple of smoothing iterations (post-relaxation) to the updated solution: \( u_i = S_i(A_i, f_i, u_i) \).

More often AMG is not used standalone, but as a preconditioner with an iterative Krylov subspace method. In this case single V-cycle is used as a preconditioning step.

So, in order to fully define an AMG method, we need to choose transfer operators \( P \) and \( R \), and smoother \( S \).

### 2.2 Design Principles

A lot of linear solver software packages are either developed in C or Fortran, or provide C-compatible application programming interface (API). The low-level API is stable and compatible with most of the programming languages. However, this also has some disadvantages: the fixed interfaces usually only support the predefined set of cases that the developers have thought of in advance. For an example, BLAS specification has separate sets of functions that deal with single, double, complex, or double complex precision values, but it is impossible to work with mixed precision inputs or with user-defined or third-party custom types. Another common drawback of large scientific packages is that users have to adopt the datatypes provided by the framework in order to work with it, which steepens the learning curve and introduces additional integration costs, such as the necessity to copy the data between various formats.

AMGCL is using modern C++ programming techniques in order to create flexible and efficient API. The users may easily extend the library or use it with their own datatypes. The following design principles are used throughout the code:

- **Policy-based design** [Alex00] of public library classes such as `amgcl::make_solver` or `amgcl::amg` allows the library users to compose their own customized version of the iterative solver and preconditioner from the provided components and easily extend and customize the library by providing their own implementation of the algorithms.

- Preference for free functions as opposed to member functions [Meye05], combined with partial template specialization allows to extend the library operations onto user-defined datatypes and to introduce new algorithmic components when required.

- The backend system of the library allows expressing the algorithms such as Krylov iterative solvers or multigrid relaxation methods in terms of generic parallel primitives which facilitates transparent acceleration of the solution phase with OpenMP, OpenCL, or CUDA technologies.

- One level below the backends are value types: AMGCL supports systems with scalar, complex, or block value types both in single and double precision. Arithmetic operations necessary for the library work may also be extended onto the user-defined types using template specialization.

#### 2.2.1 Policy-based design
Available solvers and preconditioners in AMGCL are composed by the library user from the provided components. For example, the most frequently used class template `amgcl::make_solver<P, S>` binds together an iterative solver S and a preconditioner P chosen by the user. To illustrate this, Listing 2.1 defines a conjugate gradient iterative solver preconditioned with an incomplete LU decomposition with zero fill-in in lines 2 to 10. The builtin backend (parallelized with OpenMP) with double precision is used both for the solver and the preconditioner. This approach allows the user not only to select any of the preconditioners/solvers provided by AMGCL, but also to use their own custom components, as long they conform to the generic AMGCL interface. In particular, the preconditioner class has to provide a constructor that takes the system matrix, the preconditioner parameters (defined as a subtype of the class, see below), and the backend parameters. The iterative solver constructor should take the size of the system matrix, the solver parameters, and the backend parameters.

This approach is used not only at the user-facing level of the library, but in any place where using interchangeable components makes sense. Lines 13 to 22 in Listing 2.1 show the declaration of GMRES iterative solver preconditioned with the algebraic multigrid (AMG). Smoothed aggregation is used as the AMG coarsening strategy, and diagonal sparse approximate inverse is used on each level of the multigrid hierarchy as a smoother. Similar to the solver and the preconditioner, the AMG components (coarsening and relaxation) are specified as template parameters and may be customized by the user.

Besides compile-time composition of the AMGCL algorithms described above, the library user may need to specify runtime parameters for the constructed algorithms. This is done with the `params` structure declared by each of the components as its subtype. Each parameter usually has a reasonable default value. When a class is composed...
from several components, it includes the parameters of its dependencies into its own `params` struct. This allows to provide a unified interface to the parameters of various AMGCL algorithms. Listing 2.2 shows how the parameters are declared for the `amgcl::make_solver<P,S>` class. Listing 2.3 shows an example of how the parameters for the preconditioned GMRES solver from Listing 2.1 may be specified. Namely, the number of the GMRES iterations before restart is set to 50, the relative residual threshold is set to $10^{-6}$, and the strong connectivity threshold $\epsilon_{str}$ for the smoothed aggregation is set to $10^{-3}$. The rest of the parameters are left with their default values.

Listing 2.3: Setting parameters for AMGCL components

```cpp
// Set the solver parameters
Solver2::params prm;
prm.solver.M = 50;
prm.solver.tol = 1e-6;
prm.precond.coarsening.aggr.eps_strong = 1e-3;

// Instantiate the solver
Solver2 S(A, prm);
```

### 2.2.2 Free functions and partial template specialization

Using free functions as opposed to class methods allows to decouple the library functionality from specific classes and enables support for third-party datatypes within the library [Meye05]. Moving the implementation from the free function into a struct template specialization provides more control over the mapping between the input datatype and the specific specific version of the algorithm. For example, constructors of AMGCL classes may accept an arbitrary datatype as input matrix, as long as the implementations of several basic functions supporting the datatype have been provided. Some of the free functions that need to be implemented are `amgcl::backend::rows(A)`, `amgcl::backend::cols(A)` (returning the number of rows and columns for the matrix), or `amgcl::backend::row_begin(A,i)` (returning iterator over the nonzero values for the matrix row). Listing 2.4 shows an implementation of `amgcl::backend::rows()` function for the case when the input matrix is specified as a `std::tuple(n,ptr,col,val)` of matrix size `n`, pointer vector `ptr` containing row offsets into the column index and value vectors, and the column index and values vectors `col` and `val` for the nonzero matrix entries. AMGCL provides adapters for several common input matrix formats, such as `Eigen::SparseMatrix` from Eigen, `Epetra_CrsMatrix` from Trilinos Epetra, and it is easy to adapt a user-defined datatype.

Listing 2.4: Implementation of `amgcl::backend::rows()` free function for the CRS tuple

```cpp
// Generic implementation of the rows() function.  
// Works as long as the matrix type provides rows() member function.
template <class Matrix, class Enable = void>
struct rows_impl {
    static size_t get(const Matrix &A) {
        return A.rows();
    }
};

// Returns the number of rows in a matrix.
template <class Matrix>
size_t rows(const Matrix &matrix) {
    return rows_impl<Matrix>::get(matrix);
}

// Specialization of rows_impl template for a CRS tuple.
template <typename N, typename PRng, typename CRng, typename VRng>
(continues on next page)
2.2.3 Backends

A backend in AMGCL is a class that binds datatypes like matrix and vector with parallel primitives like matrix-vector product, linear combination of vectors, or inner product computation. The backend system is implemented using the free functions combined with template specialization approach from the previous section, which decouples the implementation of common parallel primitives from the specific datatypes used in the supported backends. This allows to adopt third-party or user-defined datatypes for use within AMGCL without any modification. For example, in order to switch to the CUDA backend in Listing 2.1, we just need to replace `amgcl::backend::builtin<double>` with `amgcl::backend::cuda<double>`.

Algorithm setup in AMGCL is performed using internal data structures. As soon as the setup is completed, the necessary objects (mostly matrices and vectors) are transferred to the backend datatypes. Solution phase of the algorithms is expressed in terms of the predefined parallel primitives which makes it possible to switch parallelization technology (such as OpenMP, CUDA, or OpenCL) simply by changing the backend template parameter of the algorithm. For example, the residual norm $\|f - Ax\|$ in AMGCL is computed using `amgcl::backend::residual()` and `amgcl::backend::inner_product()` primitives:

```cpp
backend::residual(f, A, x, r);
auto e = sqrt(backend::inner_product(r, r));
```

2.2.4 Value types

Value type concept allows to generalize AMGCL algorithms onto complex or non-scalar systems. A value type defines a number of overloads for common math operations, and is used as a template parameter for a backend. Most often, a value type is simply a builtin `double` or `float` atomic value, but it is also possible to use small statically sized matrices when the system matrix has a block structure, which may decrease the setup time and the overall memory footprint, increase cache locality, or improve convergence ratio.

Value types are used during both the setup and the solution phases. Common value type operations are defined in `amgcl::math` namespace, similar to how backend operations are defined in `amgcl::backend`. Examples of such operations are `amgcl::math::norm()` or `amgcl::math::adjoint()`. Arithmetic operations like multiplication or addition are defined as operator overloads. AMGCL algorithms at the lowest level are expressed in terms of the value type interface, which makes it possible to switch precision of the algorithms, or move to complex values, simply by adjusting template parameter of the selected backend.

The generic implementation of the value type operations also makes it possible to use efficient third party implementations of the block value arithmetics. For example, using statically sized Eigen matrices instead of builtin `amgcl::static_matrix` as block value type may improve performance in case of relatively large blocks, since the Eigen library supports SIMD vectorization.

2.2.5 Runtime interface

The compile-time configuration of AMGCL solvers is not always convenient, especially if the solvers are used inside a software package or another library. The runtime interface allows to shift some of the configuration decisions to
runtime. The classes inside `amgcl::runtime` namespace correspond to their compile-time alternatives, but the only template parameter you need to specify is the backend.

Since there is no way to know the parameter structure at compile time, the runtime classes accept parameters only in form of `boost::property_tree::ptree`. The actual components of the method are set through the parameter tree as well. For example, the solver above could be constructed at runtime in the following way:

```cpp
#include <amgcl/backend/builtin.hpp>
#include <amgcl/make_solver.hpp>
#include <amgcl/amg.hpp>
#include <amgcl/coarsening/runtime.hpp>
#include <amgcl/relaxation/runtime.hpp>
#include <amgcl/solver/runtime.hpp>

typedef amgcl::backend::builtin<double> Backend;

typedef amgcl::make_solver<
    amgcl::amg<
        Backend,
        amgcl::runtime::coarsening::wrapper,
        amgcl::runtime::relaxation::wrapper
    >,
    amgcl::runtime::solver::wrapper<Backend>
> Solver;

boost::property_tree::ptree prm;
prm.put("solver.type", "bicgstab");
prm.put("solver.tol", 1e-3);
prm.put("solver.maxiter", 10);
prm.put("precond.coarsening.type", "smoothed_aggregation");
prm.put("precond.relax.type", "spai0");
Solver solve( std::tie(n, ptr, col, val), prm );
```

### 2.3 Components

AMGCL defines the following algorithmic components:

#### 2.3.1 Backends

A backend in AMGCL is a class that binds datatypes like matrix and vector with parallel primitives like matrix-vector product, linear combination of vectors, or inner product computation. The backend system is implemented using free functions and partial template specializations, which allows to decouple the implementation of common parallel primitives from the specific datatypes used in the supported backends. This makes it possible to adopt third-party or user-defined datatypes for use within AMGCL without any modification of the core library code.

Algorithm setup in AMGCL is performed using internal data structures. As soon as the setup is completed, the necessary objects (mostly matrices and vectors) are transferred to the backend datatypes. Solution phase of the algorithms is expressed in terms of the predefined parallel primitives which makes it possible to switch parallelization technology (such as OpenMP, CUDA, or OpenCL) simply by changing the backend template parameter of the algorithm. For example, the norm of the residual $\epsilon = \|f - Ax\|$ in AMGCL is computed with `amgcl::backend::residual()` and `amgcl::backend::inner_product()` primitives:
The backends currently supported by AMGCL are listed below.

**OpenMP (builtin) backend**

template<class ValueType>
class amgcl::backend::builtin
Include <amgcl/backend/builtin.hpp>.

This is the builtin backend that does not have any external dependencies and uses OpenMP-parallelization for its primitives. As with any backend in AMGCL, it is defined with a value type template parameter, which specifies the type of the system matrix elements. The backend is also used internally by AMGCL during construction phase, and so moving the constructed datatypes (matrices and vectors) to the backend has no overhead. The backend has no parameters (the params subtype is an empty struct).

class params

**NVIDIA CUDA backend**

template<class ValueType>
class amgcl::backend::cuda
Include <amgcl/backend/cuda.hpp>.

The backend uses the NVIDIA CUDA technology for the parallelization of its primitives. It depends on the Thrust and cuSPARSE libraries. The code using the backend has to be compiled with NVIDIA's nvcc compiler. The user needs to initialize the cuSPARSE library with a call to the cusparseCreate() function and pass the returned handle to AMGCL in the backend parameters.

class params

The backend parameters

cusparseHandle_t cusparse_handle

   cuSPARSE handle created with the cusparseCreate() function.

**VexCL backend**

template<class ValueType>
class amgcl::backend::vexcl
Include <amgcl/backend/vexcl.hpp>.

The backend uses the VexCL library for the implementation of its primitives. VexCL provides OpenMP, OpenCL, or CUDA parallelization, selected at compile time with a preprocessor definition. The user has to initialize the VexCL context and pass it to AMGCL via the backend parameter.

class params

The backend parameters

std::vector<vex::backend::command_queue> q

   VexCL command queues identifying the compute devices in the compute context.

bool fast_matrix_setup = true

   Transform the CSR matrices into the internal VexCL format on the GPU. This is faster, but temporarily requires more memory on the GPU.

2.3. Components
ViennaCL backend

template<class Matrix>
class amgcl::backend::viennacl
  Include <amgcl/backend/viennacl.hpp>.

  The backend uses the ViennaCL library for the implementation of its primitives. ViennaCL is a free open-source linear algebra library for computations on many-core architectures (GPUs, MIC) and multi-core CPUs. The library is written in C++ and supports CUDA, OpenCL, and OpenMP (including switches at runtime). The template parameter for the backend specifies ViennaCL matrix class to use. Possible choices are viennacl::compressed_matrix<T>, viennacl::ell_matrix<T>, and viennacl::hyb_matrix<T>. The backend has no runtime parameters.

class params
  The backend parameters

Eigen backend

template<class ValueType>
class amgcl::backend::eigen
  Include <amgcl/backend/eigen.hpp>.

  The backend uses Eigen library datatypes for implementation of its primitives. It could be useful in case the user already works with the Eigen library, for example, to assemble the linear system to be solved with AMGCL. AMGCL also provides an Eigen matrix adapter, so that Eigen matrices may be transparently used with AMGCL solvers.

class params
  The backend parameters

Blaze backend

template<class ValueType>
class amgcl::backend::blaze
  Include <amgcl/backend/blaze.hpp>.

  The backend uses Blaze library datatypes for implementation of its primitives. It could be useful in case the user already works with the Blaze library, for example, to assemble the linear system to be solved with AMGCL.

class params
  The backend parameters

2.3.2 Value Types

The value type concept allows to generalize AMGCL algorithms for the systems with complex or non-scalar coefficients. A value type defines a number of overloads for common math operations, and is used as a template parameter for a backend. Most often, a value type is simply a builtin double or float atomic value, but it is also possible to use std::complex<T>, or small statically sized matrices when the system matrix has a block structure. The latter may decrease the setup time and the overall memory footprint, increase cache locality, or improve convergence ratio.

Value types are used during both the setup and the solution phases. Common value type operations are defined in amgcl::math namespace, similar to how backend operations are defined in amgcl::backend. Examples of such operations are amgcl::math::norm() or amgcl::math::adjoint(). Arithmetic operations like multiplication or addition are defined as operator overloads. AMGCL algorithms at the lowest level are expressed in
terms of the value type interface, which makes it possible to switch precision of the algorithms, or move to complex values simply by adjusting the template parameter of the selected backend.

The generic implementation of the value type operations also makes it possible to use efficient third party implementations of the block value arithmetics. For example, using statically sized Eigen matrices instead of the builtin `amgcl::static_matrix` as block value type may improve performance in case of relatively large blocks, since the Eigen library supports SIMD vectorization.

**Scalar values**

All backends support `float` and `double` as value type. CPU-based backends (e.g. `amgcl::backend::builtin`) may also use `long double`. The use of non-trivial value types depends on whether the value type is supported by the selected backend.

**Complex values**

**Data type:** `std::complex<T>`

**Include:**

- `<amgcl/value_type/complex.hpp>`

**Supported by backends:**

- `amgcl::backend::builtin`
- `amgcl::backend::vexcl`
- `amgcl::backend::eigen`
- `amgcl::backend::blaze`

**Statically sized matrices**

**Data type:** `amgcl::static_matrix<T,N,N>`

**Include:**

- `<amgcl/value_type/static_matrix.hpp>`
- `<amgcl/backend/vexcl_static_matrix.hpp>` (in case VexCL is used as the backend)

**Supported by backends:**

- `amgcl::backend::builtin`
- `amgcl::backend::vexcl`
Eigen static matrices

Data type: Eigen::Matrix<T,N,N>

Include:

- `<amgcl/value_type/eigen.hpp>`

Supported by backends:

- `amgcl::backend::builtin`
- `amgcl::backend::eigen`

2.3.3 Matrix Adapters

A matrix adapter allows AMGCL to construct a solver from some common matrix formats. Internally, the CRS format is used, but it is easy to adapt any matrix format that allows row-wise iteration over its non-zero elements.

Tuple of CRS arrays

Include `<amgcl/adapter/crs_tuple.hpp>`

It is possible to use a `std::tuple` of CRS arrays as input matrix for any of the AMGCL algorithms. The CRS arrays may be stored either as STL containers:

```cpp
std::vector<int>   ptr;
std::vector<int>   col;
std::vector<double> val;
Solver S( std::tie(n, ptr, col, val) );
```

or as `amgcl::iterator_range`, which makes it possible to adapt raw pointers:

```cpp
int   *ptr;
int   *col;
double *val;
Solver S( std::make_tuple(n, 
                   amgcl::make_iterator_range(ptr, ptr + n + 1),
                   amgcl::make_iterator_range(col, col + ptr[n]),
                   amgcl::make_iterator_range(val, val + ptr[n])
                   ) );
```

Zero copy

Include `<amgcl/adapter/zero_copy.hpp>`

`template<class Ptr, class Col, class Val>`
std::shared_ptr<amgcl::backend::crs<Val>> zero_copy (size_t n, const Ptr *ptr, const Col *col, const Val *val)

Returns a shared pointer to the sparse matrix in internal AMGCL format. The matrix may be directly used for constructing AMGCL algorithms. Ptr and Col have to be 64bit integral datatypes (signed or unsigned). In case the amgcl::backend::builtin backend is used, no data will be copied from the CRS arrays, so it is the user’s responsibility to make sure the pointers are alive until the AMGCL algorithm is destroyed.

### Block matrix

Include `<amgcl/adapter/block_matrix.hpp>`

```cpp
template<class BlockType, class Matrix>
block_matrix_adapter<Matrix, BlockType> block_matrix (const Matrix &A)
```

Converts scalar-valued matrix to a block-valued one on the fly. The adapter allows to iterate the rows of the scalar-valued matrix as if the matrix was stored using the block values. The rows of the input matrix have to be sorted column-wise.

### Scaled system

Include `<amgcl/adapter/block_matrix.hpp>`

```cpp
template<class Backend, class Matrix>
auto scaled_diagonal (const Matrix &A, const typename Backend::params &bprm = typename Backend::params())
```

Returns a scaler object that may be used to scale the system so that the matrix has unit diagonal:

\[ A_s = D^{1/2} AD^{1/2} \]

where \( D \) is the matrix diagonal. This keeps the matrix symmetrical. The RHS also needs to be scaled, and the solution of the system has to be postprocessed:

\[ D^{1/2} AD^{1/2} y = D^{1/2} b, \quad x = D^{1/2} y \]

The scaler object may be used to scale both the matrix:

```cpp
auto A = std::tie(rows, ptr, col, val);
auto scale = amgcl::adapter::scale_diagonal<Backend>(A, bprm);
```

```cpp
// Setup solver
Solver solve(scale.matrix(A), prm, bprm);
```

and the RHS:

```cpp
// option 1: rhs is untouched
solve(*scale.rhs(b), x);
```

```cpp
// option 2: rhs is prescaled in-place
scale(b);
solve(b, x);
```

The solution vector has to be postprocessed afterwards:

```cpp
// postprocess the solution in-place:
scale(x);
```
Reordered system

Include `<amgcl/adapter/reorder.hpp>`

```cpp
template<class ordering = amgcl::reorder::cuthill_mckee<false>>
class amgcl::adapter::reorder

Reorders the matrix to reduce its bandwidth. Example:

```cpp
// Prepare the reordering:
amgcl::adapter::reorder<> perm(A);

// Create the solver using the reordered matrix:
Solver solve(perm(A), prm);

// Reorder the RHS and solve the system:
solve(perm(rhs), x_ord);

// Postprocess the solution vector to get the original ordering:
perm.inverse(x_ord, x);
```

Eigen matrix

Simply including `<amgcl/adapter/eigen.hpp>` allows to use Eigen sparse matrices in AMGCL algorithm constructors. The Eigen matrix has to be stored with the RowMajor ordering.

Epetra matrix

Including `<amgcl/adapter/epetra.hpp>` allows to use Trilinos Epetra distributed sparse matrices in AMGCL MPI algorithm constructors.

uBlas matrix

Including `<amgcl/adapter/ublas.hpp>` allows to use uBlas sparse matrices in AMGCL algorithm constructors, and directly use uBlas vectors as the RHS and solution arrays.

2.3.4 Iterative Solvers

An iterative solver is a Krylov subspace method that may be combined with a preconditioner in order to solve the linear system.

All iterative solvers in AMGCL have two template parameters, Backend and InnerProduct. The Backend template parameter specifies the backend to target, and the InnerProduct parameter is used to select the implementation of the inner product to use with the solver. The correct implementation should be automatically selected by the library depending on whether the solver is used in a shared or distributed memory setting.

All solvers provide similar interface described below:

```cpp
class constructor (size_t n, const params &prm = params(), const backend_params &bprm = backend_params())
```

The solver constructor. Takes the size of the system to solve, the solver parameters and the backend parameters.
std::tuple<size_t, scalar_type> operator() (const Matrix &A, const Precond &P, const VectorRHS &rhs, const VectorX &x)

Computes the solution for the given system matrix A and the right-hand side rhs.

Returns the number of iterations made and the achieved relative residual as a std::tuple<size_t, scalar_type>. The solution vector x provides initial approximation on input and holds the computed solution on output.

template<class Precond, class VectorRHS, class VectorX>
std::tuple<size_t, scalar_type> operator() (const Precond &P, const VectorRHS &rhs, const VectorX &x)

Computes the solution for the given right-hand side rhs. The matrix that was used to create the preconditioner P is used as the system matrix.

Returns the number of iterations made and the achieved relative residual as a std::tuple<size_t, scalar_type>. The solution vector x provides initial approximation on input and holds the computed solution on output.

AMGCL implementats the following iterative solvers:

CG

template<class Backend, class InnerProduct = amgcl::detail::default_inner_product>
class amgcl::solver::cg

Include <amgcl/solver/cg.hpp>

The Conjugate Gradient method is an effective method for symmetric positive definite systems. It is probably the oldest and best known of the nonstationary methods [Barr94], [Saad03].

typedef typename Backend::value_type value_type

The value type of the system matrix

typedef typename amgcl::math::scalar_of<value_type>::type scalar_type

The scalar type corresponding to the value type. For example, when the value type is std::complex<double>, then the scalar type is double.

class params

The solver parameters.

size_t maxiter = 100

The maximum number of iterations

scalar_type tol = 1e-8

Target relative residual error \( \varepsilon = \frac{||f-Ax||}{||f||} \)

scalar_type abstol = std::numeric_limits<scalar_type>::min()

Target absolute residual error \( \varepsilon = ||f - Ax|| \)

bool ns_search = false

Ignore the trivial solution \( x=0 \) when the RHS is zero. Useful when searching for the null-space vectors of the system.

bool verbose = false

Output the current iteration number and relative residual during solution.
### BiCGStab

**template<class Backend, class InnerProduct = amgcl::detail::default_inner_product>**

class amgcl::solver::bicgstab

**Include <amgcl/solver/bicgstab.hpp>**

The BiConjugate Gradient Stabilized method (BiCGStab) was developed to solve nonsymmetric linear systems while avoiding the often irregular convergence patterns of the Conjugate Gradient Squared method [Barr94].

**typedef typename Backend::value_type value_type**

The value type of the system matrix

**typedef typename amgcl::math::scalar_of<value_type>::type scalar_type**

The scalar type corresponding to the value type. For example, when the value type is `std::complex<double>`, then the scalar type is `double`.

**class params**

The solver parameters.

bool `check_after` = false  
Always do at least one iteration

`amgcl::preconditioner::side::type pside = amgcl::preconditioner::side::right`  
Preconditioner kind (left/right)

`size_t maxiter = 100`  
The maximum number of iterations

`scalar_type tol = 1e-8`  
Target relative residual error $\varepsilon = \frac{||f - Ax||}{||f||}$

`scalar_type abstol = std::numeric_limits<scalar_type>::min()`  
Target absolute residual error $\varepsilon = ||f - Ax||$

bool `ns_search` = false  
Ignore the trivial solution $x=0$ when the RHS is zero. Useful when searching for the null-space vectors of the system.

bool `verbose` = false  
Output the current iteration number and relative residual during solution.

### BiCGStab(L)

**template<class Backend, class InnerProduct = amgcl::detail::default_inner_product>**

class amgcl::solver::bicgstabl

**Include <amgcl/solver/bicgstabl.hpp>**

This is a generalization of the BiCGStab method [SlDi93], [Fokk96]. For $L = 1$, this algorithm coincides with BiCGStab. In some situations it may be profitable to take $L > 2$. Although the steps of BiCGStab(L) are more expensive for larger L, numerical experiments indicate that, in certain situations, due to a faster convergence, for instance, BiCGStab(4) performs better than BiCGStab(2).

**typedef typename Backend::value_type value_type**

The value type of the system matrix

---

2.3. Components 16
typedef typename amgcl::math::scalar_of<value_type>::type scalar_type
The scalar type corresponding to the value type. For example, when the value type is std::complex<double>, then the scalar type is double.

class params
The solver parameters.

int L = 2
The order of the method

scalar_type delta = 0
Threshold used to decide when to refresh computed residuals.

bool convex = true
Use a convex function of the MinRes and OR polynomials after the BiCG step instead of default MinRes

amgcl::preconditioner::side::type pside = amgcl::preconditioner::side::right
Preconditioner kind (left/right)

size_t maxiter = 100
The maximum number of iterations

scalar_type tol = 1e-8
Target relative residual error $\varepsilon = \frac{||f - Ax||}{||f||}$

scalar_type abstol = std::numeric_limits<scalar_type>::min()
Target absolute residual error $\varepsilon = ||f - Ax||$

bool ns_search = false
Ignore the trivial solution $x=0$ when the RHS is zero. Useful when searching for the null-space vectors of the system.

bool verbose = false
Output the current iteration number and relative residual during solution.

GMRES

template<class Backend, class InnerProduct = amgcl::detail::default_inner_product>
class amgcl::solver::gmres

Include <amgcl/solver/gmres.hpp>

The Generalized Minimal Residual method is an extension of MINRES (which is only applicable to symmetric systems) to unsymmetric systems. Like MINRES, it generates a sequence of orthogonal vectors, but in the absence of symmetry this can no longer be done with short recurrences; instead, all previously computed vectors in the orthogonal sequence have to be retained. For this reason, “restarted” versions of the method are used [Barr94].

typedef typename Backend::value_type value_type
The value type of the system matrix

typedef typename amgcl::math::scalar_of<value_type>::type scalar_type
The scalar type corresponding to the value type. For example, when the value type is std::complex<double>, then the scalar type is double.

class params
The solver parameters.
int $M = 30$
   The number of iterations before restart

$\text{amgcl::preconditioner::side::type } \text{pside} = \text{amgcl::preconditioner::side::right}$
   Preconditioner kind (left/right)

size_t $\text{maxiter} = 100$
   The maximum number of iterations

$\text{scalar_type } \text{tol} = 1e-8$
   Target relative residual error $\varepsilon = \frac{||f - Ax||}{||f||}$

$\text{scalar_type } \text{abstol} = \text{std::numeric_limits<scalar_type>::min()}$
   Target absolute residual error $\varepsilon = ||f - Ax||$

bool $\text{ns_search} = \text{false}$
   Ignore the trivial solution $x=0$ when the RHS is zero. Useful when searching for the null-space vectors of the system.

bool $\text{verbose} = \text{false}$
   Output the current iteration number and relative residual during solution.

"Loose" GMRES (LGMRES)

template<class $\text{Backend}$, class $\text{InnerProduct} = \text{amgcl::detail::default_inner_product}$>
$\text{class } \text{amgcl::solver::lgmres}$

#include <amgcl/solver/lgmres.hpp>

The residual vectors at the end of each restart cycle of restarted GMRES often alternate direction in a cyclic fashion, thereby slowing convergence. LGMRES is an implementation of a technique for accelerating the convergence of restarted GMRES by disrupting this alternating pattern. The new algorithm resembles a full conjugate gradient method with polynomial preconditioning, and its implementation requires minimal changes to the standard restarted GMRES algorithm [BaJM05].

typedef typename $\text{Backend}::\text{value_type}$ value_type
   The value type of the system matrix

typedef typename $\text{amgcl::math::scalar_of<value_type>::type}$ scalar_type
   The scalar type corresponding to the value type. For example, when the value type is std::complex<double>, then the scalar type is double.

class $\text{params}$
   The solver parameters.

unsigned $K = 3$
   Number of vectors to carry between inner GMRES iterations. According to [BaJM05], good values are in the range of 1-3. However, if you want to use the additional vectors to accelerate solving multiple similar problems, larger values may be beneficial.

bool $\text{always_reset} = \text{true}$
   Reset augmented vectors between solves. If the solver is used to repeatedly solve similar problems, then keeping the augmented vectors between solves may speed up subsequent solves. This flag, when set, resets the augmented vectors at the beginning of each solve.

int $M = 30$
   The number of iterations before restart

2.3. Components
amgcl::preconditioner::side::type pside = amgcl::preconditioner::side::right
Preconditioner kind (left/right)
size_t maxiter = 100
The maximum number of iterations
scalar_type tol = 1e-8
Target relative residual error \( \varepsilon = \frac{|f - Ax|}{|f|} \)
scalar_type abstol = std::numeric_limits<scalar_type>::min()
Target absolute residual error \( \varepsilon = ||f - Ax|| \)
bool ns_search = false
Ignore the trivial solution \( x=0 \) when the RHS is zero. Useful when searching for the null-space vectors of the system.
bool verbose = false
Output the current iteration number and relative residual during solution.

Flexible GMRES (FGMRES)

template<class Backend, class InnerProduct = amgcl::detail::default_inner_product>
class amgcl::solver::fgmres

Include <amgcl/solver/fgmres.hpp>

Often, the application of the preconditioner \( P \) is a result of some unspecified computation, possibly another iterative process. In such cases, it may well happen that \( P \) is not a constant operator. The preconditioned iterative solvers may not converge if \( P \) is not constant. There are a number of variants of iterative procedures developed in the literature that can accommodate variations in the preconditioner, i.e., that allow the preconditioner to vary from step to step. Such iterative procedures are called “flexible” iterations. The method implements flexible variant of the GMRES algorithm [Saad03].

typedef typename Backend::value_type value_type
The value type of the system matrix

typedef typename amgcl::math::scalar_of<value_type>::type scalar_type
The scalar type corresponding to the value type. For example, when the value type is std::complex<double>, then the scalar type is double.

class params
The solver parameters.

int M = 30
The number of iterations before restart
size_t maxiter = 100
The maximum number of iterations
scalar_type tol = 1e-8
Target relative residual error \( \varepsilon = \frac{|f - Ax|}{|f|} \)
scalar_type abstol = std::numeric_limits<scalar_type>::min()
Target absolute residual error \( \varepsilon = ||f - Ax|| \)
bool ns_search = false
Ignore the trivial solution \( x=0 \) when the RHS is zero. Useful when searching for the null-space vectors of the system.

2.3. Components
bool verbose = false
Output the current iteration number and relative residual during solution.

**IDR(s)**

```cpp
template<class Backend, class InnerProduct = amgcl::detail::default_inner_product>
class amgcl::solver::idrs
```

*Include <amgcl/solver/idrs.hpp>*

This is a very stable and efficient IDR(s) variant as described in [GiSo11]. The Induced Dimension Reduction method, IDR(s), is a robust and efficient short-recurrence Krylov subspace method for solving large nonsymmetric systems of linear equations.

IDR(s) compared to BI-CGSTAB/BiCGStab:

- Faster.
- More robust.
- More flexible.

```cpp
typedef typename Backend::value_type value_type
The value type of the system matrix
```

```cpp
typedef typename amgcl::math::scalar_of<
    value_type
>::type
scalar_type
```

The scalar type corresponding to the value type. For example, when the value type is `std::complex<double>`, then the scalar type is `double`.

**class params**
The solver parameters.

```cpp
unsigned s = 4
Dimension of the shadow space in IDR(s).
```

```cpp
scalar_type omega = 0.7
Computation of omega.
```

- If omega = 0, a standard minimum residual step is performed.
- If omega > 0, omega is increased if the cosine of the angle between Ar and r < omega.

```cpp
bool smoothing = false
Specifies if residual smoothing must be applied.
```

```cpp
bool replacement = false
Residual replacement. Determines the residual replacement strategy. If true, the recursively computed residual is replaced by the true residual.
```

```cpp
size_t maxiter = 100
The maximum number of iterations
```

```cpp
scalar_type tol = 1e-8
Target relative residual error \( \varepsilon = \frac{||f - Ax||}{||f||} \)
```

```cpp
scalar_type abstol = std::numeric_limits<scalar_type>::min()
Target absolute residual error \( \varepsilon = ||f - Ax|| \)
```

```cpp
bool ns_search = false
Ignore the trivial solution \( x=0 \) when the RHS is zero. Useful when searching for the null-space vectors of the system.
```

2.3. Components
bool verbose = false
Output the current iteration number and relative residual during solution.

Richardson iteration

template<class Backend, class InnerProduct = amgcl::detail::default_inner_product>
class amgcl::solver::richardson

Include <amgcl/solver/richardson.hpp>

The preconditioned Richardson iterative method

\[ x_{i+1} = x_i + \omega P(f - Ax_i) \]

typedef typename Backend::value_type value_type
The value type of the system matrix

typedef typename amgcl::math::scalar_of<value_type>::type scalar_type
The scalar type corresponding to the value type. For example, when the value type is
std::complex<double>, then the scalar type is double.

class params
The solver parameters.

scalar_type damping = 1.0
The damping factor \( \omega \)

size_t maxiter = 100
The maximum number of iterations

scalar_type tol = 1e-8
Target relative residual error \( \varepsilon = \frac{||f - Ax||}{||f||} \)

scalar_type abstol = std::numeric_limits<scalar_type>::min()
Target absolute residual error \( \varepsilon = ||f - Ax|| \)

bool ns_search = false
Ignore the trivial solution \( x=0 \) when the RHS is zero. Useful when searching for the null-space
vectors of the system.

bool verbose = false
Output the current iteration number and relative residual during solution.

PreOnly

template<class Backend, class InnerProduct = amgcl::detail::default_inner_product>
class amgcl::solver::preonly

Include <amgcl/solver/preonly.hpp>

Only apply the preconditioner once. This is not very useful as a standalone solver, but may be used in composite
preconditioners as a nested solver, so that the composite preconditioner itself remains linear and may be used
with a non-flexible iterative solver.
typedef typename \texttt{Backend}::value\_type \texttt{value\_type}  
The value type of the system matrix

typedef typename amgcl::math::scalar\_of<\texttt{value\_type}>\::type \texttt{scalar\_type}  
The scalar type corresponding to the value type. For example, when the value type is \texttt{std::complex<double>}, then the scalar type is \texttt{double}.

class \texttt{params}  
The solver parameters.

2.3.5 Preconditioners

Aside from the AMG, AMGCL implements preconditioners for some common problem types. For example, there is a Schur complement pressure correction preconditioner for Navie-Stokes type problems, or CPR preconditioner for reservoir simulations. Also, it is possible to use single level relaxation method as a preconditioner.

General preconditioners

These preconditioners do not take the origin or structure of matrix into account, and may be useful both on their own, as well as building blocks for the composite preconditioners.

AMG

template<class \texttt{Backend}, template<class> class \texttt{Coarsening}, template<class> class \texttt{Relax}>  
class amgcl::amg

Include \texttt{<amgcl/amg.hpp>}

AMG is one the most effective methods for the solution of large sparse unstructured systems of equations, arising, for example, from discretization of PDEs on unstructured grids [TrOS01]. The method may be used as a black-box solver for various computational problems, since it does not require any information about the underlying geometry.

The three template parameters allow the user to select the exact components of the method:

- The \texttt{Backend} to transfer the constructed hierarchy to,
- The \texttt{Coarsening} strategy for the hierarchy construction, and
- The \texttt{Relaxation} scheme (smoother to use during the solution phase).

typedef typename \texttt{Backend}::params \texttt{backend\_params}  
The backend parameters

typedef typename \texttt{Backend}::value\_type \texttt{value\_type}  
The value type of the system matrix

typedef typename amgcl::math::scalar\_of<\texttt{value\_type}>\::type \texttt{scalar\_type}  
The scalar type corresponding to the value type. For example, when the value type is \texttt{std::complex<double>}, then the scalar type is \texttt{double}.

typedef \texttt{Coarsening<Backend>} \texttt{coarsening\_type}  
The coarsening class instantiated for the specified backend

typedef \texttt{Relax<Backend>} \texttt{relax\_type}  
The relaxation class instantiated for the specified backend
class params
The AMG parameters. The coarsening and the relaxation parameters are encapsulated as part of the AMG parameters.

typedef typename coarsening_type::params coarsening_params
  The type of the coarsening parameters

typedef typename relax_type::params relax_params
  The type of the relaxation parameters

coarsening_params coarsening
  The coarsening parameters

relax_params relax
  The relaxation parameters

unsigned coarse_enough = Backend::direct_solver::coarse_enough()
  Specifies when a hierarchy level is coarse enough to be solved directly. If the number of variables at the level is lower than this threshold, then the hierarchy construction is stopped and the linear system is solved directly at this level. The default value comes from the direct solver class defined as part of the backend.

bool direct_coarse = true
  Use direct solver at the coarsest level. When set, the coarsest level is solved with a direct solver. Otherwise a smoother is used as a solver. This may be useful when the system is singular, but is still possible to solve with an iterative solver.

unsigned max_levels = std::numeric_limits<unsigned>::max()
  The maximum number of levels. If this number is reached even when the size of the last level is greater than coarse_enough, then the hierarchy construction is stopped. The coarsest level will not be solved directly, but will use a smoother.

unsigned npre = 1
  The number of pre-relaxations.

unsigned npost = 1
  The number of post-relaxations.

unsigned ncycle = 1
  The shape of AMG cycle (1 for V-cycle, 2 for W-cycle, etc).

unsigned pre_cycles = 1
  The number of cycles to make as part of preconditioning.

Single-level relaxation

template<class Backend, template<class> class Relax>
class amgcl::relaxation::as_preconditioner

Include <amgcl/relaxation/as_preconditioner.hpp>

Allows to use a relaxation method as a standalone preconditioner.

Relax<backend> smoother;
  The relaxation class instantiated for the specified backend

typedef typename smoother::params params
  The relaxation params are inherited as the parameters for the preconditioner
Dummy

template<class Backend>
class amgcl::preconditioner::dummy

Include <amgcl/preconditioner/dummy.hpp>

The dummy preconditioner, equivalent to an identity matrix. May be used to test the convergence of unpreconditioned iterative solvers.

class params

There are no parameters

Composite preconditioners

The preconditioners in this section take the into account the block structure of the system and properties of the individual blocks. Most often the preconditioners are used for the solution of saddle point or Stokes-like systems, where the system matrix may be represented in the following form:

\[
\begin{pmatrix}
A & B_1^T \\
B_2 & C
\end{pmatrix}
\begin{pmatrix}
u \\
p
\end{pmatrix} = 
\begin{pmatrix}
b_u \\
b_p
\end{pmatrix}
\]

(2.1)

CPR

template<class PPrecond, class SPrecond>
class amgcl::preconditioner::cpr

Include <amgcl/preconditioner/cpr.hpp>

The Constrained Pressure Residual (CPR) preconditioner [Stue07]. The CPR preconditioners are based on the idea that coupled system solutions are mainly determined by the solution of their elliptic components (i.e., pressure). Thus, the procedure consists of extracting and accurately solving pressure subsystems. Residuals associated with this solution are corrected with an additional preconditioning step that recovers part of the global information contained in the original system.

The template parameters PPrecond and SPrecond for the CPR preconditioner specify which preconditioner to use with the pressure subblock (the \(C\) matrix in (2.1)), and with the complete system.

The system matrix should be ordered by grid nodes, so that the pressure and saturation/concentration unknowns belonging to the same grid node are compactly located together in the vector of unknowns. The pressure should be the first unknown in the block of unknowns associated with a grid node.

class params

The CPR preconditioner parameters

typedef typename SPrecond::value_type value_type
The value type of the system matrix

typedef typename PPrecond::params pprecond_params
The type of the pressure preconditioner parameters

typedef typename SPrecond::params sprecond_params
The type of the global preconditioner parameters
The pressure preconditioner parameters

The global preconditioner parameters

```
int block_size = 2
```

The number of unknowns associated with each grid node. The default value is 2 when the system matrix has scalar value type. Otherwise, the block size of the system matrix value type is used.

```
size_t active_rows = 0
```

When non-zero, only unknowns below this number are considered to be pressure. May be used when a system matrix contains unstructured tail block (for example, the unknowns associated with wells).

## CPR (DRS)

```
template<class PPrecond, class SPrecond>
class amgcl::preconditioner::cpr_drs
```

Include `<amgcl/precondition/cpr.hpp>`

The Constrained Pressure Residual (CPR) preconditioner with weighted dynamic row sum (WDRS) [Grie14], [BrCC15].

The template parameters `PPrecond` and `SPrecond` for the CPR WDRS preconditioner specify which preconditioner to use with the pressure subblock (the $C$ matrix in (2.1)), and with the complete system.

The system matrix should be ordered by grid nodes, so that the pressure and saturation/concentration unknowns belonging to the same grid node are compactly located together in the vector of unknowns. The pressure should be the first unknown in the block of unknowns associated with a grid node.

### class params

The CPR preconditioner parameters

```
typedef typename SPrecond::value_type value_type
```

The value type of the system matrix

```
typedef typename PPrecond::params pprecond_params
```

The type of the pressure preconditioner parameters

```
typedef typename SPrecond::params sprecond_params
```

The type of the global preconditioner parameters

```
pprecond_params pprecond
```

The pressure preconditioner parameters

```
sprecond_params sprecond
```

The global preconditioner parameters

```
int block_size = 2
```

The number of unknowns associated with each grid node. The default value is 2 when the system matrix has scalar value type. Otherwise, the block size of the system matrix value type is used.

```
size_t active_rows = 0
```

When non-zero, only unknowns below this number are considered to be pressure. May be used when a system matrix contains unstructured tail block (for example, the unknowns associated with wells).

```
double eps_dd = 0.2
```

Controls the severity of the violation of diagonal dominance. See [Grie14] for more details.
double \texttt{eps\_ps} = 0.02
Controls the pressure/saturation coupling. See [Grie14] for more details.

\texttt{std::vector\<\texttt{double}\> weights}
The weights for the weighted DRS method. See [BrCC15] for more details.

\textbf{Schur Pressure Correction}

\texttt{template<class USolver, class PSolver> class amgcl::preconditioner::schur_pressure_correction}

\texttt{Include <amgcl/preconditioner/schur_pressure_correction.hpp>}

The system (2.1) may be rewritten as
\[
\begin{pmatrix}
A & B^T_1 \\
0 & S
\end{pmatrix}
\begin{pmatrix}
u \\
p
\end{pmatrix} =
\begin{pmatrix}
b_u \\
(b_p - B_2 A^{-1} b_u)
\end{pmatrix}
\]

where \( S = C - B_2 A^{-1} B^T_1 \) is the Schur complement. The Schur complement pressure correction preconditioner uses this representation and an approximation to the Schur complement matrix in order to decouple the pressure and the velocity parts of the system [ElHS08].

The two template parameters for the method, \texttt{USolver} and \texttt{PSolver}, specify the \texttt{preconditioned solvers} for the \( A \) and \( S \) blocks.

\texttt{class params}
The parameters for the Schur pressure correction preconditioner

\texttt{typedef typename USolver::params usolver_params}
The type of the USolver parameters

\texttt{typedef typename PSolver::params psolver_params}
The type of the PSolver parameters

\texttt{usolver_params usolver}
The USolver parameters

\texttt{psolver_params psolver}
The PSolver parameters

\texttt{std::vector\<\texttt{char}\> pmask}
The indicator vector, containing 1 for pressure unknowns, and 0 otherwise.

\texttt{int type = 1}
The variant of the block preconditioner to use.

- When \texttt{type} = 1:

  \[
  Sp = b_p - B_2 A^{-1} b_u \\
  Au = b_u - B^T_1 p
  \]

- When \texttt{type} = 2:

  \[
  Sp = b_p \\
  Au = b_u - B^T_1 p
  \]

\texttt{bool approx\_schur = false}
When set, approximate \( A^{-1} \) as \( \text{diag}(A)^{-1} \) during computation of the matrix-less Schur complement when solving the \( Sp = b_p \) system. Otherwise, the full solve using \texttt{USolver} is used.

\textbf{2.3. Components}
int adjust_p = 1
Adjust the matrix used to construct the preconditioner for the Schur complement system.
• When adjust_p = 0, use the unmodified $C$ matrix;
• When adjust_p = 1, use $C - \text{diag}(B_2\text{diag}(A)^{-1}B_1^T)$;
• When adjust_p = 2, use $C - B_2\text{diag}(A)^{-1}B_1^T$.

bool simplec_dia = true
When set, use $\frac{1}{\sum_j \|A_{i,j}\|}$ instead of $\text{diag}(A)^{-1}$ as the approximation for $A^{-1}$ (similar to the SIMPLEC algorithm).

int verbose = 0
• When verbose >= 1, show the number of iterations and the relative residual achieved after each nested solve.
• When verbose >= 2, save the $A$ and $C$ submatrices as Kuu.mtx and Kpp.mtx.

### 2.3.6 Relaxation
A relaxation method or a smoother is used on each level of the AMG hierarchy during solution phase.

**Damped Jacobi**

template<class Backend>
class amgcl::relaxation::damped_jacobi

Include `<amgcl/relaxation/damped_jacobi.hpp>`

The damped Jacobi relaxation

class params
Damped Jacobi relaxation parameters
typedef typename Backend::value_type value_type
The value type of the system matrix
typedef typename amgcl::math::scalar_of<value_type>::type scalar_type
The scalar type corresponding to the value type. For example, when the value type is std::complex<double>, then the scalar type is double.

scalar_type damping = 0.72
The damping factor

**Gauss-Seidel**

template<class Backend>
class amgcl::relaxation::gauss_seidel

Include `<amgcl/relaxation/gauss_seidel.hpp>`

The Gauss-Seidel relaxation. The relaxation is only available for the backends where the matrix supports row-wise iteration over its non-zero values.

class params
Gauss-Seidel relaxation parameters
bool serial = false
Use the serial version of the algorithm

Chebyshev

template<class Backend>
class amgcl::relaxation::chebyshev

Include <amgcl/relaxation/chebyshev.hpp>

Chebyshev iteration is an iterative method for the solution of a system of linear equations, and unlike Jacobi, it is not a stationary method. However, it does not require inner products like many other nonstationary methods (most Krylov methods). These inner products can be a performance bottleneck on certain distributed memory architectures. Furthermore, Chebyshev iteration is, like Jacobi, easier to parallelize than for instance Gauss–Seidel smoothers. The Chebyshev iteration requires some information about the spectrum of the matrix. For symmetric matrices, it needs an upper bound for the largest eigenvalue and a lower bound for the smallest eigenvalue [GhKK12].

class params
Chebyshev relaxation parameters

unsigned degree = 5
The degree of Chebyshev polynomial

float higher = 1.0
The highest eigen value safety upscaling. use boosting factor for a more conservative upper bound estimate [ABHT03].

float lower = 1.0 / 30
Lowest-to-highest eigen value ratio.

int power_iters = 0
The number of power iterations to apply for the spectral radius estimation. When 0, use Gershgorin disk theorem to estimate the spectral radius.

bool scale = false
Scale the system matrix

Incomplete LU relaxation

The incomplete LU factorization process computes a sparse lower triangular matrix \( L \) and a sparse upper triangular matrix \( U \) so that the residual matrix \( R = LU - A \) satisfies certain constraints, such as having zero entries in some locations. The relaxations in this section use various approaches to computation of the triangular factors \( L \) and \( U \), but share the triangular system solution implementation required in order to apply the relaxation. The parameters for the triangular solution algorithm are defined as follows:

template<class Backend>
class amgcl::relaxation::detail::ilu_solve

For the builtin OpenMP backend the incomplete triangular factors are solved using the OpenMP-parallel level scheduling approach. For the GPGPU backends, the triangular factors are solved approximately, using multiple damped Jacobi iterations [ChPa15].

class params
bool serial = false
Use the serial version of the algorithm. This parameter is only used with the
amgcl::backend::builtin backend.

unsigned iters = 2
The number of Jacobi iterations to approximate the triangular system solution. This parameter is only
used with GPGPU backends.

scalar_type damping = 1.0
The damping factor for the triangular solve approximation. This parameter is only used with GPGPU
backends.

ILU0

template<class Backend>
class amgcl::relaxation::ilu0

Include <amgcl/relaxation/ilu0.hpp>

The incomplete LU factorization with zero fill-in [Saad03]. The zero pattern for the triangular factors \( L \) and \( U \) is taken to be exactly the zero pattern of the system matrix \( A \).

class params
   ILU0 relaxation parameters

   typedef typename Backend::value_type value_type
      The value type of the system matrix

   typedef typename amgcl::math::scalar_of<value_type>::type scalar_type
      The scalar type corresponding to the value type. For example, when the value type is
      std::complex<double>, then the scalar type is double.

   scalar_type damping = 1.0
      The damping factor

   typename amgcl::relaxation::detail::ilu_solve<Backend>::params solve
      The parameters for the triangular factor solver

ILUK

template<class Backend>
class amgcl::relaxation::iluk

Include <amgcl/relaxation/iluk.hpp>

The ILU(k) relaxation.

The incomplete LU factorization with the level of fill-in [Saad03]. The accuracy of the ILU0 incomplete factor-
ization may be insufficient to yield an adequate rate of convergence. More accurate incomplete LU factoriza-
tions are often more efficient as well as more reliable. These more accurate factorizations will differ from ILU(0) by
allowing some fill-in. Thus, ILUK(k) keeps the ‘k-th order fill-ins’ [Saad03].

The ILU(1) factorization results from taking the zero pattern for triangular factors to be the zero pattern of the
product \( L_0U_0 \) of the factors \( L_0, U_0 \) obtained from ILU(0). This process is repeated to obtain the higher level of
fill-in factorizations.

2.3. Components
class params
ILUK relaxation parameters

typedef typename Backend::value_type value_type
The value type of the system matrix

typedef typename amgcl::math::scalar_of/value_type::type scalar_type
The scalar type corresponding to the value type. For example, when the value type is
std::complex<double>, then the scalar type is double.

int k = 1
The level of fill-in

scalar_type damping = 1.0
The damping factor

typename amgcl::relaxation::detail::ilu_solve<Backend>::params solve
The parameters for the triangular factor solver

ILUP

template<class Backend>
class amgcl::relaxation::ilup

Include <amgcl/relaxation/ilup.hpp>

The ILUP(k) relaxation.

This variant of the ILU relaxation is similar to ILUK, but differs in the way the zero pattern for the triangular factors is determined. Instead of the recursive definition using the product $LU$ of the factors from the previous level of fill-in, ILUP uses the powers of the boolean matrix $S$ sharing the zero pattern with the system matrix $A$ [MiKu03]. ILUP(0) coincides with ILU0, ILUP(1) has the same zero pattern as $S^2$, etc.

class params
ILUP relaxation parameters

typedef typename Backend::value_type value_type
The value type of the system matrix

typedef typename amgcl::math::scalar_of/value_type::type scalar_type
The scalar type corresponding to the value type. For example, when the value type is
std::complex<double>, then the scalar type is double.

int k = 1
The level of fill-in

scalar_type damping = 1.0
The damping factor

typename amgcl::relaxation::detail::ilu_solve<Backend>::params solve
The parameters for the triangular factor solver

ILUT

template<class Backend>
class amgcl::relaxation::ilut
Include `<amgcl/relaxation/ilut.hpp>`

The ILUT\((p, \tau)\) relaxation.

Incomplete factorizations which rely on the levels of fill are blind to numerical values because elements that are dropped depend only on the structure of \(A\). This can cause some difficulties for realistic problems that arise in many applications. A few alternative methods are available which are based on dropping elements in the Gaussian elimination process according to their magnitude rather than their locations. With these techniques, the zero pattern \(P\) is determined dynamically.

A generic ILU algorithm with threshold can be derived from the IKJ version of Gaussian elimination by including a set of rules for dropping small elements. In the factorization ILUT\((p, \tau)\), the following rule is used:

1. an element is dropped (i.e., replaced by zero) if it is less than the relative tolerance \(\tau_i\) obtained by multiplying \(\tau\) by the original 2-norm of the \(i\)-th row.
2. Only the \(pl_i\) largest elements are kept in the \(L\) part of the row and the \(pu_i\) largest elements in the \(U\) part of the row in addition to the diagonal element, which is always kept. \(l_i\) and \(u_i\) are the number of nonzero elements in the \(i\)-th row of the system matrix \(A\) below and above the diagonal.

```cpp
class params
    ILUT relaxation parameters

    typedef typename Backend::value_type value_type
        The value type of the system matrix

    typedef typename amgcl::math::scalar_of<value_type>::type scalar_type
        The scalar type corresponding to the value type. For example, when the value type is std::complex<double>, then the scalar type is double.

    scalar_type p = 2
        The fill factor

    scalar_type tau = 1e-2
        The minimum magnitude of non-zero elements relative to the current row norm.

    scalar_type damping = 1.0
        The damping factor

    typename amgcl::relaxation::detail::ilu_solve<Backend>::params solve
        The parameters for the triangular factor solver
```

Sparse Approximate Inverse relaxation

Sparse approximate inverse (SPAI) smoothers based on the SPAI algorithm by Grote and Huckle [GrHu97]. The SPAI algorithm computes an approximate inverse \(M\) explicitly by minimizing \(I - MA\) in the Frobenius norm. Both the computation of \(M\) and its application as a smoother are inherently parallel. Since an effective sparsity pattern of \(M\) is in general unknown a priori, the computation cost can be greatly reduced by choosing an a priori sparsity pattern for \(M\). For SPAI-0 and SPAI-1 the sparsity pattern of \(M\) is fixed: \(M\) is diagonal for SPAI-0, whereas for SPAI-1 the sparsity pattern of \(M\) is that of \(A\) [BrGr02].

```cpp
template<class Backend>
class amgcl::relaxation::spai0
```
**Include** `<amgcl/relaxation/spai0.hpp>`

The SPAI-0 variant of the sparse approximate inverse smother [BrGr02].

```cpp
class params
    The SPAI-0 has no parameters
```

**SPAI1**

```cpp
template<class Backend>
class amgcl::relaxation::spai1
```

**Include** `<amgcl/relaxation/spai1.hpp>`

The SPAI-1 variant of the sparse approximate inverse smother [BrGr02].

```cpp
class params
    The SPAI-1 has no parameters
```

### 2.3.7 Coarsening Strategies

A coarsening strategy defines various options for creating coarse systems in the AMG hierarchy. A coarsening strategy takes the system matrix $A$ at the current level, and returns prolongation operator $P$ and the corresponding restriction operator $R$.

**Ruge-Stuben**

```cpp
template<class Backend>
class amgcl::coarsening::ruge_stuben
```

**Include** `<amgcl/coarsening/ruge_stuben>`

The classic Ruge-Stuben coarsening with direct interpolation [Stue99].

```cpp
class params
```

```cpp
float eps_strong = 0.25
    Parameter $\varepsilon_{str}$ defining strong couplings.

    Variable $i$ is defined to be strongly negatively coupled to another variable, $j$, if

    $$-a_{ij} \geq \varepsilon_{str} \max_{a_{ik} < 0} |a_{ik}|$$

    with fixed $0 < \varepsilon_{str} < 1$.

    In practice, a value of $\varepsilon_{str} = 0.25$ is usually taken.

bool do_trunc = true
    Prolongation operator truncation. Interpolation operators, and, hence coarse operators may increase substantially towards coarser levels. Without truncation, this may become too costly. Truncation ignores all interpolatory connections which are smaller (in absolute value) than the largest one by a factor of $\varepsilon_{tr}$. The remaining weights are rescaled so that the total sum remains unchanged. In practice, a value of $\varepsilon_{tr} = 0.2$ is usually taken.
```
Aggregation-based coarsening

The aggregation-base class of coarsening methods split the nodes at the fine grid into disjoint sets of nodes, the so-called aggregates that act as nodes on the coarse grid. The prolongation operators are then built by first constructing a tentative prolongator using the knowledge of zero energy modes of the principal part of the differential operator with natural boundary conditions (e.g., near null-space vectors, or rigid body modes for elasticity). In case of smoothed aggregation the prolongation operator is then smoothed by a carefully selected iteration.

All of the aggregation based methods take the aggregation and the nullspace parameters:

```cpp
class amgcl::coarsening::pointwise_aggregates
Pointwise aggregation. When the system matrix has a block structure, it is converted to a poinwise matrix (single value per block), and the aggregates are created for this reduced matrix instead.

class params
The aggregation parameters.

float eps_strong = 0.08
Parameter $\varepsilon_{\text{strong}}$ defining strong couplings. Connectivity is defined in a symmetric way, that is, two variables $i$ and $j$ are considered to be connected to each other if $\frac{a_{ij}}{a_{ii} a_{jj}} > \varepsilon_{\text{strong}}$ with fixed $0 < \varepsilon_{\text{strong}} < 1$.

int block_size = 1
The block size in case the system matrix has a block structure.
```

```cpp
class amgcl::coarsening::nullspace_params
The nullspace parameters.

int cols = 0
The number of near nullspace vectors.

std::vector<double> B
The near nullspace vectors. The vectors are represented as columns of a 2D matrix stored in row-major order.
```

Aggregation

```cpp
template<class Backend>
class amgcl::coarsening::aggregation

Include <amgcl/coarsening/aggregation.hpp>

The non-smoothed aggregation coarsening [Stue99].

```
float \texttt{over\_interp} = 1.5

Over-interpolation factor $\alpha$ [Stue99]. In case of aggregation coarsening, coarse-grid correction of smooth error, and by this the overall convergence, can often be substantially improved by using “over-interpolation”, that is, by multiplying the actual correction (corresponding to piecewise constant interpolation) by some factor $\alpha > 1$. Equivalently, this means that the coarse-level Galerkin operator is re-scaled by $1/\alpha$:

$$I_H^H A_h I_H^h \rightarrow \frac{1}{\alpha} I_H^H A_h I_H^h.$$ 

\section*{Smoothed Aggregation}

template<class \texttt{Backend}>
\begin{verbatim}

class amgcl::coarsening::smoothed_aggregation

Include \texttt{<amgcl/coarsening/smoothed_aggregation.hpp>}

The smoothed aggregation coarsening [VaMB96].

\textbf{class params}

The smoothed aggregation coarsening parameters

\begin{verbatim}
amgcl::coarsening::pointwise_aggregates::params aggr;
\end{verbatim}

The aggregation parameters

\begin{verbatim}
amgcl::coarsening::nullspace_params nullspace
\end{verbatim}

The near nullspace parameters

float \texttt{relax} = 1.0

The relaxation factor $r$. Used as a scaling for the damping factor $\omega$. When \texttt{estimate\_spectral\_radius} is set, then

$$\omega = r \times (4/3)/\rho.$$ 

where $\rho$ is the spectral radius of the system matrix. Otherwise

$$\omega = r \times (2/3).$$ 

The tentative prolongation $\tilde{P}$ from the non-smoothed aggregation is improved by smoothing to get the final prolongation matrix $P$. Simple Jacobi smoother is used here, giving the prolongation matrix

$$P = (I - \omega D^{-1} A^F) \tilde{P}.$$ 

Here $A^F = (a^F_{ij})$ is the filtered matrix given by

$$a^F_{ij} = \begin{cases} a_{ij} & \text{if } j \in N_i, \\ 0 & \text{otherwise, } \text{if } i \neq j, \end{cases}$$ 

and

$$a^F_{ii} = a_{ii} - \sum_{j=1, j \neq i}^{n} (a_{ij} - a^F_{ij}).$$ 

where $N_i$ is the set of variables strongly coupled to variable $i$, and $D$ denotes the diagonal of $A^F$.

bool \texttt{estimate\_spectral\_radius} = false

Estimate the matrix spectral radius. This usually improves the convergence rate and results in faster solves, but costs some time during setup.
int power_iters = 0
The number of power iterations to apply for the spectral radius estimation. Use Gershgorin disk
theorem when power_iters = 0.

Smoothed Aggregation with Energy Minimization

template<class Backend>
class amgcl::coarsening::smoothed_aggr_emin

Include <amgcl/coarsening/smoothed_aggr_emin.hpp>

The smoothed aggregation with energy minimization coarsening [SaTu08].

class params
The smoothed aggregation with energy minimization coarsening parameters

amgcl::coarsening::pointwise_aggregates::params aggr;
The aggregation parameters

amgcl::coarsening::nullspace_params nullspace
The near nullspace parameters

2.3.8 Coupling Solvers with Preconditioners

These classes provide a convenient way to couple an iterative solver and a preconditioner. This may be used both for
convenience and as a building block for a composite preconditioner.

make_solver

template<class Precond, class IterSolver>
class amgcl::make_solver

Include <amgcl/make_solver.hpp>

The class has two template parameters: Precond and IterSolver, which specify the preconditioner and
the iterative solver to use. During construction of the class, instances of both components are constructed and
are ready to use as a whole.

typedef typename Backend::params backend_params
The backend parameters

typedef typename Backend::value_type value_type
The value type of the system matrix

typedef typename amgcl::math::scalar_of<value_type>::type scalar_type
The scalar type corresponding to the value type. For example, when the value type is
std::complex<double>, then the scalar type is double.

class params
The coupled solver parameters

typename Precond::params precond
The preconditioner parameters

2.3. Components
### IterSolver::params solver

The iterative solver parameters

```cpp
template<class Matrix>
make_solver (const Matrix &A, const params &prm = params(), const backend_params &bprm = backend_params())
    The constructor
```

```cpp
template<class Matrix, class VectorRHS, class VectorX>
std::tuple<size_t, scalar_type> operator() (const Matrix &A, const VectorRHS &rhs, VectorX &x) const
    Computes the solution for the given system matrix A and the right-hand side rhs. Returns the number of iterations made and the achieved residual as a std::tuple. The solution vector x provides initial approximation on input and holds the computed solution on output.

    The system matrix may differ from the matrix used during initialization. This may be used for the solution of non-stationary problems with slowly changing coefficients. There is a strong chance that a preconditioner built for a time step will act as a reasonably good preconditioner for several subsequent time steps [DeSh12].
```

```cpp
template<class VectorRHS, class VectorX>
std::tuple<size_t, scalar_type> operator() (const VectorRHS &rhs, VectorX &x) const
    Computes the solution for the given right-hand side rhs. Returns the number of iterations made and the achieved residual as a std::tuple. The solution vector x provides initial approximation on input and holds the computed solution on output.
```

```cpp
const Precond &precond() const
    Returns reference to the constructed preconditioner
```

```cpp
const IterSolver &solver() const
    Returns reference to the constructed iterative solver
```

### make_block_solver

```cpp
template<class Precond, class IterSolver>
class amgcl::make_block_solver
```

**Include** `<amgcl/make_block_solver.hpp>`

Creates coupled solver which targets a block valued backend, but may be initialized with a scalar system matrix, and used with scalar vectors.

The scalar system matrix is transparently converted to the block-valued on using the amgcl::adapter::block_matrix() adapter in the class constructor, and the scalar vectors are reinterpreted to the block-valued ones upon application.

This class may be used as a building block in a composite preconditioner, when one (or more) of the subsystems has block values, but has to be computed as a scalar matrix.

The interface is the same as that of amgcl::make_solver.

### deflated_solver

```cpp
template<class Precond, class IterSolver>
class amgcl::deflated_solver
```

2.3. Components 36
Include `<amgcl/deflated_solver.hpp>`

This function creates a preconditioned deflated solver. Deflated Krylov subspace methods are supposed to solve problems with large jumps in the coefficients on layered domains. It appears that the convergence of a deflated solver is independent of the size of the jump in the coefficients. The specific variant of the deflation method used here is A-DEF2 from [TNVE09].

```cpp
typedef typename Backend::params backend_params
  The backend parameters

typedef typename Backend::value_type value_type
  The value type of the system matrix

typedef typename amgcl::math::scalar_of<value_type>::type scalar_type
  The scalar type corresponding to the value type. For example, when the value type is std::complex<double>, then the scalar type is double.

class params
  The deflated solver parameters
  int nvec = 0
    The number of deflation vectors
  scalar_type *vec = nullptr
    The deflation vectors stored as a [nvec x n] matrix in row-major order

typename Precond::params precond
  The preconditioner parameters

IterSolver::params solver
  The iterative solver parameters
```

2.4 Tutorial

In this section we demonstrate the solution of some common types of problems. The first three problems are matrices from the SuiteSparse Matrix Collection, which is a widely used set of sparse matrix benchmarks. The Stokes problem may be downloaded from the dataset accompanying the [DeMW20] paper. The solution timings used in the sections below were obtained on an Intel Core i5-3570K CPU. The timings for the GPU backends were obtained with the NVIDIA GeForce GTX 1050 Ti GPU.

2.4.1 Poisson problem

This system may be downloaded from the poisson3Db page (use the Matrix Market download option). The system matrix has 85,623 rows and 2,374,949 nonzeros (which is on average is about 27 non-zero elements per row). The matrix has an interesting structure, presented on the figure below:

A Poisson problem should be an ideal candidate for a solution with an AMG preconditioner, but before we start writing any code, let us check this with the examples/solver utility provided by AMGCL. It can take the input matrix and the RHS in the Matrix Market format, and allows to play with various solver and preconditioner options.

Note: The examples/solver is convenient not only for testing the systems obtained elsewhere. You can also save your own matrix and the RHS vector into the Matrix Market format with `amgcl::io::mm_write()` function. This way you can find the AMGCL options that work for your problem without the need to rewrite the code and recompile the program.
Fig. 2.1: Poisson3Db matrix portrait
The default options of BiCGStab iterative solver preconditioned with a smoothed aggregation AMG (a simple diagonal SPAI(0) relaxation is used on each level of the AMG hierarchy) should work very well for a Poisson problem:

```bash
$ solver -A poisson3Db.mtx -f poisson3Db_b.mtx
Solver
======
Type: BiCGStab
Unknowns: 85623
Memory footprint: 4.57 M

Preconditioner
==============
Number of levels: 3
Operator complexity: 1.20
Grid complexity: 1.08
Memory footprint: 58.93 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>85623</td>
<td>2374949</td>
<td>50.07 M (83.20%)</td>
</tr>
<tr>
<td>1</td>
<td>6361</td>
<td>446833</td>
<td>7.78 M (15.65%)</td>
</tr>
<tr>
<td>2</td>
<td>384</td>
<td>32566</td>
<td>1.08 M (1.14%)</td>
</tr>
</tbody>
</table>

Iterations: 24
Error: 8.33789e-09

[Profile: 2.351 s] (100.00%)
[ reading: 1.623 s] (69.01%)
[ setup: 0.136 s] (5.78%)
[ solve: 0.592 s] (25.17%)
```

As we can see from the output, the solution converged in 24 iterations to the default relative error of 1e-8. The solver setup took 0.136 seconds and the solution time is 0.592 seconds. The iterative solver used 4.57M of memory, and the preconditioner required 58.93M. This looks like a well-performing solver already, but we can try a couple of things just in case. We cannot use the simpler CG solver, because the matrix is reported as a non-symmetric on the poisson3Db page. Using the GMRES solver seems to work equally well (the solution time is just slightly lower, but the solver requires more memory to store the orthogonal vectors). The number of iterations seems to have grown, but keep in mind that each iteration of BiCGStab requires two matrix-vector products and two preconditioner applications, while GMRES only makes one of each:

```bash
$ solver -A poisson3Db.mtx -f poisson3Db_b.mtx solver.type=gmres
Solver
======
Type: GMRES(30)
Unknowns: 85623
Memory footprint: 20.91 M

Preconditioner
==============
Number of levels: 3
Operator complexity: 1.20
Grid complexity: 1.08
Memory footprint: 58.93 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>85623</td>
<td>2374949</td>
<td>50.07 M (83.20%)</td>
</tr>
</tbody>
</table>
```

(continues on next page)
We can also try different relaxation options for the AMG preconditioner. But as we can see below, the simplest SPAI(0) works well enough for a Poisson problem. The incomplete LU decomposition with zero fill-in makes less iterations, but is more expensive to setup:

```
$ solver -A poisson3Db.mtx -f poisson3Db_b.mtx precond.relax.type=ilu0
Solver
======
Type: BiCGStab
Unknowns: 85623
Memory footprint: 4.57 M
Preconditioner
=============
Number of levels: 3
Operator complexity: 1.20
Grid complexity: 1.08
Memory footprint: 103.44 M

level | unknowns | nonzeros | memory
---------------------------------------------
0     | 85623    | 2374949  | 87.63 M (83.20%)  
1     | 6361     | 446833   | 14.73 M (15.65%) 
2     | 384      | 32566    | 1.08 M ( 1.14%)  

Iterations: 12
Error: 7.99207e-09

[Profile: 2.510 s] (100.00%)
[ reading: 1.614 s] ( 64.30%)
[ setup: 0.464 s] ( 18.51%)
[ solve: 0.427 s] ( 17.01%)
```

On the other hand, the Chebyshev relaxation has cheap setup but its application is expensive as it involves multiple matrix-vector products. So, even though it requires less iterations, the overall solution time does not improve that much:

```
$ solver -A poisson3Db.mtx -f poisson3Db_b.mtx precond.relax.type=chebyshev
Solver
======
Type: BiCGStab
Unknowns: 85623
Memory footprint: 4.57 M
Preconditioner

(continues on next page)
Number of levels: 3
Operator complexity: 1.20
Grid complexity: 1.08
Memory footprint: 59.63 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>85623</td>
<td>2374949</td>
<td>50.72 M (83.20%)</td>
</tr>
<tr>
<td>1</td>
<td>6361</td>
<td>446833</td>
<td>7.83 M (15.65%)</td>
</tr>
<tr>
<td>2</td>
<td>384</td>
<td>32566</td>
<td>1.08 M (1.14%)</td>
</tr>
</tbody>
</table>

Iterations: 8
Error: 5.21588e-09

[Profile: 2.316 s (100.00%)
 [ reading: 1.607 s (69.39%)
 [ setup: 0.134 s (5.78%)
 [ solve: 0.574 s (24.80%)

Now that we have the feel of the problem, we can actually write some code. The complete source may be found in tutorial/1.poisson3Db/poisson3Db.cpp and is presented below:

Listing 2.5: The source code for the solution of the poisson3Db problem.

```cpp
#include <vector>
#include <iostream>

#include <amgcl/backend/builtin.hpp>
#include <amgcl/adapter/crs_tuple.hpp>
#include <amgcl/make_solver.hpp>
#include <amgcl/amg.hpp>
#include <amgcl/coarsening/smoothed_aggregation.hpp>
#include <amgcl/relaxation/spai0.hpp>
#include <amgcl/solver/bicgstab.hpp>

#include <amgcl/io/mm.hpp>
#include <amgcl/profiler.hpp>

int main(int argc, char *argv[]) {
  // The matrix and the RHS file names should be in the command line options:
  if (argc < 3) {
    std::cerr << "Usage: " << argv[0] << " <matrix.mtx> <rhs.mtx>" << std::endl;
    return 1;
  }

  // The profiler:
  amgcl::profiler<> prof("poisson3Db");

  // Read the system matrix and the RHS:
  ptrdiff_t rows, cols;
  std::vector<ptrdiff_t> ptr, col;
  std::vector<double> val, rhs;

  prof.tic("read");
  std::tie(rows, cols) = amgcl::io::mm_reader(argv[1])(ptr, col, val);
```
In lines 4–10 we include the necessary AMGCL headers: the builtin backend uses the OpenMP threading model; the `crs_tuple` matrix adapter allows to use a `std::tuple` of CRS arrays as an input matrix; the `amgcl::make_solver` class binds together a preconditioner and an iterative solver; `amgcl::amg` class is
the AMG preconditioner; \texttt{amgcl::coarsening::smoothed_aggregation} defines the smoothed aggregation
coarsening strategy; \texttt{amgcl::relaxation::spai0} is the sparse approximate inverse relaxation used on each
level of the AMG hierarchy; and \texttt{amgcl::solver::bicgstab} is the BiCGStab iterative solver. In lines 12–13
we include the \texttt{Matrix Market} reader and the AMGCL profiler.

After checking the validity of the command line arguments (lines 16–20), and initializing the profiler (line 23), we
read the system matrix and the RHS vector from the \texttt{Matrix Market} files specified on the command line (lines 30–36).

Now we are ready to actually solve the system. First, we define the backends that we use with the iterative solver
and the preconditioner (lines 44–51). The backend have to belong to the same class (in this case, \texttt{amgcl::backend::builtin}),
but may have different value type precision. Here we use a double precision backend for the iterative solver, but choose
either a double or a single precision for the preconditioner backend, depending on whether the preprocessor macro \texttt{MIXED_PRECISION}
was defined during compilation. Using a single precision preconditioner may be both more memory efficient and faster, since the iterative solvers performance is
usually memory-bound.

The defined backends are used in the solver definition (lines 53–60). Here we are using the \texttt{amgcl::make_solver}
class to couple the AMG preconditioner with the BiCGStab iterative solver. We instantiate the solver in line 64.

In line 76 we solve the system for the given RHS vector, starting with a zero initial approximation (the $x$ vector acts
as an initial approximation on input, and contains the solution on output).

Below is the output of the program when compiled with a double precision preconditioner. The results are close to
what we have seen with the \texttt{examples/solver} utility above, which is a good sign:

```bash
$ ./poisson3Db poisson3Db.mtx poisson3Db_b.mtx
Matrix poisson3Db.mtx: 85623x85623
RHS poisson3Db_b.mtx: 85623x1
Solver
======
Type: BiCGStab
Unknowns: 85623
Memory footprint: 4.57 M
Preconditioner
==============
Number of levels: 3
Operator complexity: 1.20
Grid complexity: 1.08
Memory footprint: 58.93 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>85623</td>
<td>2374949</td>
<td>50.07 M (83.20%)</td>
</tr>
<tr>
<td>1</td>
<td>6361</td>
<td>446833</td>
<td>7.78 M (15.65%)</td>
</tr>
<tr>
<td>2</td>
<td>384</td>
<td>32566</td>
<td>1.08 M (1.14%)</td>
</tr>
</tbody>
</table>

Iters: 24
Error: 8.33789e-09

[poisson3Db:  2.412 s] (100.00%)
[   read:  1.618 s] ( 67.08%)
[   setup:  0.143 s] (  5.94%)
[   solve:  0.651 s] ( 26.98%)
```

Looking at the output of the mixed precision version, it is apparent that it uses less memory for the preconditioner
(43.59M as opposed to 58.93M in the double-precision case), and is slightly faster during both the setup and the
solution phases:
$ ./poisson3Db_mixed poisson3Db.mtx poisson3Db_b.mtx
Matrix poisson3Db.mtx: 85623x85623
RHS poisson3Db_b.mtx: 85623x1
Solver
======
Type: BiCGStab
Unknowns: 85623
Memory footprint: 4.57 M

Preconditioner
==============
Number of levels: 3
Operator complexity: 1.20
Grid complexity: 1.08
Memory footprint: 43.59 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>85623</td>
<td>2374949</td>
<td>37.23 M</td>
</tr>
<tr>
<td>1</td>
<td>6361</td>
<td>446833</td>
<td>5.81 M</td>
</tr>
<tr>
<td>2</td>
<td>384</td>
<td>32566</td>
<td>554.90 K</td>
</tr>
</tbody>
</table>

Iters: 24
Error: 7.33493e-09

[poisson3Db: 2.234 s] (100.00%)
[ read: 1.559 s] ( 69.78%)
[ setup: 0.125 s] ( 5.59%)
[ solve: 0.550 s] (24.62%)

We may also try to switch to the CUDA backend in order to accelerate the solution using an NVIDIA GPU. We only need to use the `amgcl::backend::cuda` instead of the `builtin` backend, and we also need to initialize the CUSPARSE library and pass the handle to AMGCL as the backend parameters. Unfortunately, we cannot use the mixed precision approach, as CUSPARSE does not support that (we could use the VexCL backend though, see Poisson problem (MPI version)). The source code is very close to what we have seen above and is available at tutorial/1.poisson3Db/poisson3Db_cuda.cu. The listing below has the differences highlighted:

Listing 2.6: The source code for the solution of the poisson3Db problem using the CUDA backend.

```cpp
#include <vector>
#include <iostream>

#include <amgcl/backend/cuda.hpp>
#include <amgcl/adapter/crs_tuple.hpp>
#include <amgcl/make_solver.hpp>
#include <amgcl/amg.hpp>
#include <amgcl/coarsening/smoothed_aggregation.hpp>
#include <amgcl/relaxation/spai0.hpp>
#include <amgcl/solver/bicgstab.hpp>
#include <amgcl/io/mm.hpp>
#include <amgcl/profiler.hpp>

int main(int argc, char *argv[]) {
    // The matrix and the RHS file names should be in the command line options:
    if (argc < 3) {
        (continues on next page)
std::cerr << "Usage: " << argv[0] << " <matrix.mtx> <rhs.mtx>" << std::endl;
    return 1;
}

// Show the name of the GPU we are using:
int device;
cudaDeviceProp prop;
cudaGetDevice(&device);
cudaGetDeviceProperties(&prop, device);
std::cout << prop.name << std::endl;

// The profiler:
amgcl::profiler<> prof("poisson3Db");

// Read the system matrix and the RHS:
ptrdiff_t rows, cols;
std::vector<ptrdiff_t> ptr, col;
std::vector<double> val, rhs;

prof.tic("read");
std::tie(rows, cols) = amgcl::io::mm_reader(argv[1])(ptr, col, val);
std::cout << "Matrix " << argv[1] << ": " << rows << "x" << cols << std::endl;

std::tie(rows, cols) = amgcl::io::mm_reader(argv[2])(rhs);
std::cout << "RHS " << argv[2] << ": " << rows << "x" << cols << std::endl;
prof.toc("read");

// We use the tuple of CRS arrays to represent the system matrix.
// Note that std::tie creates a tuple of references, so no data is actually
// copied here:
auto A = std::tie(rows, ptr, col, val);

// Compose the solver type
typedef amgcl::backend::cuda<double> Backend;
typedef amgcl::make_solver<
    amgcl::amg<
    Backend,
    amgcl::coarsening::smoothed_aggregation,
    amgcl::relaxation::spai0
    >,
    amgcl::solver::bicgstab<Backend>
> Solver;

// We need to initialize the CUSPARSE library and pass the handle to AMGCL
// in backend parameters:
Backend::params bprm;
cusparseCreate(&bprm.cusparse_handle);

// There is no way to pass the backend parameters without passing the
// solver parameters, so we also need to create those. But we can leave
// them with the default values:
Solver::params prm;

// Initialize the solver with the system matrix:
prof.tic("setup");
Solver solve(A, prm, bprm);
prof.toc("setup");

(continues on next page)
// Show the mini-report on the constructed solver:
std::cout << solve << std::endl;

// Solve the system with the zero initial approximation.
// The RHS and the solution vectors should reside in the GPU memory:
int iters;
double error;
thrust::device_vector<double> f(rhs);
thrust::device_vector<double> x(rows, 0.0);

prof.tic("solve");
std::tie(iters, error) = solve(f, x);    
prof.toc("solve");

// Output the number of iterations, the relative error,
// and the profiling data:
// (continued from previous page)
std::cout << "Iters: " << iters << std::endl
<< "Error: " << error << std::endl
<< prof << std::endl;
}

Using the consumer level GeForce GTX 1050 Ti GPU, the solution phase is almost 4 times faster than with the OpenMP backend. On the contrary, the setup is slower, because we now need to additionally initialize the GPU-side structures. Overall, the complete solution is about twice faster (comparing with the double precision OpenMP version):

```
$ ./poisson3Db_cuda poisson3Db.mtx poisson3Db_b.mtx
GeForce GTX 1050 Ti
Matrix poisson3Db.mtx: 85623x85623
RHS poisson3Db_b.mtx: 85623x1
Solver
======
Type: BiCGStab
Unknowns: 85623
Memory footprint: 4.57 M

Preconditioner
=============
Number of levels: 3
Operator complexity: 1.20
Grid complexity: 1.08
Memory footprint: 44.81 M

level unknowns nonzeros memory
---------------------------------------------
0  85623  2374949  37.86 M (83.20%)
1   6361  446833   5.86 M (15.65%)
2    384  32566   1.09 M ( 1.14%)

Iters: 24
Error: 8.33789e-09

[poisson3Db: 2.253 s] (100.00%)
[ self: 0.223 s] ( 9.90%)
[ read: 1.676 s] (74.39%)
```
2.4.2 Poisson problem (MPI version)

In section Poisson problem we looked at the solution of the 3D Poisson problem (available for download at poisson3Db page) using the shared memory approach. Let's solve the same problem using the Message Passing Interface (MPI), or the distributed memory approach. We already know that using the smoothed aggregation AMG with the simple SPAI(0) smoother is working well, so we may start writing the code immediately. The following is the complete MPI-based implementation of the solver (tutorial/1.poisson3Db/poisson3Db_mpi.cpp). We discuss it in more details below.

Listing 2.7: The MPI solution of the poisson3Db problem

```cpp
#include <vector>
#include <iostream>
#include <amgcl/backend/builtin.hpp>
#include <amgcl/adapter/crs_tuple.hpp>
#include <amgcl/mpi/distributed_matrix.hpp>
#include <amgcl/mpi/make_solver.hpp>
#include <amgcl/mpi/amg.hpp>
#include <amgcl/mpi/coarsening/smoothed_aggregation.hpp>
#include <amgcl/mpi/relaxation/spai0.hpp>
#include <amgcl/mpi/solver/bicgstab.hpp>
#include <amgcl/mpi/io/binary.hpp>
#include <amgcl/mpi/profiler.hpp>

#include <amgcl/io/binary.hpp>

#include <amgcl/profiler.hpp>

#if defined(AMGCL_HAVE_PARMETIS)
#include <amgcl/mpi/partition/parmetis.hpp>
#elif defined(AMGCL_HAVE_SCOTCH)
#include <amgcl/mpi/partition/ptscotch.hpp>
#endif

int main(int argc, char *argv[]) {
    // The matrix and the RHS file names should be in the command line options:
    if (argc < 3) {
        std::cerr << "Usage: " << argv[0] << " matrix.bin <rhs.bin>" << std::endl;
        return 1;
    }
    amgcl::mpi::init mpi(argc, argv);
    amgcl::mpi::communicator world(MPI_COMM_WORLD);

    // The profiler:
    amgcl:: profiler prof("poisson3Db MPI");

    // Read the system matrix and the RHS:
    prof.tic("read");
    // Get the global size of the matrix:
    ptrdiff_t rows = amgcl::io::crs_size<ptrdiff_t>(argv[1]);
```
ptrdiff_t cols;

// Split the matrix into approximately equal chunks of rows
ptrdiff_t chunk = (rows + world.size - 1) / world.size;
ptrdiff_t row_beg = std::min(rows, chunk * world.rank);
ptrdiff_t row_end = std::min(rows, row_beg + chunk);
chunk = row_end - row_beg;

// Read our part of the system matrix and the RHS.
std::vector<ptrdiff_t> ptr, col;
std::vector<double> val, rhs;
amgcl::io::read_crs(argv[1], rows, ptr, col, val, row_beg, row_end);
amgcl::io::read_dense(argv[2], rows, cols, rhs, row_beg, row_end);
prof.toc("read");

if (world.rank == 0)
  std::cout
    << "World size: " << world.size << std::endl
    << "Matrix " << argv[1] << ": " << rows << "x" << rows << std::endl
    << "RHS " << argv[2] << ": " << rows << "x" << cols << std::endl;

// Compose the solver type
typedef amgcl::backend::builtin<double> DBackend;
typedef amgcl::backend::builtin<float> FBackend;
typedef amgcl::mpi::make_solver<
  amgcl::mpi::amg<
    amgcl::mpi::coarsening::smoothed_aggregation<FBackend>,
    amgcl::mpi::relaxation::spai0<FBackend>
  >,
  amgcl::mpi::solver::bicgstab<DBackend>
> Solver;

// Create the distributed matrix from the local parts.
auto A = std::make_shared<amgcl::mpi::distributed_matrix<DBackend>>(
  world, std::tie(chunk, ptr, col, val));

// Partition the matrix and the RHS vector.
// If neither ParMETIS nor PT-SCOTCH are not available,
// just keep the current naive partitioning.
#elif defined(AMGCL_HAVE_PARMETIS) || defined(AMGCL_HAVE_SCOTCH)
  #if defined(AMGCL_HAVE_PARMETIS)
    typedef amgcl::mpi::partition::parmetis<DBackend> Partition;
  # elif defined(AMGCL_HAVE_SCOTCH)
    typedef amgcl::mpi::partition::ptscotch<DBackend> Partition;
  # endif
  #define AMGCL_HAVE_PARMETIS
  #define AMGCL_HAVE_SCOTCH
  #endif

if (world.size > 1) {
  prof.tic("partition");
  Partition part;

  // part(A) returns the distributed permutation matrix:
  auto P = part(*A);
  auto R = transpose(*P);

  // Reorder the matrix:
}
In lines 4–21 we include the required components. Here we are using the builtin (OpenMP-based) backend and the CRS tuple adapter. Next we include MPI-specific headers that provide the distributed-memory implementation of AMGCL algorithms. This time, we are reading the system matrix and the RHS vector in the binary format, and include `<amgcl/io/binary.hpp>` header instead of the usual `<amgcl/io/mm.hpp>`. The binary format is not only faster to read, but it also allows to read the matrix and the RHS vector in chunks, which is what we need for the distributed approach.

After checking the validity of the command line parameters, we initialize the MPI context and communicator in lines 31–32:

```
amgcl::mpi::init mpi($argc, $argv);
amgcl::mpi::communicator world(MPI_COMM_WORLD);
```

The `amgcl::mpi::init` is a convenience RAII wrapper for `MPI_Init()`. It will call `MPI_Finalize()` in the destructor when its instance (`mpi`) goes out of scope at the end of the program. We don’t have to use the wrapper, but it simply makes things easier. `amgcl::mpi::communicator` is an equally thin wrapper for `MPI_Comm`. 

### 2.4. Tutorial
amgcl::mpi::communicator and MPI_Comm may be used interchangeably both with the AMGCL MPI interface and the native MPI functions.

The system has to be divided (partitioned) between multiple MPI processes. The simplest way to do this is presented on the following figure:

![Matrix Partitioning Diagram](image)

Fig. 2.2: Poisson3Db matrix partitioned between the 4 MPI processes

Assuming we are using 4 MPI processes, the matrix is split into 4 continuous chunks of rows, so that each MPI process owns approximately 25% of the matrix. This works well enough for a small number of processes, but as the size of the compute cluster grows, the simple partitioning becomes less and less efficient. Creating efficient partitioning is outside of AMGCL scope, but AMGCL does provide wrappers for the ParMETIS and PT-SCOTCH libraries specializing in this. The difference between the naive and the optimal partitioning is demonstrated on the next figure:

The figure shows the finite-difference discretization of a 2D Poisson problem on a $4 \times 4$ grid in a unit square. The nonzero pattern of the system matrix is presented on the lower left plot. If the grid nodes are numbered row-wise, then the naive partitioning of the system matrix for the 4 MPI processes is shown on the upper left plot. The subdomains belonging to each of the MPI processes correspond to the continuous ranges of grid node indices and are elongated along the X axis. This results in high MPI communication traffic, as the number of the interface nodes is high relative to...
Fig. 2.3: Naive vs optimal partitioning of a $4 \times 4$ grid between 4 MPI processes.
to the number of interior nodes. The upper right plot shows the optimal partitioning of the domain for the 4 MPI processes. In order to keep the rows owned by a single MPI process adjacent to each other (so that each MPI process owns a continuous range of rows, as required by AMGCL), the grid nodes have to be renumbered. The labels in the top left corner of each grid node show the original numbering, and the lower-right labels show the new numbering. The renumbering of the matrix may be expressed as the permutation matrix $P$, where $P_{ij} = 1$ if the $j$-th unknown in the original ordering is mapped to the $i$-th unknown in the new ordering. The reordered system may be written as

$$P^T A P y = P^T f$$

The reordered matrix $P^T A P$ and the corresponding partitioning are shown on the lower right plot. Note that off-diagonal blocks on each MPI process have as much as twice fewer non-zeros compared to the naive partitioning of the matrix. The solution $x$ in the original ordering may be obtained with $x = P y$.

In lines 37–54 we read the system matrix and the RHS vector using the naive ordering (a nicer ordering of the unknowns will be determined later):

```cpp
// Read the system matrix and the RHS:
prof.tic("read");
// Get the global size of the matrix:
ptrdiff_t rows = amgcl::io::crs_size<ptrdiff_t>(argv[1]);
ptrdiff_t cols;

// Split the matrix into approximately equal chunks of rows
ptrdiff_t chunk = (rows + world.size - 1) / world.size;
ptrdiff_t row_beg = std::min(rows, chunk * world.rank);
ptrdiff_t row_end = std::min(rows, row_beg + chunk);
chunk = row_end - row_beg;

// Read our part of the system matrix and the RHS.
std::vector<ptrdiff_t> ptr, col;
std::vector<double> val, rhs;
amgcl::io::read_crs(argv[1], rows, ptr, col, val, row_beg, row_end);
amgcl::io::read_dense(argv[2], rows, cols, rhs, row_beg, row_end);
prof.toc("read");
```

First, we read the total (global) number of rows in the matrix from the binary file using the `amgcl::io::crs_size()` function. Next, we divide the global rows between the MPI processes, and read our portions of the matrix and the RHS using `amgcl::io::read_crs()` and `amgcl::io::read_dense()` functions. The `row_beg` and `row_end` parameters to the functions specify the regions (in row numbers) to read. The column indices are kept in global numbering.

In lines 62–72 we define the backend and the solver types:

```cpp
// Compose the solver type
typedef amgcl::backend::builtin<double> DBackend;
typedef amgcl::backend::builtin<float> FBackend;
typedef amgcl::mpi::make_solver<
    amgcl::mpi::amg<
        FBackend,
        amgcl::mpi::coarsening::smoothed_aggregation<FBackend>,
        amgcl::mpi::relaxation::spai0<FBackend>
    >,
    amgcl::mpi::solver::bicgstab<DBackend>
> Solver;
```

The structure of the solver is the same as in the shared memory case in the *Poisson problem* tutorial, but we are using the components from the `amgcl::mpi` namespace. Again, we are using the mixed-precision approach and the preconditioner backend is defined with a single-precision value type.
In lines 74–76 we create the distributed matrix from the local strips read by each of the MPI processes:

```cpp
// Create the distributed matrix from the local parts.
auto A = std::make_shared<amgcl::mpi::distributed_matrix<DBackend>>(
    world, std::tie(chunk, ptr, col, val));
```

We could directly use the tuple of the CRS arrays `std::tie(chunk, ptr, col, val)` to construct the solver (the distributed matrix would be created behind the scenes for us), but here we need to explicitly create the matrix for a couple of reasons. First, since we are using the mixed-precision approach, we need the double-precision distributed matrix for the solution step. And second, the matrix will be used to repartition the system using either ParMETIS or PT-SCOTCH libraries in lines 78–110:

```cpp
// Partition the matrix and the RHS vector.
// If neither ParMETIS not PT-SCOTCH are not available,
// just keep the current naive partitioning.
#if defined(AMGCL_HAVE_PARMETIS) || defined(AMGCL_HAVE_SCOTCH)
    #if defined(AMGCL_HAVE_PARMETIS)
        typedef amgcl::mpi::partition::parmetis<DBackend> Partition;
    #elif defined(AMGCL_HAVE_SCOTCH)
        typedef amgcl::mpi::partition::ptscotch<DBackend> Partition;
    # endif
#endif

if (world.size > 1) {
    prof.tic("partition");
    Partition part;

    // part(A) returns the distributed permutation matrix:
    auto P = part(*A);
    auto R = transpose(*P);

    // Reorder the matrix:

    // and the RHS vector:
    std::vector<double> new_rhs(R->loc_rows());
    R->move_to_backend(typename DBackend::params());
    amgcl::backend::spmv(1, *R, rhs, 0, new_rhs);
    rhs.swap(new_rhs);

    // Update the number of the local rows
    // (it may have changed as a result of permutation):
    chunk = A->loc_rows();
    prof.toc("partition");
}
```

We determine if either ParMETIS or PT-SCOTCH is available in lines 81–86, and use the corresponding wrapper provided by the AMGCL. The wrapper computes the permutation matrix $P$, which is used to reorder both the system matrix and the RHS vector. Since the reordering may change the number of rows owned by each MPI process, we update the number of local rows stored in the `chunk` variable.

```cpp
// Initialize the solver:
prof.tic("setup");
Solver solve(world, A);
prof.toc("setup");
```

```
// Show the mini-report on the constructed solver:
```

(continues on next page)
if (world.rank == 0)
    std::cout << solve << std::endl;

// Solve the system with the zero initial approximation:
    int iters;
    double error;
    std::vector<double> x(chunk, 0.0);
    prof.tic("solve");
    std::tie(iters, error) = solve(*A, rhs, x);
    prof.toc("solve");

At this point we are ready to initialize the solver (line 115), and solve the system (line 128). Here is the output of the compiled program. Note that the environment variable OMP_NUM_THREADS is set to 1 in order to not oversubscribe the available CPU cores:

$ export OMP_NUM_THREADS=1
$ mpirun -np 4 ./poisson3Db_mpi poisson3Db.bin poisson3Db_b.bin
World size: 4
Matrix poisson3Db.bin: 85623x85623
RHS poisson3Db_b.bin: 85623x1
Partitioning[ParMETIS] 4 -> 4
Type: BiCGStab
Unknowns: 21671
Memory footprint: 1.16 M

Number of levels: 3
Operator complexity: 1.20
Grid complexity: 1.08

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>85623</td>
<td>2374949  (83.06%) [4]</td>
</tr>
<tr>
<td>1</td>
<td>6377</td>
<td>450473   (15.75%) [4]</td>
</tr>
<tr>
<td>2</td>
<td>401</td>
<td>34039    ( 1.19%) [4]</td>
</tr>
</tbody>
</table>

Iters: 24
Error: 6.09835e-09

[poisson3Db MPI: 1.273 s] (100.00%)
[ self: 0.044 s] ( 3.49%)
[ partition: 0.626 s] (49.14%)
[ read: 0.012 s] ( 0.93%)
[ setup: 0.152 s] (11.92%)
[ solve: 0.439 s] (34.52%)

Similarly to how it was done in the Poisson problem section, we can use the GPU backend in order to speed up the solution step. Since the CUDA backend does not support the mixed-precision approach, we will use the VexCL backend, which allows to employ CUDA, OpenCL, or OpenMP compute devices. The source code (tutorial/1.poisson3Db/poisson3Db_mpi_vexcl.cpp) is very similar to the version using the builtin backend and is shown below with the differences highlighted.

```cpp
#include <vector>
#include <iostream>
#include <amgcl/backend/vexcl.hpp>
```
```cpp
#include <amgcl/adapter/crs_tuple.hpp>
#include <amgcl/mpi/distributed_matrix.hpp>
#include <amgcl/mpi/make_solver.hpp>
#include <amgcl/mpi/amg.hpp>
#include <amgcl/mpi/coarsening/smoothed_aggregation.hpp>
#include <amgcl/mpi/relaxation/spai0.hpp>
#include <amgcl/mpi/solver/bicgstab.hpp>
#include <amgcl/io/binary.hpp>
#include <amgcl/profiler.hpp>

#include <amgcl/mpi/partition/parmetis.hpp>
#include <amgcl/mpi/partition/ptscotch.hpp>

int main(int argc, char *argv[]) {
  // The matrix and the RHS file names should be in the command line options:
  if (argc < 3) {
    std::cerr << "Usage: " << argv[0] << " <matrix.bin> <rhs.bin>" << std::endl;
    return 1;
  }

  amgcl::mpi::init mpi(&argc, &argv);
  amgcl::mpi::communicator world(MPI_COMM_WORLD);

  vex::Context ctx(vex::Filter::Exclusive(vex::Filter::Count(1)));
  for (int i = 0; i < world.size; ++i) {
    // unclutter the output:
    if (i == world.rank)
      std::cout << world.rank << ":: " << ctx.queue(0) << std::endl;
    MPI_Barrier(world);
  }

  // The profiler:
  amgcl::profiler<> prof("poisson3Db MPI(VexCL)");

  // Read the system matrix and the RHS:
  prof.tic("read");
  // Get the global size of the matrix:
  ptrdiff_t rows = amgcl::io::crs_size<ptrdiff_t>(argv[1]);
  ptrdiff_t cols;

  // Split the matrix into approximately equal chunks of rows
  ptrdiff_t chunk = (rows + world.size - 1) / world.size;
  ptrdiff_t row_beg = std::min(rows, chunk * world.rank);
  ptrdiff_t row_end = std::min(rows, row_beg + chunk);
  chunk = row_end - row_beg;

  // Read our part of the system matrix and the RHS.
  std::vector<ptrdiff_t> ptr, col;
  std::vector<double> val, rhs;
```

(continues on next page)
amgcl::io::read_crs(argv[1], rows, ptr, col, val, row_beg, row_end);
amgcl::io::read_dense(argv[2], rows, cols, rhs, row_beg, row_end);
prof.toc("read");

// Copy the RHS vector to the backend:

vex::vector<double> f(ctx, rhs);

if (world.rank == 0)
    std::cout
    << "World size: " << world.size << std::endl
    << "Matrix " << argv[1] << ": " << rows << "x" << rows << std::endl
    << "RHS " << argv[2] << ": " << rows << "x" << cols << std::endl;

// Compose the solver type

typedef amgcl::backend::vexcl<double> DBackend;
typedef amgcl::backend::vexcl<float> FBackend;
typedef amgcl::mpi::make_solver<
    amgcl::mpi::amg<
        FBackend,
        amgcl::mpi::coarsening::smoothed_aggregation<FBackend>,
        amgcl::mpi::relaxation::spai0<FBackend>
    >,
    amgcl::mpi::solver::bicgstab<DBackend>
> Solver;

// Create the distributed matrix from the local parts.
auto A = std::make_shared<amgcl::mpi::distributed_matrix<DBackend>>(
    world, std::tie(chunk, ptr, col, val));

// Partition the matrix and the RHS vector.
// If neither ParMETIS nor PT-SCOTCH are not available,
// just keep the current naive partitioning.
#if defined(AMGCL_HAVE_PARMETIS) || defined(AMGCL_HAVE_SCOTCH)
    #if defined(AMGCL_HAVE_PARMETIS)
        typedef amgcl::mpi::partition::parmetis<DBackend> Partition;
    # elif defined(AMGCL_HAVE_SCOTCH)
        typedef amgcl::mpi::partition::ptscotch<DBackend> Partition;
    # endif
    typedef amgcl::mpi::partition::parmetis<DBackend> Partition;
#endif

if (world.size > 1) {
    prof.tic("partition");
    Partition part;

    // part(A) returns the distributed permutation matrix:
    auto P = part(*A);
    auto R = transpose(*P);

    // Reorder the matrix:

    // and the RHS vector:
    vex::vector<double> new_rhs(ctx, R->loc_rows());
    R->move_to_backend<
        typename DBackend::params()
    >;
    amgcl::backend::spmv(1, *R, f, 0, new_rhs);
    f.swap(new_rhs);

    // Update the number of the local rows
Basically, we replace the built-in backend with the vexcl one, initialize the VexCL context and reference the context in the backend parameters. The RHS and the solution vectors are need to be transfered/allocated on the GPUs. Below is the output of the VexCL version using the OpenCL technology. Note that the system the tests were performed on has only two GPUs, so the test used just two MPI processes. The environment variable OMP_NUM_THREADS was set to 2 in order to fully utilize all available CPU cores:
Number of levels: 3
Operator complexity: 1.20
Grid complexity: 1.08

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>85623</td>
<td>2374949 (83.03%) [2]</td>
</tr>
<tr>
<td>1</td>
<td>6381</td>
<td>451279 (15.78%) [2]</td>
</tr>
<tr>
<td>2</td>
<td>396</td>
<td>34054 (1.19%) [2]</td>
</tr>
</tbody>
</table>

Iters: 24
Error: 9.14603e-09

[poisson3Db MPI(VexCL): 1.132 s] (100.00%)
[self: 0.040 s] (3.56%)
[partition: 0.607 s] (53.58%)
[read: 0.015 s] (1.31%)
[setup: 0.287 s] (25.31%)
[solve: 0.184 s] (16.24%)

2.4.3 Structural problem

This system may be downloaded from the Serena page (use the Matrix Market download option). According to the description, the system represents a 3D gas reservoir simulation for CO2 sequestration, and was obtained from a structural problem discretizing a gas reservoir with tetrahedral Finite Elements. The medium is strongly heterogeneous and characterized by a complex geometry consisting of alternating sequences of thin clay and sand layers. More details available in [FGJT10]. Note that the RHS vector for the Serena problem is not provided, and we use the RHS vector filled with ones.

The system matrix is symmetric, and has 1,391,349 rows and 64,131,971 nonzero values, which corresponds to an average of 46 nonzeros per row. The matrix portrait is shown on the figure below.

As in the case of Poisson problem tutorial, we start experimenting with the examples/solver utility provided by AMGCL. The default options do not seem to work this time. The relative error did not reach the required threshold of 1e-8 and the solver exited after the default limit of 100 iterations:

$ solver -A Serena.mtx
Solver
======
Type: BiCGStab
Unknowns: 1391349
Memory footprint: 74.31 M

Preconditioner
=============
Number of levels: 4
Operator complexity: 1.22
Grid complexity: 1.08
Memory footprint: 1.45 G

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1391349</td>
<td>64531701</td>
<td>1.22 G (82.01%)</td>
</tr>
<tr>
<td>1</td>
<td>98824</td>
<td>13083884</td>
<td>218.40 M (16.63%)</td>
</tr>
<tr>
<td>2</td>
<td>5721</td>
<td>1038749</td>
<td>16.82 M (1.32%)</td>
</tr>
</tbody>
</table>
Fig. 2.4: Serena matrix portrait
The system is quite large and just reading from the text-based Matrix Market format takes 18.5 seconds. No one has that amount of free time on their hands, so let's convert the matrix into the binary format with the examples/mm2bin utility. This should make the experiments slightly less painful:

```
mm2bin -i Serena.mtx -o Serena.bin
Wrote 1391349 by 1391349 sparse matrix, 64531701 nonzeros
```

The -B option tells the solver that the input is in binary format now. Let's also increase the maximum iteration limit this time to see if the solver manages to converge at all:

```
$ solver -B -A Serena.bin solver.maxiter=1000
Solver
======
Type: BiCGStab
Unknowns: 1391349
Memory footprint: 74.31 M
Preconditioner
==============
Number of levels: 4
Operator complexity: 1.22
Grid complexity: 1.08
Memory footprint: 1.45 G
```

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1391349</td>
<td>64531701</td>
<td>1.22 G (82.01%)</td>
</tr>
<tr>
<td>1</td>
<td>98824</td>
<td>13083884</td>
<td>218.40 M (16.63%)</td>
</tr>
<tr>
<td>2</td>
<td>5721</td>
<td>1038749</td>
<td>16.82 M (1.32%)</td>
</tr>
<tr>
<td>3</td>
<td>279</td>
<td>29151</td>
<td>490.75 K (0.04%)</td>
</tr>
</tbody>
</table>

Iterations: 211
Error: 8.54558e-09

```
[Profile: 114.703 s] (100.00%)
[ reading: 0.550 s] ( 0.48%)
[ setup: 2.114 s] ( 1.84%)
[ solve: 112.034 s] ( 97.67%)
```

The input matrix is read much faster now, and the solver does converge, but the convergence rate is not great. Looking closer at the Serena matrix portrait figure, the matrix seems to have block structure with $3 \times 3$ blocks. This is usually the case when the system has been obtained via discretization of a system of coupled PDEs, or has vector unknowns. We have to guess here, but since the problem is described as “structural”, then each block probably corresponds to the 3D displacement vector of a single grid node. We can communicate this piece of information to AMGCL using the block_size parameter of the aggregation method:
This has definitely improved the convergence! We also know that the matrix is symmetric, so let's switch the solver from the default BiCGStab to the slightly less expensive CG:

```bash
$ solver -B -A Serena.bin \
   solver.type=cg \ 
   solver.maxiter=1000 \ 
   precond.coarsening.aggr.block_size=3
Solver
=======
Type:       CG
Unknowns:   1391349
Memory footprint: 42.46 M

Preconditioner
==============
Number of levels:  4
Operator complexity:  1.31
Grid complexity:    1.08
Memory footprint:   1.84 G

level  unknowns  nonzeros  memory
-------------------------------
  0  1391349  64531701  1.50 G (76.48%)
  1  109764   17969220  316.66 M (21.30%)
  2   6291   1788507  29.51 M ( 2.12%)
  3   429    82719   1.23 M ( 0.10%)
[Profile: 73.296 s] (100.00%)
[ reading: 0.587 s] ( 0.80%)
[ setup: 2.709 s] ( 3.70%)
[ solve: 69.994 s] ( 95.49%)
```
This reduces the solution time, even though the number of iterations has grown. Each iteration of BiCGStab costs about twice as much as a CG iteration, because BiCGStab does two matrix-vector products and preconditioner applications per iteration, while CG only does one.

The problem description states that the medium is strongly heterogeneous and characterized by a complex geometry consisting of alternating sequences of thin clay and sand layers. This may result in high contrast between matrix coefficients in the neighboring rows, which is confirmed by the plot of the matrix diagonal in Serena matrix portrait: the diagonal coefficients span more than 10 orders of magnitude! Scaling the matrix (so that it has the unit diagonal) should help with the convergence. The $-s$ option tells the solver to do that:

```
$ solver -B -A Serena.bin -s \
    solver.type=cg solver.maxiter=200 \
    precond.coarsening.aggr.block_size=3
```

And the convergence has indeed been improved! Finally, when the matrix has block structure, as in this case, it often pays to use the block-valued backend, so that the system matrix has three times fewer rows and columns, but each nonzero entry is a statically sized $3 \times 3$ matrix. This should be done instead of specifying the block_size aggregation parameter, as the aggregation now naturally operates with the $3 \times 3$ blocks:

```
$ solver -B -A Serena.bin solver.type=cg solver.maxiter=200 -s -b3
```
Solver
======
Type: CG
Unknowns: 463783
Memory footprint: 42.46 M

Preconditioner
==============
Number of levels: 4
Operator complexity: 1.27
Grid complexity: 1.08
Memory footprint: 1.04 G

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>463783</td>
<td>7170189</td>
<td>891.53 M (78.80%)</td>
</tr>
<tr>
<td>1</td>
<td>33052</td>
<td>1772434</td>
<td>159.22 M (19.48%)</td>
</tr>
<tr>
<td>2</td>
<td>1722</td>
<td>151034</td>
<td>12.72 M (1.66%)</td>
</tr>
<tr>
<td>3</td>
<td>98</td>
<td>5756</td>
<td>612.72 K (0.06%)</td>
</tr>
</tbody>
</table>

Iterations: 162
Error: 9.7497e-09

Preconditioner now requires 1.04G of memory as opposed to 1.82G in the scalar case. The setup is about 2.5 times faster, and the solution phase performance has been slightly improved, even though the number of iteration has grown. This is explained by the fact that the matrix is now symbolically smaller, and is easier to analyze during setup. The matrix also occupies less memory for the CRS arrays, and is more cache-friendly, which helps to speed up the solution phase. This seems to be the best we can get with this system, so let us implement this version. We will also use the mixed precision approach in order to get as much performance as possible from the solution. The listing below shows the complete solution and is also available in tutorial/2.Serena/serena.cpp.

Listing 2.8: The source code for the solution of the Serena problem.

```cpp
#include <vector>
#include <iostream>
#include <amgcl/backend/builtin.hpp>
#include <amgcl/adapter/crs_tuple.hpp>
#include <amgcl/make_solver.hpp>
#include <amgcl/amg.hpp>
#include <amgcl/coarsening/smoothed_aggregation.hpp>
#include <amgcl/relaxation/spai0.hpp>
#include <amgcl/solver/cg.hpp>
#include <amgcl/value_type/static_matrix.hpp>
#include <amgcl/adapter/block_matrix.hpp>
#include <amgcl/io/mm.hpp>
#include <amgcl/profiler.hpp>

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

(continues on next page)
// The command line should contain the matrix file name:
if (argc < 2) {
    std::cerr << "Usage: " << argv[0] << " <matrix.mtx>" << std::endl;
    return 1;
}

// The profiler:
amgcl::profiler<> prof("Serena");

// Read the system matrix:
ptrdiff_t rows, cols;
std::vector<ptrdiff_t> ptr, col;
std::vector<double> val;

prof.tic("read");
std::tie(rows, cols) = amgcl::io::mm_reader(argv[1])(ptr, col, val);
std::cout << "Matrix " << argv[1] << ": " << rows << "x" << cols << std::endl;
prof.toc("read");

// The RHS is filled with ones:
std::vector<double> f(rows, 1.0);

// Scale the matrix so that it has the unit diagonal.
// First, find the diagonal values:
std::vector<double> D(rows, 1.0);
for (ptrdiff_t i = 0; i < rows; ++i) {
    for (ptrdiff_t j = ptr[i], e = ptr[i+1]; j < e; ++j) {
        if (col[j] == i) {
            D[i] = 1 / sqrt(val[j]);
            break;
        }
    }
}

// Then, apply the scaling in-place:
for (ptrdiff_t i = 0; i < rows; ++i) {
    for (ptrdiff_t j = ptr[i], e = ptr[i+1]; j < e; ++j) {
        val[j] *= D[i] * D[col[j]];
    }
    f[i] *= D[i];
}

// We use the tuple of CRS arrays to represent the system matrix.
// Note that std::tie creates a tuple of references, so no data is actually
// copied here:
auto A = std::tie(rows, ptr, col, val);

// Compose the solver type
typedef amgcl::static_matrix<double, 3, 3> dmat_type; // matrix value type in
double precision
typedef amgcl::static_matrix<double, 3, 1> dvec_type; // the corresponding vector
value type
typedef amgcl::static_matrix<float, 3, 3> smat_type; // matrix value type in
single precision
typedef amgcl::backend::builtin<dmat_type> SBackend; // the solver backend
typedef amgcl::backend::builtin<smat_type> PBackend; // the preconditioner backend

(continues on next page)
typedef amgcl::make_solver<
    amgcl::amg<PBackend,
    amgcl::coarsening::smoothed_aggregation,
    amgcl::relaxation::spai0
    >,
    amgcl::solver::cg<SBackend>
    > Solver;

// Solver parameters
Solver::params prm;
prm.solver.maxiter = 500;

// Initialize the solver with the system matrix.
// Use the block_matrix adapter to convert the matrix into
// the block format on the fly:
prof.tic("setup");
auto Ab = amgcl::adapter::block_matrix<dmat_type>(A);
Solver solve(Ab, prm);
prof.toc("setup");

// Show the mini-report on the constructed solver:
std::cout << solve << std::endl;

// Solve the system with the zero initial approximation:
int iters;
double error;
std::vector<double> x(rows, 0.0);

// Reinterpret both the RHS and the solution vectors as block-valued:
auto f_ptr = reinterpret_cast<dvec_type*>(f.data());
auto x_ptr = reinterpret_cast<dvec_type*>(x.data());
auto F = amgcl::make_iterator_range(f_ptr, f_ptr + rows / 3);
auto X = amgcl::make_iterator_range(x_ptr, x_ptr + rows / 3);

prof.tic("solve");
std::tie(iters, error) = solve(Ab, F, X);
prof.toc("solve");

// Output the number of iterations, the relative error,
// and the profiling data:
std::cout << "Iters: " << iters << std::endl
    << "Error: " << error << std::endl
    << prof << std::endl;
}

In addition the the includes described in Poisson problem, we also include the headers for the
amgcl::static_matrix value type, and the amgcl::adapter::block_matrix() adapter that transpar-
ently converts a scalar matrix to the block format. In lines 42–58 we apply the scaling according to the following
formula:

\[ A_s = D^{-1/2} A D^{-1/2}, \quad f_s = D^{-1/2} f \]

where \( A_s \) and \( f_s \) are the scaled matrix and the RHS vector, and \( D \) is the diagonal of the matrix \( A \). After solving the
scaled system \( A_s y = f_s \), the solution to the original system may be found as \( x = D^{-1/2} y \).

In lines 66–68 we define the block value types for the matrix and the RHS and solution vectors. dmat_type and
smat_type are $3 \times 3$ static matrices used as value types with the double precision solver backend and the single precision preconditioner backend. dvec_type is a double precision $3 \times 1$ matrix (or a vector) used as a value type for the RHS and the solution.

The solver class definition in lines 73–80 is almost the same as in the Poisson problem case, with the exception that we are using the CG iterative solver this time. In lines 83–84 we define the solver parameters. Namely, we increase the maximum iterations limit to 500 iterations.

In lines 90–91 we instantiate the solver, using the block_matrix adapter in order to convert the scalar matrix into the block format. The adapter operates on a row-by-row basis and does not create a temporary copy of the matrix.

In lines 103–106 we convert the scalar RHS and solution vectors to the block-valued ones. We use the fact that 3 consecutive elements of a scalar array may be reinterpreted as a single $3 \times 1$ static matrix. Using the reinterpret_cast trick we can get the block-valued view into the RHS and the solution vectors data without extra memory copies.

Here is the output of the program:

```
$ ./serena Serena.mtx
Matrix Serena.mtx: 1391349x1391349
Solver
======
Type: CG
Unknowns: 463783
Memory footprint: 42.46 M

Preconditioner
==============
Number of levels: 4
Operator complexity: 1.27
Grid complexity: 1.08
Memory footprint: 585.33 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>463783</td>
<td>7170189</td>
<td>490.45 M (78.80%)</td>
</tr>
<tr>
<td>1</td>
<td>33052</td>
<td>1772434</td>
<td>87.58 M (19.48%)</td>
</tr>
<tr>
<td>2</td>
<td>1722</td>
<td>151034</td>
<td>7.00 M (1.66%)</td>
</tr>
<tr>
<td>3</td>
<td>98</td>
<td>5756</td>
<td>306.75 K (0.06%)</td>
</tr>
</tbody>
</table>

Iters: 162
Error: 9.74929e-09

[Serena: 48.427 s] (100.00%)
[ self: 0.166 s] ( 0.34%)
[ read: 21.115 s] ( 43.60%)
[ setup: 0.749 s] ( 1.55%)
[ solve: 26.397 s] (54.51%)
```

Note that due to the use of mixed precision the preconditioner consumes 585.33M of memory as opposed to 1.08G from the example above. The setup and the solution are faster that the full precision version by about 30% and 10% correspondingly.

Let us see if using a GPU backend may further improve the performance. The CUDA backend does not support block value types, so we will use the VexCL backend (which, in turn, may use either OpenCL, CUDA, or OpenMP). The listing below contains the complete source for the solution (available at tutorial/2.Serena/serena_vexcl.cpp). The differences with the builtin backend version are highlighted.
Listing 2.9: The solution of the Serena problem with the VexCL backend.

```cpp
#include <vector>
#include <iostream>

#include <amgcl/backend/vexcl.hpp>
#include <amgcl/backend/vexcl_static_matrix.hpp>
#include <amgcl/adapter/crs_tuple.hpp>
#include <amgcl/make_solver.hpp>
#include <amgcl/amg.hpp>
#include <amgcl/coarsening/smoothed_aggregation.hpp>
#include <amgcl/relaxation/spai0.hpp>
#include <amgcl/solver/cg.hpp>
#include <amgcl/value_type/static_matrix.hpp>
#include <amgcl/adapter/block_matrix.hpp>

#include <amgcl/io/mm.hpp>
#include <amgcl/profiler.hpp>

int main(int argc, char *argv[]) {
    // The command line should contain the matrix file name:
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " <matrix.mtx>" << std::endl;
        return 1;
    }

    // Create VexCL context. Set the environment variable OCL_DEVICE to
    // control which GPU to use in case multiple are available,
    // and use single device:
    vex::Context ctx(vex::Filter::Env && vex::Filter::Count(1));
    std::cout << ctx << std::endl;

    // Enable support for block-valued matrices in the VexCL kernels:
    vex::scoped_program_header h1(ctx, amgcl::backend::vexcl_static_matrix_declaration
    <double, 3>());
    vex::scoped_program_header h2(ctx, amgcl::backend::vexcl_static_matrix_declaration
    <float, 3>());

    // The profiler:
    amgcl::profiler<> prof("Serena (VexCL)");

    // Read the system matrix:
    ptrdiff_t rows, cols;
    std::vector<ptrdiff_t> ptr, col;
    std::vector<double> val;

    prof.tic("read");
    std::tie(rows, cols) = amgcl::io::mm_reader(argv[1])(ptr, col, val);
    std::cout << "Matrix " << argv[1] << ": " " rows " " x " cols << std::endl;
    prof.toc("read");

    // The RHS is filled with ones:
    std::vector<double> f(rows, 1.0);

    // Scale the matrix so that it has the unit diagonal.
    // First, find the diagonal values:
    std::vector<double> D(rows, 1.0);
```
for (ptrdiff_t i = 0; i < rows; ++i) {
    for (ptrdiff_t j = ptr[i], e = ptr[i+1]; j < e; ++j) {
        if (col[j] == i) {
            D[i] = 1 / sqrt(val[j]);
            break;
        }
    }
}

// Then, apply the scaling in-place:
for (ptrdiff_t i = 0; i < rows; ++i) {
    for (ptrdiff_t j = ptr[i], e = ptr[i+1]; j < e; ++j) {
        val[j] *= D[i] * D[col[j]];
    }
    f[i] *= D[i];
}

// We use the tuple of CRS arrays to represent the system matrix.
// Note that std::tie creates a tuple of references, so no data is actually
// copied here:
auto A = std::tie(rows, ptr, col, val);

// Compose the solver type
typedef amgcl::static_matrix<double, 3, 3> dmat_type; // matrix value type in
    // double precision
typedef amgcl::static_matrix<double, 3, 1> dvec_type; // the corresponding vector
        // value type
typedef amgcl::static_matrix<float, 3, 3> smat_type; // matrix value type in
    // single precision

typedef amgcl::backend::vexcl<dmat_type> SBackend; // the solver backend
typedef amgcl::backend::vexcl<smat_type> PBackend; // the preconditioner backend

typedef amgcl::make_solver<
    amgcl::amg<
        PBackend,
        amgcl::coarsening::smoothed_aggregation,
        amgcl::relaxation::spai0
    >,
    amgcl::solver::cg<SBackend>
> Solver;

// Solver parameters
Solver::params prm;
prm.solver.maxiter = 500;

// Set the VexCL context in the backend parameters
SBackend::params bprm;
bprm.q = ctx;

// Initialize the solver with the system matrix.
// We use the block_matrix adapter to convert the matrix into the block
// format on the fly:
prof.tic("setup");
auto Ab = amgcl::adapter::block_matrix<dmat_type>(A);
Solver solve(Ab, prm, bprm);
prof.toc("setup");

(continues on next page)
In the include section, we replace the header for the builtin backend with the one for the VexCL backend, and also include the header with support for block values in VexCL (lines 4–5). In lines 28–29 we initialize the VexCL context, and in lines 32–33 we enable the VexCL support for $3 \times 3$ static matrices in both double and single precision.

In lines 81–82 we define the solver and preconditioner backends as VexCL backends with the corresponding matrix value types. In lines 98–99 we reference the VexCL context in the backend parameters.

Since we are using the GPU backend, we have to explicitly form the block valued matrix and transfer it to the GPU. This is done in lines 119–120. In lines 127–128 we copy the RHS and the solution vectors to the GPU, and we solve the system in line 131.

The output of the program is shown below:

```
$ ./serena_vexcl_cuda Serena.mtx
1. GeForce GTX 1050 Ti
Matrix Serena.mtx: 1391349x1391349
Solver ======
Type:       CG
Unknowns:  463783
Memory footprint: 42.46 M
```

(continues on next page)
Preconditioner
===============
Number of levels: 4
Operator complexity: 1.27
Grid complexity: 1.08
Memory footprint: 585.33 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>463783</td>
<td>7170189</td>
<td>490.45 M (78.80%)</td>
</tr>
<tr>
<td>1</td>
<td>33052</td>
<td>1772434</td>
<td>87.58 M (19.48%)</td>
</tr>
<tr>
<td>2</td>
<td>1722</td>
<td>151034</td>
<td>7.00 M (1.66%)</td>
</tr>
<tr>
<td>3</td>
<td>98</td>
<td>5756</td>
<td>309.04 K (0.06%)</td>
</tr>
</tbody>
</table>

Iters: 162
Error: 9.74928e-09

[Serena (VexCL): 27.208 s] (100.00%)
[ self: 0.180 s] ( 0.66%)
[ GPU matrix: 0.604 s] ( 2.22%)
[ read: 18.699 s] ( 68.73%)
[ setup: 1.308 s] ( 4.81%)
[ solve: 6.417 s] ( 23.59%)

The setup time has increased from 0.7 seconds for the builtin backend to 1.3 seconds, and we also see the additional 0.6 seconds for transferring the matrix to the GPU. But the solution time has decreased from 26.4 to 6.4 seconds, which is about 4 times faster.

2.4.4 Structural problem (MPI version)

In this section we look at how to use the MPI version of the AMGCL solver with the Serena system. We have already determined in the Structural problem section that the system is best solved with the block-valued backend, and needs to be scaled so that it has the unit diagonal. The MPI solution will be very closer to the one we have seen in the Poisson problem (MPI version) section. The only differences are:

- The system needs to be scaled so that it has the unit diagonal. This is complicated by the fact that the matrix product $D^{-1/2} A D^{-1/2}$ has to done in the distributed memory environment.
- The solution has to use the block-valued backend, and the partitioning needs to take this into account. Namely, the partitioning should not split any of the $3 \times 3$ blocks between MPI processes.
- Even though the system is symmetric, the convergence of the CG solver in the distributed case stalls at the relative error of about $10^{-6}$. Switching to the BiCGStab solver helps with the convergence.

The next listing is the MPI version of the Serena system solver (tutorial/2.Serena/serena_mpi.cpp). In the following paragraphs we highlight the differences between this version and the code in the Poisson problem (MPI version) and Structural problem sections.

Listing 2.10: The MPI solution of the Serena problem

```cpp
#include <vector>
#include <iostream>

#include <amgcl/backend/builtin.hpp>
#include <amgcl/value_type/static_matrix.hpp>
#include <amgcl/adapter/crs_tuple.hpp>
```
```cpp
#include <amgcl/adapter/block_matrix.hpp>
#include <amgcl/mpi/distributed_matrix.hpp>
#include <amgcl/mpi/make_solver.hpp>
#include <amgcl/mpi/amg.hpp>
#include <amgcl/mpi/coarsening/smoothed_aggregation.hpp>
#include <amgcl/mpi/relaxation/spai0.hpp>
#include <amgcl/mpi/solver/bicgstab.hpp>
#include <amgcl/io/binary.hpp>
#include <amgcl/profiler.hpp>

#ifdef(AMGCL_HAVE_PARMETIS)
#include <amgcl/mpi/partition/parmetis.hpp>
#endif
#ifdef(AMGCL_HAVE_SCOTCH)
#include <amgcl/mpi/partition/ptscotch.hpp>
#endif

// Block size
const int B = 3;

int main(int argc, char *argv[]) { 
    // The command line should contain the matrix file name:
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " <matrix.bin>" << std::endl;
        return 1;
    }

    amgcl::mpi::init mpi(&argc, &argv);
    amgcl::mpi::communicator world(MPI_COMM_WORLD);

    // The profiler:
    amgcl::profiler<> prof("Serena MPI");
    prof.tic("read");
    // Get the global size of the matrix:
    ptrdiff_t rows = amgcl::io::crs_size<ptrdiff_t>(argv[1]);

    // Split the matrix into approximately equal chunks of rows, and
    // make sure each chunk size is divisible by the block size.
    ptrdiff_t chunk = (rows + world.size - 1) / world.size;
    if (chunk % B) chunk += B - chunk % B;

    ptrdiff_t row_beg = std::min(rows, chunk * world.rank);
    ptrdiff_t row_end = std::min(rows, row_beg + chunk);
    chunk = row_end - row_beg;

    // Read our part of the system matrix.
    std::vector<ptrdiff_t> ptr, col;
    std::vector<double> val;
    amgcl::io::read_crs(argv[1], rows, ptr, col, val, row_beg, row_end);
    prof.toc("read");
    if (world.rank == 0) std::cout
        << "World size: " << world.size << std::endl
        << "Matrix " << argv[1] << ": " << rows << "x" << rows << std::endl;
```
// Declare the backend and the solver types
typedef amgcl::static_matrix<double, B, B> dmat_type;
typedef amgcl::static_matrix<double, B, 1> dvec_type;
typedef amgcl::static_matrix<float, B, B> fmat_type;
typedef amgcl::backend::builtin<dmat_type> DBackend;
typedef amgcl::backend::builtin<fmat_type> FBackend;

typedef amgcl::mpi::make_solver<
    amgcl::mpi::amg<
        FBackend,
        amgcl::mpi::coarsening::smoothed_aggregation<FBackend>,
        amgcl::mpi::relaxation::spai0<FBackend>
    >,
    amgcl::mpi::solver::bicgstab<DBackend>
> Solver;

// Solver parameters
Solver::params prm;
prm.solver.maxiter = 200;

// We need to scale the matrix, so that it has the unit diagonal.
// Since we only have the local rows for the matrix, and we may need the
// remote diagonal values, it is more convenient to represent the scaling
// with the matrix-matrix product (As = D^-1/2 A D^-1/2).
prof.tic("scale");
// Find the local diagonal values,
// and form the CRS arrays for a diagonal matrix.
std::vector<double> dia(chunk, 1.0);
std::vector<ptrdiff_t> d_ptr(chunk + 1), d_col(chunk);
for(ptrdiff_t i = 0, I = row_beg; i < chunk; ++i, ++I) {
    d_ptr[i] = i;
    d_col[i] = I;
    for(ptrdiff_t j = ptr[i], e = ptr[i+1]; j < e; ++j) {
        if (col[j] == I) {
            dia[i] = 1 / sqrt(val[j]);
            break;
        }
    }
}
d_ptr.back() = chunk;

// Create the distributed diagonal matrix:
amgcl::mpi::distributed_matrix<DBackend> D(world,
    amgcl::adapter::block_matrix<dmat_type>(
        std::tie(chunk, d_ptr, d_col, dia)));

// The scaled matrix is formed as product D * A * D,
// where A is the local chunk of the matrix
// converted to the block format on the fly.
auto A = product(0, product(
    amgcl::mpi::distributed_matrix<DBackend>(world,
        amgcl::adapter::block_matrix<dmat_type>(
            std::tie(chunk, ptr, col, val)));
    D));
prof.toc("scale");
// Since the RHS in this case is filled with ones, 
// the scaled RHS is equal to dia.

auto f_ptr = reinterpret_cast<dvec_type*>(dia.data());
std::vector<dvec_type> rhs(f_ptr, f_ptr + chunk / B);

// Reinterpret the pointer to dia data to get the RHS in the block format:

// Partition the matrix and the RHS vector.
// If neither ParMETIS nor PT-SCOTCH are not available,
// just keep the current naive partitioning.
#ifdef AMGCL_HAVE_PARMETIS
#ifdef AMGCL_HAVE_SCOTCH
typedef amgcl::mpi::partition::parmetis<DBackend> Partition;
#elif defined(AMGCL_HAVE_SCOTCH)
typedef amgcl::mpi::partition::ptscotch<DBackend> Partition;
#endif
#endif

if (world.size > 1) {
  prof.tic("partition");
  Partition part;

  // part(A) returns the distributed permutation matrix:
  auto P = part(*A);
  auto R = transpose(*P);

  // Reorder the matrix:

  // and the RHS vector:
  std::vector<dvec_type> new_rhs(R->loc_rows());
  R->move_to_backend();
  amgcl::backend::spmv(1, *R, rhs, 0, new_rhs);
  rhs.swap(new_rhs);

  // Update the number of the local rows 
  // (it may have changed as a result of permutation).
  // Note that A->loc_rows() returns the number of blocks,
  // as the matrix uses block values.
  chunk = A->loc_rows();
  prof.toc("partition");
}

// Initialize the solver:
prof.tic("setup");
Solver solve(world, A, prm);
prof.toc("setup");

// Show the mini-report on the constructed solver:
if (world.rank == 0) std::cout << solve << std::endl;

// Solve the system with the zero initial approximation:
int iters;
double error;
std::vector<dvec_type> x(chunk, amgcl::math::zero<dvec_type>());

prof.tic("solve");
std::tie(iters, error) = solve(*A, rhs, x);
178 prof.toc("solve");

179 // Output the number of iterations, the relative error,
180 // and the profiling data:
181 if (world.rank == 0) std::cout
182  << "Iterations: " << iters << std::endl
183  << "Error: " << error << std::endl
184  << prof << std::endl;

186 }

We make sure that the partitioning takes the block structure of the matrix into account by keeping the number of rows in the initial naive partitioning divisible by 3 (here the constant $B$ is equal to 3):

46 // Split the matrix into approximately equal chunks of rows, and
47 // make sure each chunk size is divisible by the block size.
48 ptrdiff_t chunk = (rows + world.size - 1) / world.size;
49 if (chunk % B) chunk += B - chunk % B;
50
51 ptrdiff_t row_beg = std::min(rows, chunk * world.rank);
52 ptrdiff_t row_end = std::min(rows, row_beg + chunk);
53 chunk = row_end - row_beg;

We also create all the distributed matrices using the block values, so the partitioning naturally is block-aware. We are using the mixed precision approach, so the preconditioner backend is defined with the single precision:

65 // Declare the backend and the solver types
66 typedef amgcl::static_matrix<
67 double, B, B> dmat_type;
68 typedef amgcl::static_matrix<
69 double, B, 1> dvec_type;
70 typedef amgcl::static_matrix<
71 float, B, B> fmat_type;
72 typedef amgcl::backend::builtin<dmat_type> DBackend;
73 typedef amgcl::backend::builtin<fmat_type> FBackend;
74
75 typedef amgcl::mpi::make_solver<
76 amgcl::mpi::amg<
77 FBackend,
78 amgcl::mpi::coarsening::smoothed_aggregation<
79 FBackend>,
80 amgcl::mpi::relaxation::spai0<
81 FBackend>
82 >,
83 amgcl::mpi::solver::bicgstab<
84 DBackend>
85 > Solver;

The scaling is done similarly to how we apply the reordering: first, we find the diagonal of the local diagonal block on each of the MPI processes, and then we create the distributed diagonal matrix with the inverted square root of the system matrix diagonal. After that, the scaled matrix $A_s = D^{-1/2}AD^{-1/2}$ is computed using the amgcl::mpi::product() function. The scaled RHS vector $f_s = D^{-1/2}f$ in principle may be found using the amgcl::backend::spmv() primitive, but, since the RHS vector in this case is simply filled with ones, the scaled RHS $f_s = D^{-1/2}$.

85 // We need to scale the matrix, so that it has the unit diagonal.
86 // Since we only have the local rows for the matrix, and we may need the
87 // remote diagonal values, it is more convenient to represent the scaling
88 // with the matrix-matrix product (As = D^-1/2 AD^-1/2).
89 prof.tic("scale");
90 // Find the local diagonal values,
91 // and form the CRS arrays for a diagonal matrix.
92 std::vector<double> dia(chunk, 1.0);
std::vector<ptrdiff_t> d_ptr(chunk + 1), d_col(chunk);
for(ptrdiff_t i = 0, I = row_beg; i < chunk; ++i, ++I) {
    d_ptr[i] = i;
    d_col[i] = I;
    for(ptrdiff_t j = ptr[i], e = ptr[i+1]; j < e; ++j) {
        if (col[j] == I) {
            dia[i] = 1 / sqrt(val[j]);
            break;
        }
    }
}  

d_ptr.back() = chunk;

// Create the distributed diagonal matrix:
    amgcl::mpi::distributed_matrix<DBackend> D(world,
        amgcl::adapter::block_matrix<dmat_type>(
            std::tie(chunk, d_ptr, d_col, dia)));

// The scaled matrix is formed as product D * A * D,
// where A is the local chunk of the matrix
// converted to the block format on the fly.
    auto A = product(D, *product(
        amgcl::mpi::distributed_matrix<DBackend>(world,
            amgcl::adapter::block_matrix<dmat_type>(
                std::tie(chunk, ptr, col, val))));

// Since the RHS in this case is filled with ones,
// the scaled RHS is equal to dia.
// Reinterpret the pointer to dia data to get the RHS in the block format:
    auto f_ptr = reinterpret_cast<dvec_type*>(dia.data());
    std::vector<dvec_type> rhs(f_ptr, f_ptr + chunk / B);

Here is the output from the compiled program:

$ export OMP_NUM_THREADS=1
$ mpirun -np 4 ./serena_mpi Serena.bin
World size: 4
Matrix Serena.bin: 1391349x1391349
Partitioning[ParMETIS] 4 -> 4
Type: BiCGStab
Unknowns: 118533
Memory footprint: 18.99 M

Number of levels: 4
Operator complexity: 1.27
Grid complexity: 1.07

level unknowns nonzeros
--------------------
0 463783 7170189 (79.04%) [4]
1 32896 1752778 (19.32%) [4]
2 1698 144308 ( 1.59%) [4]
3 95 4833 ( 0.05%) [4]

Iterations: 80

(continues on next page)
The version that uses the VexCL backend should be familiar at this point. Below is the source code (tutorial/2.Serena/serena_mpi_vexcl.cpp) where the differences with the builtin backend version are highlighted:

Listing 2.11: The MPI solution of the Serena problem using the VexCL backend

```cpp
#include <vector>
#include <iostream>

#include <amgcl/backend/vexcl.hpp>
#include <amgcl/backend/vexcl_static_matrix.hpp>
#include <amgcl/value_type/static_matrix.hpp>
#include <amgcl/adapter/crs_tuple.hpp>
#include <amgcl/adapter/block_matrix.hpp>

#include <amgcl/mpi/distributed_matrix.hpp>
#include <amgcl/mpi/make_solver.hpp>
#include <amgcl/mpi/amg.hpp>
#include <amgcl/mpi/coarsening/smoothed_aggregation.hpp>
#include <amgcl/mpi/relaxation/spai0.hpp>
#include <amgcl/mpi/solver/bicgstab.hpp>

#include <amgcl/io/binary.hpp>
#include <amgcl/profiler.hpp>

#if defined(AMGCL_HAVE_PARMETIS)
# include <amgcl/mpi/partition/parmetis.hpp>
#elif defined(AMGCL_HAVE_SCOTCH)
# include <amgcl/mpi/partition/ptscotch.hpp>
#endif

// Block size
const int B = 3;

// Create VexCL context. Use vex::Filter::Exclusive so that different MPI processes get different GPUs. Each process gets a single GPU:
```
42 vex::Context ctx(vex::Filter::Exclusive(vex::Filter::Env &&
43 vex::Filter::Count(1)));
44 for(int i = 0; i < world.size; ++i) {
45 // unclutter the output:
46 if (i == world.rank)
47 std::cout << world.rank << ": " << ctx.queue(0) << std::endl;
48 MPI_Barrier(world);
49 }
50 // Enable support for block-valued matrices in the VexCL kernels:
51 vex::scoped_program_header h1(ctx, amgcl::backend::vexcl_static_matrix_declaration
52 <double,B>();
53 vex::scoped_program_header h2(ctx, amgcl::backend::vexcl_static_matrix_declaration
54 <float,B>());
55 // The profiler:
56 amgcl::profiler<> prof("Serena MPI(VexCL)");  
57 prof.tic("read");
58 // Get the global size of the matrix:
59 ptrdiff_t rows = amgcl::io::crs_size<ptrdiff_t>(argv[1]);
60 // Split the matrix into approximately equal chunks of rows, and
61 // make sure each chunk size is divisible by the block size.
62 ptrdiff_t chunk = (rows + world.size - 1) / world.size;
63 if (chunk % B) chunk += B - chunk % B;
64 ptrdiff_t row_beg = std::min(rows, chunk * world.rank);
65 ptrdiff_t row_end = std::min(rows, row_beg + chunk);
66 chunk = row_end - row_beg;
67 // Read our part of the system matrix.
68 std::vector<ptrdiff_t> ptr, col;
69 std::vector<double> val;
70 amgcl::io::read_crs(argv[1], rows, ptr, col, val, row_beg, row_end);
71 prof.toc("read");
72 if (world.rank == 0) std::cout
73 "World size: " << world.size << std::endl
74 "Matrix " << argv[1] << ": " << rows << "x" << rows << std::endl;
75 // Declare the backend and the solver types
76 typedef amgcl::static_matrix<double, B, B> dmat_type;
77 typedef amgcl::static_matrix<double, B, 1> dvec_type;
78 typedef amgcl::static_matrix<float, B, B> fmat_type;
79 typedef amgcl::backend::vexcl<dmat_type> DBackend;
80 typedef amgcl::backend::vexcl<fmat_type> FBackend;
81  
82 typedef amgcl::mpi::make_solver<
83 amgcl::mpi::amg<
84 DBackend,
85 amgcl::mpi::coarsening::smoothed_aggregation<FBackend>,
86 amgcl::mpi::relaxation::spai0<FBackend>>>,
87 amgcl::mpi::solver::bicgstab<DBackend>
88 > Solver;
// Solver parameters
Solver::params prm;
prm.solver.maxiter = 200;

// Set the VexCL context in the backend parameters
DBackend::params bprm;
bprm.q = ctx;

// We need to scale the matrix, so that it has the unit diagonal.
// Since we only have the local rows for the matrix, and we may need the
// remote diagonal values, it is more convenient to represent the scaling
// with the matrix-matrix product (As = D^{-1/2} A D^{-1/2}).
prof.tic("scale");
// Find the local diagonal values,
// and form the CRS arrays for a diagonal matrix.
std::vector<double> dia(chunk, 1.0);
std::vector<ptrdiff_t> d_ptr(chunk + 1), d_col(chunk);
for(ptrdiff_t i = 0, I = row_beg; i < chunk; ++i, ++I) {
    d_ptr[i] = i;
    d_col[i] = I;
    for(ptrdiff_t j = ptr[i], e = ptr[i+1]; j < e; ++j) {
        if (col[j] == I) {
            dia[i] = 1 / sqrt(val[j]);
            break;
        }
    }
}
d_ptr.back() = chunk;

// Create the distributed diagonal matrix:
amgcl::mpi::distributed_matrix<DBackend> D(world,
    amgcl::adapter::block_matrix<dmat_type>(
        std::tie(chunk, d_ptr, d_col, dia)));

// The scaled matrix is formed as product D * A * D,
// where A is the local chunk of the matrix
// converted to the block format on the fly.
auto A = product(D, *product(
    amgcl::mpi::distributed_matrix<DBackend>(world,
        amgcl::adapter::block_matrix<dmat_type>(
            std::tie(chunk, ptr, col, val)));
    D));
prof.toc("scale");

// Since the RHS in this case is filled with ones,
// the scaled RHS is equal to dia.
// Reinterpret the pointer to dia data to get the RHS in the block format:
auto f_ptr = reinterpret_cast<dvec_type*>(dia.data());
for(dvec_type* d = f_ptr; d < f_ptr + chunk / B; ++d) {
    *d = 1.0;
}

// Partition the matrix and the RHS vector.
// If neither ParMETIS not PT-SCOTCH are not available,
// just keep the current naive partitioning.
#if defined(AMGCL_HAVE_PARMETIS) || defined(AMGCL_HAVE_SCOTCH)
#define Partition amgcl::mpi::partition::parmetis<DBackend>
#else
#define Partition amgcl::mpi::partition::metis<DBackend>
#endif
Partition C;

(continues on next page)
```cpp
typedef amgcl::mpi::partition::ptscotch<DBackend> Partition;

if (world.size > 1) {
  prof.tic("partition");
  Partition part;

  // part(A) returns the distributed permutation matrix:
  auto P = part(*A);
  auto R = transpose(*P);

  // Reorder the matrix:

  // and the RHS vector:
  vex::vector<dvec_type> new_rhs(ctx, R->loc_rows());
  R->move_to_backend(bprm);
  amgcl::backend::spmv(1, *R, rhs, 0, new_rhs);
  rhs.swap(new_rhs);

  // Update the number of the local rows
  // (it may have changed as a result of permutation).
  // Note that A->loc_rows() returns the number of blocks,
  // as the matrix uses block values.
  chunk = A->loc_rows();
  prof.toc("partition");
}

// Initialize the solver:
prof.tic("setup");
Solver solve(world, A, prm, bprm);
prof.toc("setup");

// Show the mini-report on the constructed solver:
if (world.rank == 0) std::cout << solve << std::endl;

// Solve the system with the zero initial approximation:
int iters;
double error;
vex::vector<dvec_type> x(ctx, chunk);
x = amgcl::math::zero<dvec_type>();

prof.tic("solve");
std::tie(iters, error) = solve(*A, rhs, x);
prof.toc("solve");

// Output the number of iterations, the relative error,
// and the profiling data:
if (world.rank == 0) std::cout
  << "Iterations: " << iters << std::endl
  << "Error: " << error << std::endl
  << prof << std::endl;
}
```

Here is the output of the MPI version with the VexCL backend:
$ export OMP_NUM_THREADS=1
$ mpirun -np 2 ./serena_mpi_vexcl_cl Serena.bin

0: GeForce GTX 960 (NVIDIA CUDA)
1: GeForce GTX 1050 Ti (NVIDIA CUDA)

World size: 2
Matrix Serena.bin: 1391349x1391349
Partitioning[ParMETIS] 2 -> 2
Type: BiCGStab
Unknowns: 231112
Memory footprint: 37.03 M

Number of levels: 4
Operator complexity: 1.27
Grid complexity: 1.07

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>463783</td>
<td>7170189</td>
</tr>
<tr>
<td>1</td>
<td>32887</td>
<td>1754795</td>
</tr>
<tr>
<td>2</td>
<td>1708</td>
<td>146064</td>
</tr>
<tr>
<td>3</td>
<td>85</td>
<td>4059</td>
</tr>
</tbody>
</table>

Iterations: 83
Error: 9.80582e-09

[Serena MPI(VexCL): 10.943 s] (100.00%)
[ partition: 1.357 s] (12.40%)
[ read: 0.370 s] (3.38%)
[ scale: 0.729 s] (6.66%)
[ setup: 1.966 s] (17.97%)
[ solve: 6.512 s] (59.51%)

2.4.5 Fully coupled poroelastic problem

This system may be downloaded from the CoupCons3D page (use the Matrix Market download option). According to the description, the system has been obtained through a Finite Element transient simulation of a fully coupled consolidation problem on a three-dimensional domain using Finite Differences for the discretization in time. More details available in [FePG09] and [FeJP12]. The RHS vector for the CoupCons3D problem is not provided, and we use the RHS vector filled with ones.

The system matrix is non-symmetric and has 416,800 rows and 17,277,420 nonzero values, which corresponds to an average of 41 nonzeros per row. The matrix portrait is shown on the figure below.

Once again, lets start our experiments with the examples/solver utility after converting the matrix into binary format with examples/mm2bin. The default options do not seem to work for this problem:

$ solver -B -A CoupCons3D.bin
Solver
======
Type: BiCGStab
Unknowns: 416800
Memory footprint: 22.26 M

Preconditioner
=============
Number of levels: 4
Fig. 2.5: CoupCons3D matrix portrait
Operator complexity: 1.11
Grid complexity: 1.09
Memory footprint: 447.17 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>416800</td>
<td>22322336</td>
<td>404.08 M (90.13%)</td>
</tr>
<tr>
<td>1</td>
<td>32140</td>
<td>2214998</td>
<td>38.49 M (8.94%)</td>
</tr>
<tr>
<td>2</td>
<td>3762</td>
<td>206242</td>
<td>3.58 M (0.83%)</td>
</tr>
<tr>
<td>3</td>
<td>522</td>
<td>22424</td>
<td>1.03 M (0.09%)</td>
</tr>
</tbody>
</table>

Iterations: 100
Error: 0.705403

[Profile: 16.981 s] (100.00%)
[reading: 0.187 s] ( 1.10%)
[setup: 0.584 s] ( 3.44%)
[solve: 16.209 s] ( 95.45%)

What seems to works is using the higher quality relaxation (incomplete LU decomposition with zero fill-in):

$ solver -B -A CoupCons3D.bin precond.relax.type=ilu0

Solver
======
Type: BiCGStab
Unknowns: 416800
Memory footprint: 22.26 M

Preconditioner
==============
Number of levels: 4
Operator complexity: 1.11
Grid complexity: 1.09
Memory footprint: 832.12 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>416800</td>
<td>22322336</td>
<td>751.33 M (90.13%)</td>
</tr>
<tr>
<td>1</td>
<td>32140</td>
<td>2214998</td>
<td>72.91 M (8.94%)</td>
</tr>
<tr>
<td>2</td>
<td>3762</td>
<td>206242</td>
<td>6.85 M (0.83%)</td>
</tr>
<tr>
<td>3</td>
<td>522</td>
<td>22424</td>
<td>1.03 M (0.09%)</td>
</tr>
</tbody>
</table>

Iterations: 47
Error: 4.8263e-09

[Profile: 13.664 s] (100.00%)
[reading: 0.188 s] ( 1.38%)
[setup: 1.708 s] ( 12.50%)
[solve: 11.765 s] ( 86.11%)

From the matrix diagonal plot in CoupCons3D matrix portrait it is clear that the system, as in Structural problem case, has high contrast coefficients. Scaling the matrix so it has the unit diagonal should help here as well:

$ solver -B -A CoupCons3D.bin precond.relax.type=ilu0 -s

Solver
======

(continues on next page)
Type: BiCGStab
Unknowns: 416800
Memory footprint: 22.26 M

Preconditioner
==============
Number of levels: 3
Operator complexity: 1.10
Grid complexity: 1.08
Memory footprint: 834.51 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>416800</td>
<td>22322336</td>
<td>751.33 M (90.54%)</td>
</tr>
<tr>
<td>1</td>
<td>32140</td>
<td>2214998</td>
<td>73.06 M ( 8.98%)</td>
</tr>
<tr>
<td>2</td>
<td>2221</td>
<td>116339</td>
<td>10.12 M ( 0.47%)</td>
</tr>
</tbody>
</table>

Iterations: 11
Error: 9.79966e-09

[Profile: 4.826 s] (100.00%)
[ self: 0.064 s] ( 1.34%)
[ reading: 0.188 s] ( 3.90%)
[ setup: 1.885 s] ( 39.06%)
[ solve: 2.689 s] ( 55.71%)

Another thing to note from the CoupCons3D matrix portrait is that the system matrix has block structure, with two diagonal subblocks. The upper left subblock contains 333,440 unknowns and seems to have a block structure of its own with small $4 \times 4$ blocks, and the lower right subblock is a simple diagonal matrix with 83,360 unknowns. Fortunately, 83,360 is divisible by 4, so we should be able to treat the whole system as if it had $4 \times 4$ block structure:

$ solver -B -A CoupCons3D.bin precond.relax.type=ilu0 -s -b4$

Solver
=====
Type: BiCGStab
Unknowns: 104200
Memory footprint: 22.26 M

Preconditioner
==============
Number of levels: 3
Operator complexity: 1.18
Grid complexity: 1.11
Memory footprint: 525.68 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>104200</td>
<td>1395146</td>
<td>445.04 M (84.98%)</td>
</tr>
<tr>
<td>1</td>
<td>10365</td>
<td>235821</td>
<td>70.07 M (14.36%)</td>
</tr>
<tr>
<td>2</td>
<td>600</td>
<td>10792</td>
<td>10.57 M ( 0.66%)</td>
</tr>
</tbody>
</table>

Iterations: 4
Error: 2.90461e-09

[Profile: 1.356 s] (100.00%)
[ self: 0.063 s] ( 4.62%)

(continues on next page)
This is much better! Looks like switching to the block-valued backend not only improved the setup and solution performance, but also increased the convergence speed. This version is about 12 times faster than the first working approach. Let’s see how this translates to the code, with the added bonus of using the mixed precision solution. The source below shows the complete solution and is also available in tutorial/3.CoupCons3D/coupcons3d.cpp. The only differences (highlighted in the listing) with the solution from Structural problem are the choices of the iterative solver and the smoother, and the block size.

Listing 2.12: The source code for the solution of the CoupCons3D problem.

```cpp
#include <vector>
#include <iostream>

#include <amgcl/backend/builtin.hpp>
#include <amgcl/adapter/crs_tuple.hpp>
#include <amgcl/make_solver.hpp>
#include <amgcl/amg.hpp>
#include <amgcl/coarsening/smoothed_aggregation.hpp>
#include <amgcl/relaxation/ilu0.hpp>
#include <amgcl/solver/bicgstab.hpp>
#include <amgcl/value_type/static_matrix.hpp>
#include <amgcl/adapter/block_matrix.hpp>
#include <amgcl/io/mm.hpp>
#include <amgcl/profiler.hpp>

int main(int argc, char *argv[]) {
  // The command line should contain the matrix file name:
  if (argc < 2) {
    std::cerr << "Usage: " "<argv[0]" "<matrix.mtx>" "<< std::endl;
    return 1;
  }

  // The profiler:
  amgcl::profiler<> prof("Serena");

  // Read the system matrix:
  ptrdiff_t rows, cols;
  std::vector<ptrdiff_t> ptr, col;
  std::vector<double> val;
  prof.tic("read");
  std::tie(rows, cols) = amgcl::io::mm_reader(argv[1])(ptr, col, val);
  std::cout << "Matrix " << argv[1] << "x" << rows << "x" << cols << std::endl;
  prof.toc("read");

  // The RHS is filled with ones:
  std::vector<double> f(rows, 1.0);

  // Scale the matrix so that it has the unit diagonal.
  // First, find the diagonal values:
  std::vector<double> D(rows, 1.0);
```

(continues on next page)
```cpp
for(ptrdiff_t i = 0; i < rows; ++i) {
    for(ptrdiff_t j = ptr[i], e = ptr[i+1]; j < e; ++j) {
        if (col[j] == i) {
            D[i] = 1 / sqrt(val[j]);
            break;
        }
    }
}

// Then, apply the scaling in-place:
for(ptrdiff_t i = 0; i < rows; ++i) {
    for(ptrdiff_t j = ptr[i], e = ptr[i+1]; j < e; ++j) {
        val[j] *= D[i] * D[col[j]];
    }
    f[i] *= D[i];
}

// We use the tuple of CRS arrays to represent the system matrix.
// Note that std::tie creates a tuple of references, so no data is actually
// copied here:
auto A = std::tie(rows, ptr, col, val);

// Compose the solver type
typedef amgcl::static_matrix<double, 4, 4> dmat_type; // matrix value type in double precision
typedef amgcl::static_matrix<double, 4, 1> dvec_type; // the corresponding vector value type
typedef amgcl::static_matrix<float, 4, 4> smat_type; // matrix value type in single precision

typedef amgcl::backend::builtin<dmat_type> SBackend; // the solver backend
typedef amgcl::backend::builtin<smat_type> PBackend; // the preconditioner backend

typedef amgcl::make_solver<
    amgcl::amg<
        PBackend,
        amgcl::coarsening::smoothed_aggregation,
        amgcl::relaxation::ilu0
    >,
    amgcl::solver::bicgstab<SBBackend>
> Solver;

// Initialize the solver with the system matrix.
// Use the block_matrix adapter to convert the matrix into
// the block format on the fly:
prof.tic("setup");
auto Ab = amgcl::adapter::block_matrix<dmat_type>(A);
Solver solve(Ab);
prof.toc("setup");

// Show the mini-report on the constructed solver:
std::cout << solve << std::endl;

// Solve the system with the zero initial approximation:
int iters;
double error;
std::vector<double> x(rows, 0.0);
```

(continues on next page)
// Reinterpret both the RHS and the solution vectors as block-valued:
auto f_ptr = reinterpret_cast<dvec_type*>(f.data());
auto x_ptr = reinterpret_cast<dvec_type*>(x.data());

auto F = amgcl::make_iterator_range(f_ptr, f_ptr + rows / 4);
auto X = amgcl::make_iterator_range(x_ptr, x_ptr + rows / 4);

prof.tic("solve");
std::tie(iters, error) = solve(Ab, F, X);
prof.toc("solve");

// Output the number of iterations, the relative error,
// and the profiling data:
std::cout << "Iters: " << iters << std::endl
<< "Error: " << error << std::endl
<< prof << std::endl;

The output from the compiled program is given below. The main improvement here is the reduced memory footprint
of the single-precision preconditioner: it takes 279.83M as opposed to 525.68M in the full precision case. The setup
and the solution are slightly faster as well:

$ ./coupcons3d CoupCons3D.mtx
Matrix CoupCons3D.mtx: 416800x416800
Solver
======
Type: BiCGStab
Unknowns: 104200
Memory footprint: 22.26 M

Preconditioner
==============
Number of levels: 3
Operator complexity: 1.18
Grid complexity: 1.11
Memory footprint: 279.83 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>104200</td>
<td>1395146</td>
<td>237.27 M (84.98%)</td>
</tr>
<tr>
<td>1</td>
<td>10365</td>
<td>235821</td>
<td>37.27 M (14.36%)</td>
</tr>
<tr>
<td>2</td>
<td>600</td>
<td>10792</td>
<td>5.29 M ( 0.66%)</td>
</tr>
</tbody>
</table>

Iters: 4
Error: 2.90462e-09

[Serena: 14.415 s] (100.00%)
[  self: 0.057 s] ( 0.39%)
[  read: 13.426 s] ( 93.14%)
[  setup: 0.345 s] ( 2.39%)
[  solve: 0.588 s] ( 4.08%)

We can also use the VexCL backend to accelerate the solution using the GPU. Again, this is very close to the approach
described in Structural problem (see tutorial/3.CoupCons3D/coupcons3d_vexcl.cpp). However, the ILU(0) relaxation
is an intrinsically serial algorithm, and is not effective with the fine grained parallelism of the GPU. Instead, the
solutions of the lower and upper parts of the incomplete LU decomposition in AMGCL are approximated with several
Jacobi iterations [ChPa15]. This makes the relaxation relatively more expensive than on the CPU, and the speedup

2.4. Tutorial
from using the GPU backend is not as prominent:

```bash
$ ./coupcons3d_vexcl_cuda CoupCons3D.mtx
1. GeForce GTX 1050 Ti

Matrix CoupCons3D.mtx: 416800x416800
Solver
=====
Type: BiCGStab
Unknowns: 104200
Memory footprint: 22.26 M

Preconditioner
==============
Number of levels: 3
Operator complexity: 1.18
Grid complexity: 1.11
Memory footprint: 281.49 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>104200</td>
<td>1395146</td>
<td>238.79 M (84.98%)</td>
</tr>
<tr>
<td>1</td>
<td>10365</td>
<td>235821</td>
<td>37.40 M (14.36%)</td>
</tr>
<tr>
<td>2</td>
<td>600</td>
<td>10792</td>
<td>5.31 M (0.66%)</td>
</tr>
</tbody>
</table>

Iters: 5
Error: 6.30647e-09

[Serena: 14.432 s] (100.00%)
[ self: 0.060 s] ( 0.41%)
[ GPU matrix: 0.213 s] ( 1.47%)
[ read: 13.381 s] (92.72%)
[ setup: 0.549 s] ( 3.81%)
[ solve: 0.229 s] ( 1.59%)
```

**Note:** We used the fact that the matrix size is divisible by 4 in order to use the block-valued backend. If it was not the case, we could use the Schur pressure correction preconditioner to split the matrix into two large subsystems, and use the block-valued solver for the upper left subsystem. See an example of such a solution in tutorial/3.CoupCons3D/coupcons3d_spc.cpp. The performance is worse than what we were able to achieve above, but still is better than the first working version:

```bash
$ ./coupcons3d_spc CoupCons3D.mtx 333440
Matrix CoupCons3D.mtx: 416800x416800
Solver
=====
Type: BiCGStab
Unknowns: 416800
Memory footprint: 22.26 M

Preconditioner
==============
Schur complement (two-stage preconditioner)
  Unknowns: 416800(83360)
  Nonzeros: 22322336
  Memory: 549.90 M

[ U ]
```

(continues on next page)
2.4.6 Stokes-like problem

In this section we consider a saddle point system which was obtained by discretization of the steady incompressible Stokes flow equations in a unit cube with a locally divergence-free weak Galerkin finite element method. The UCube(4) system studied here may be downloaded from the dataset accompanying the paper [DeMW20]. We will use the UCube(4) system from the dataset. The system matrix is symmetric and has 554,496 rows and 14,292,884 nonzero values, which corresponds to an average of 26 nonzero entries per row. The matrix sparsity portrait is shown on the figure below.
Fig. 2.6: UCube(4) matrix portrait
As with any Stokes-like problem, the system has a general block-wise structure:

\[
\begin{bmatrix}
A & B_2^T \\
B_2 & C
\end{bmatrix}
\begin{bmatrix}
x_u \\
x_p
\end{bmatrix}
=
\begin{bmatrix}
b_u \\
b_p
\end{bmatrix}
\]

In this case, the upper left subblock \(A\) corresponds to the flow unknowns, and itself has block-wise structure with small \(3 \times 3\) blocks. The lower right subblock \(C\) corresponds to the pressure unknowns. There is a lot of research dedicated to the efficient solution of such systems, see [BeGL05] for an extensive overview. The direct approach of using a monolithic preconditioner usually does not work very well, but we may try it to have a reference point. The AMG preconditioning does not yield a converging solution, but a single level ILU(0) relaxation seems to work with a CG iterative solver:

```
$ solver -B -A ucube_4_A.bin -f ucube_4_b.bin \
    solver.type=cg solver.maxiter=500 \
    precond.class=relaxation precond.type=ilu0
```

Solver
======
Type: CG
Unknowns: 554496
Memory footprint: 16.92 M

Preconditioner
==============
Relaxation as preconditioner
Unknowns: 554496
Nonzeros: 14292884
Memory: 453.23 M

Iterations: 270
Error: 6.84763e-09

[Profile: 9.300 s] (100.00%)
[ reading: 0.133 s] ( 1.43%)
[ setup: 0.561 s] ( 6.03%)
[ solve: 8.599 s] ( 92.46%)

A preconditioner that takes the structure of the system into account should be a better choice performance-wise. AMGCL provides an implementation of the Schur complement pressure correction preconditioner. The preconditioning step consists of solving two linear systems:

\[
Sx_p = b_p - B_2 A^{-1} b_u,
\]

\[
A x_u = b_u - B_2^T b_p.
\]

Here \(S\) is the Schur complement \(S = C - B_2 A^{-1} B_2^T\). Note that forming the Schur complement matrix explicitly is prohibitively expensive, and the following approximation is used to create the preconditioner for the first equation in (2.2):

\[
\hat{S} = C - \text{diag} (B_2 \text{diag}(A)^{-1} B_2^T).
\]

There is no need to solve the equations (2.2) exactly. It is enough to perform a single application of the corresponding preconditioner as an approximation to \(S^{-1}\) and \(A^{-1}\). This means that the overall preconditioner is linear, and we may use a non-flexible iterative solver with it. The approximation matrix \(\hat{S}\) has a simple band diagonal structure, and a diagonal SPAI(0) preconditioner should have reasonable performance.

Similar to the examples/solver, the examples/schur_pressure_correction utility allows to play with the Schur pressure correction preconditioner options before trying to write any code. We found that using the non-smoothed aggregation with ILU(0) smoothing on each level for the flow subsystem (usolver) and single-level SPAI(0) relaxation for the Schur complement subsystem (psolver) works best. We also disable lumping of the diagonal of the \(A\) matrix in...
the Schur complement approximation with the `precond.simplec_dia=false` option, and enable block-valued backend for the flow subsystem with the `--ub 3` option. The `-m '>'456192'` option sets the pressure mask pattern. It tells the solver that all unknowns starting with the 456192-th belong to the pressure subsystem:

```bash
$ schur_pressure_correction -B -A ucube_4_A.bin -f ucube_4_b.bin -m '>'456192' \
   -p solver.type=cg solver.maxiter=200 \precond.simplec_dia=false \precond.usolver.solver.type=preonly \precond.usolver.precond.coarsening.type=aggregation \precond.usolver.precond.relax.type=ilu0 \precond.psolver.solver.type=preonly \precond.psolver.precond.class=relaxation \--ub 3
```

**Solver**

```
Solver
=====
Type: CG
Unknowns: 554496
Memory footprint: 16.92 M
```

**Preconditioner**

```
Preconditioner
==============
Schur complement (two-stage preconditioner)
  Unknowns: 554496 (98304)
  Nonzeros: 14292884
  Memory: 587.45 M

[ U ]
Solver
=====
Type: PreOnly
Unknowns: 152064
Memory footprint: 0.00 B
```

**Preconditioner**

```
Preconditioner
==============
Number of levels: 4
Operator complexity: 1.25
Grid complexity: 1.14
Memory footprint: 233.07 M
```

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>152064</td>
<td>982416</td>
<td>188.13 M (80.25%)</td>
</tr>
<tr>
<td>1</td>
<td>18654</td>
<td>197826</td>
<td>35.07 M (16.16%)</td>
</tr>
<tr>
<td>2</td>
<td>2619</td>
<td>35991</td>
<td>6.18 M (2.94%)</td>
</tr>
<tr>
<td>3</td>
<td>591</td>
<td>7953</td>
<td>3.69 M (0.65%)</td>
</tr>
</tbody>
</table>

[ P ]

**Solver**

```
Solver
=====
Type: PreOnly
Unknowns: 98304
Memory footprint: 0.00 B
```

**Preconditioner**

```
Preconditioner
==============
Relaxation as preconditioner
```

(continues on next page)
Unknowns: 98304
Nonzeros: 274472
Memory: 5.69 M

Iterations: 35
Error: 8.57921e-09

[Profile: 3.872 s] (100.00%)  
[ reading: 0.131 s] ( 3.38%)  
[ schur_complement: 3.741 s] ( 96.62%)  
[ self: 0.031 s] ( 0.79%)  
[ setup: 0.301 s] ( 7.78%)  
[ solve: 3.409 s] ( 88.05%)

Let's see how this translates to the code. Below is the complete listing of the solver (tutorial/4.Stokes/stokes_ucube.cpp) which uses the mixed precision approach.

Listing 2.13: The source code for the solution of the UCube(4) problem.

```cpp
#include <iostream>
#include <string>
#include <amgcl/backend/builtin.hpp>
#include <amgcl/adapter/crs_tuple.hpp>
#include <amgcl/value_type/static_matrix.hpp>
#include <amgcl/adapter/block_matrix.hpp>
#include <amgcl/preconditioner/schur_pressure_correction.hpp>
#include <amgcl/make_solver.hpp>
#include <amgcl/make_block_solver.hpp>
#include <amgcl/amg.hpp>
#include <amgcl/solver/cg.hpp>
#include <amgcl/solver/preonly.hpp>
#include <amgcl/coarsening/aggregation.hpp>
#include <amgcl/relaxation/ilu0.hpp>
#include <amgcl/relaxation/spai0.hpp>
#include <amgcl/relaxation/as_preconditioner.hpp>
#include <amgcl/io/binary.hpp>
#include <amgcl/profiler.hpp>

//------------------------------

int main(int argc, char *argv[]) {
    // The command line should contain the matrix and the RHS file names, 
    // and the number of unknowns in the flow subsystem:
    if (argc < 4) {
        std::cerr << "Usage: " << argv[0] << " matrix.bin rhs.bin <nu>" << std::endl;
        return 1;
    }

    // The profiler:
    amgcl::profiler<> prof("UCube4");

    // Read the system matrix:
    ptrdiff_t rows, cols;
```

(continues on next page)
std::vector<ptrdiff_t> ptr, col;
std::vector<double> val, rhs;

prof.tic("read");
amgcl::io::read_crs(argv[1], rows, ptr, col, val);
amgcl::io::read_dense(argv[2], rows, cols, rhs);
std::cout << "Matrix " << argv[1] << ": " << rows << "x" << rows << std::endl;
std::cout << "RHS " << argv[2] << ": " << rows << "x" << cols << std::endl;
prof.toc("read");

// The number of unknowns in the U subsystem
ptrdiff_t nu = std::stoi(argv[3]);

// We use the tuple of CRS arrays to represent the system matrix.
// Note that std::tie creates a tuple of references, so no data is actually
// copied here:
auto A = std::tie(rows, ptr, col, val);

// Compose the solver type
typedef amgcl::backend::builtin<double> SBackend; // the outer iterative solver
typedef amgcl::backend::builtin<float> PBackend; // the PSolver backend
typedef amgcl::static_matrix<float, 3, 3> UBackend; // the USolver backend

typedef amgcl::make_solver<
    amgcl::preconditioner::schur_pressure_correction<
        amgcl::make_block_solver<
            amgcl::amg<
                UBackend,
                amgcl::coarsening::aggregation,
                amgcl::relaxation::ilu0
            >,
            amgcl::solver::preonly<UBackend>
        >,
        amgcl::make_solver<
            amgcl::relaxation::as_preconditioner<
                PBackend,
                amgcl::relaxation::spai0
            >,
            amgcl::solver::preonly<PBackend>
        >
    >,
    amgcl::solver::cg<SBackend>
> Solver;

// Solver parameters
Solver::params prm;
prm.precond.simplec_dia = false;
prm.precond.pmask.resize(rows);
for(ptrdiff_t i = 0; i < rows; ++i) prm.precond.pmask[i] = (i >= nu);

// Initialize the solver with the system matrix.
prof.tic("setup");
Solver solve(A, prm);
prof.toc("setup");

(continues on next page)
Schur pressure correction is composite preconditioner. Its definition includes definition of two nested iterative solvers, one for the “flow” (U) subsystem, and the other for the “pressure” (P) subsystem. In lines 55–58 we define the backends used in the outer iterative solver, and in the two nested solvers. Note that both backends for nested solvers use single precision values, and the flow subsystem backend has block value type:

```cpp
typedef amgcl::backend::builtin<double> SBackend; // the outer iterative solver backend

typedef amgcl::backend::builtin<float> PBackend; // the PSolver backend

typedef amgcl::backend::builtin<
    amgcl::static_matrix<float, 3, 3>> UBackend; // the USolver backend
```

In lines 60-79 we define the solver type. The flow solver is defined in lines 62-69, and the pressure solver – in lines 70–77. Both are using `amgcl::solver::preonly` as “iterative” solver, which in fact only applies the specified preconditioner once. The flow solver is defined with `amgcl::make_block_solver`, which automatically converts its input matrix $A$ to the block format during the setup and reinterprets the scalar RHS and solution vectors as having block values during solution:

```cpp
typedef amgcl::make_solver<
    amgcl::preconditioner::schur_pressure_correction<
        amgcl::make_block_solver<
            amgcl::amg<
                UBackend,
                amgcl::coarsening::aggregation,
                amgcl::relaxation::ilu0
                >,
                amgcl::solver::preonly<UBackend>
                >,
                amgcl::make_solver<
                    amgcl::relaxation::as_preconditioner<
                        PBackend,
                        amgcl::relaxation::spai0
                        >,
                        amgcl::solver::preonly<PBackend>
                        >
                        >,
                    amgcl::solver::cg<SBackend>
                    > Solver;
```

In the solver parameters we disable lumping of the matrix $A$ diagonal for the Schur complement approximation (line
83) and fill the pressure mask to indicate which unknowns correspond to the pressure subsystem (lines 84–85):

```cpp
// Solver parameters
Solver::params prm;
prm.precond.simplec_dia = false;
prm.precond.pmask.resize(rows);
for (ptrdiff_t i = 0; i < rows; ++i) prm.precond.pmask[i] = (i >= nu);
```

Here is the output from the compiled program. The preconditioner uses 398M or memory, as opposed to 587M in the case of the full precision preconditioner used in the examples/schur_pressure_correction, and both the setup and the solution are about 50% faster due to the use of the mixed precision approach:

```
$ ./stokes_ucube ucube_4_A.bin ucube_4_b.bin 456192
Matrix ucube_4_A.bin: 554496x554496
RHS ucube_4_b.bin: 554496x1
Solver
======
Type:     CG
Unknowns: 554496
Memory footprint: 16.92 M

Preconditioner
==============
Schur complement (two-stage preconditioner)
  Unknowns: 554496(98304)
  Nonzeros: 14292884
  Memory:   398.39 M

[ U ]
Solver
======
Type:     PreOnly
Unknowns: 152064
Memory footprint: 0.00 B

Preconditioner
==============
Number of levels: 4
Operator complexity: 1.25
Grid complexity:   1.14
Memory footprint: 130.49 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>152064</td>
<td>982416</td>
<td>105.64 M (80.25%)</td>
</tr>
<tr>
<td>1</td>
<td>18654</td>
<td>197826</td>
<td>19.56 M (16.16%)</td>
</tr>
<tr>
<td>2</td>
<td>2619</td>
<td>35991</td>
<td>3.44 M (2.94%)</td>
</tr>
<tr>
<td>3</td>
<td>591</td>
<td>7953</td>
<td>1.85 M (0.65%)</td>
</tr>
</tbody>
</table>

[ P ]
Solver
======
Type:     PreOnly
Unknowns: 98304
Memory footprint: 0.00 B

Preconditioner
```

(continues on next page)
Relaxation as preconditioner
Unknowns: 98304
Nonzeros: 274472
Memory: 4.27 M

Iters: 35
Error: 8.57996e-09

[UCube4: 2.502 s] (100.00%)
[ read: 0.129 s] ( 5.16%)
[ setup: 0.240 s] ( 9.57%)
[ solve: 2.132 s] (85.19%)

Converting the solver to the VexCL backend in order to accelerate the solution with GPGPU is straightforward. Below is the complete source code of the solver (tutorial/4.Stokes/stokes_ucube_vexcl.cpp), with the differences between the OpenMP and the VexCL versions highlighted. Note that the GPU version of the ILU(0) smoother approximates the lower and upper triangular solves in the incomplete LU decomposition with a couple of Jacobi iterations [ChPa15]. Here we set the number of iterations to 4 (line 94).

Listing 2.14: The source code for the solution of the UCube(4) problem with the VexCL backend.

```cpp
#include <iostream>
#include <string>

#include <amgcl/backend/vexcl.hpp>
#include <amgcl/backend/vexcl_static_matrix.hpp>
#include <amgcl/adapter/crs_tuple.hpp>
#include <amgcl/value_type/static_matrix.hpp>
#include <amgcl/adapter/block_matrix.hpp>
#include <amgcl/preconditioner/schur_pressure_correction.hpp>
#include <amgcl/make_solver.hpp>
#include <amgcl/make_block_solver.hpp>
#include <amgcl/amg.hpp>
#include <amgcl/solver/cg.hpp>
#include <amgcl/solver/preonly.hpp>
#include <amgcl/coarsening/aggregation.hpp>
#include <amgcl/relaxation/illu0.hpp>
#include <amgcl/relaxation/spai0.hpp>
#include <amgcl/relaxation/as_preconditioner.hpp>

#include <amgcl/io/binary.hpp>
#include <amgcl/profiler.hpp>

int main(int argc, char *argv[]) {
    // The command line should contain the matrix and the RHS file names,
    // and the number of unknowns in the flow subsystem:
    if (argc < 4) {
        std::cerr << "Usage: " << argv[0] << " <matrix.bin> <rhs.bin> <nu>" << std::endl;
        return 1;
    }
...
// Create VexCL context. Set the environment variable OCL_DEVICE to
// control which GPU to use in case multiple are available,
// and use single device:
vox::Context ctx(vox::Filter::Env && vox::Filter::Count(1));
std::cout << ctx << std::endl;

// Enable support for block-valued matrices in the VexCL kernels:
vox::scoped_program_header header(ctx, amgcl::backend::vexcl_static_matrix_
declaration<float,3>>());

// The profiler:
amgcl::profiler<> prof("UCube4 (VexCL"));

// Read the system matrix:
ptrdiff_t rows, cols;
std::vector<ptrdiff_t> ptr, col;
std::vector<double> val, rhs;

prof.tic("read");
amgcl::io::read_crs(argv[1], rows, ptr, col, val);
amgcl::io::read_dense(argv[2], rows, cols, rhs);
std::cout << "Matrix " << argv[1] << ": " << rows << "x" << rows << std::endl;
std::cout << "RHS " << argv[2] << ": " << rows << "x" << cols << std::endl;
prof.toc("read");

// The number of unknowns in the U subsystem
ptrdiff_t nu = std::stoi(argv[3]);

// We use the tuple of CRS arrays to represent the system matrix.
// Note that std::tie creates a tuple of references, so no data is actually
// copied here:
auto A = std::tie(rows, ptr, col, val);

// Compose the solver type
typedef amgcl::backend::vexcl<double> SBackend; // the outer iterative solver
typedef amgcl::backend::vexcl<float> PBackend; // the PSolver backend
typedef amgcl::backend::vexcl<
    amgcl::static_matrix<float,3,3>> UBackend; // the USolver backend

typedef amgcl::make_solver<
    amgcl::preconditioner::schur_pressure_correction<
        amgcl::make_block_solver<
            amgcl::amg<
                UBackend,
                amgcl::coarsening::aggregation,
                amgcl::relaxation::ilu0
            >,
            amgcl::solver::preonly<UBackend>
        >,
        amgcl::make_solver<
            amgcl::relaxation::as_preconditioner<
                PBackend,
                amgcl::relaxation::spai0
            >,
            amgcl::solver::preonly<PBackend>
        >
    >
>
Solver parameters
Solver::params prm;
prm.precond.simplec_dia = false;
prm.precond.usolver.precond.relax.solve.iters = 4;
prm.precond.pmask.resize(rows);
for(ptrdiff_t i = 0; i < rows; ++i) prm.precond.pmask[i] = (i >= nu);

// Set the VexCL context in the backend parameters
SBackend::params bprm;
bprm.q = ctx;

// Initialize the solver with the system matrix.
prof.tic("setup");
Solver solve(A, prm, bprm);
prof.toc("setup");

// Show the mini-report on the constructed solver:
std::cout << solve << std::endl;

// Since we are using mixed precision, we have to transfer the system matrix to the GPU:
prof.tic("GPU matrix");
auto A_gpu = SBackend::copy_matrix(std::make_shared<amgcl::backend::crs<double>>(A), bprm);
prof.toc("GPU matrix");

// Solve the system with the zero initial approximation:
int iters;
double error;
vex::vector<double> f(ctx, rhs);
vex::vector<double> x(ctx, rows);
x = 0.0;

prof.tic("solve");
std::tie(iters, error) = solve(*A_gpu, f, x);
prof.toc("solve");

// Output the number of iterations, the relative error, and the profiling data:
std::cout << "Iters: " << iters << std::endl << "Error: " << error << std::endl
<< prof << std::endl;
}

The output of the VexCL version is shown below. The solution phase is about twice as fast as the OpenMP version:

```
$ ./stokes_ucube_vexcl_cuda ucube_4_A.bin ucube_4_b.bin 456192
1. GeForce GTX 1050 Ti
Matrix ucube_4_A.bin: 554496x554496
RHS ucube_4_b.bin: 554496x1
Solver

(continues on next page)
Type: CG
Unknowns: 554496
Memory footprint: 16.92 M

Preconditioner
==============
Schur complement (two-stage preconditioner)
Unknowns: 554496 (98304)
Nonzeros: 14292884
Memory: 399.66 M

[ U ]
Solver
=====
Type: PreOnly
Unknowns: 152064
Memory footprint: 0.00 B

Preconditioner
==============
Number of levels: 4
Operator complexity: 1.25
Grid complexity: 1.14
Memory footprint: 131.76 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>152064</td>
<td>982416</td>
<td>106.76 M (80.25%)</td>
</tr>
<tr>
<td>1</td>
<td>18654</td>
<td>197826</td>
<td>19.68 M (16.16%)</td>
</tr>
<tr>
<td>2</td>
<td>2619</td>
<td>35991</td>
<td>3.45 M (2.94%)</td>
</tr>
<tr>
<td>3</td>
<td>591</td>
<td>7953</td>
<td>1.86 M (0.65%)</td>
</tr>
</tbody>
</table>

[ P ]
Solver
=====
Type: PreOnly
Unknowns: 98304
Memory footprint: 0.00 B

Preconditioner
==============
Relaxation as preconditioner
Unknowns: 98304
Nonzeros: 274472
Memory: 4.27 M

Iters: 36
Error: 7.26253e-09

[UCube4 (VexCL): 1.858 s] (100.00%)
[  self: 0.004 s] ( 0.20%)
[ GPU matrix: 0.213 s] (11.46%)
[  read: 0.128 s] ( 6.87%)
[  setup: 0.519 s] (27.96%)
[  solve: 0.994 s] (53.52%)
2.4.7 Using near null-space vectors

Using near null-space vectors may greatly improve the quality of the aggregation AMG preconditioner. For the elasticity or structural problems the near-null space vectors may be computed as rigid body modes from the coordinates of the discretization grid nodes. In this tutorial we will use the system obtained by discretization of a 3D elasticity problem modeling a connecting rod:

![The connecting rod geometry with the computed displacements](image)

Fig. 2.7: The connecting rod geometry with the computed displacements

The dataset was kindly provided by David Herrero Pérez (@davidherreroperez) in the issue #135 on Github and is available for download at doi:10.5281/zenodo.4299865. The system matrix is symmetric, has block structure with small $3 \times 3$ blocks, and has 81,657 rows and 3,171,111 nonzero values (about 39 nonzero entries per row on average). The matrix portrait is shown on the figure below:

It is possible to solve the system using the CG iterative solver preconditioned with the smoothed aggregation AMG, but the convergence is not that great:

```
$ solver -A A.mtx -f b.mtx solver.type=cg solver.maxiter=1000
Solver
=====
Type: CG
Unknowns: 81657
Memory footprint: 2.49 M
```

(continues on next page)
Fig. 2.8: The nonzero portrait of the connecting rod system.
Preconditioner
==============

Number of levels: 3
Operator complexity: 1.14
Grid complexity: 1.07
Memory footprint: 70.09 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>81657</td>
<td>3171111</td>
<td>62.49 M (87.98%)</td>
</tr>
<tr>
<td>1</td>
<td>5067</td>
<td>417837</td>
<td>7.16 M (11.59%)</td>
</tr>
<tr>
<td>2</td>
<td>305</td>
<td>15291</td>
<td>450.07 K (0.42%)</td>
</tr>
</tbody>
</table>

Iterations: 698
Error: 8.96391e-09

Profile: 11.717 s (100.00%)
Reading: 2.123 s (18.12%)
Setup: 0.122 s (1.04%)
Solve: 9.472 s (80.84%)

We can improve the solution time by taking the block structure of the system into account in the aggregation algorithm:

```bash
$ solver -A A.mtx -f b.mtx solver.type=cg solver.maxiter=1000 \
    precond.coarsening.aggr.block_size=3
```

Solver
======
Type: CG
Unknowns: 81657
Memory footprint: 2.49 M

Preconditioner
==============

Number of levels: 3
Operator complexity: 1.29
Grid complexity: 1.10
Memory footprint: 92.40 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>81657</td>
<td>3171111</td>
<td>75.83 M (77.71%)</td>
</tr>
<tr>
<td>1</td>
<td>7773</td>
<td>858051</td>
<td>15.70 M (21.03%)</td>
</tr>
<tr>
<td>2</td>
<td>555</td>
<td>51327</td>
<td>890.16 K (1.26%)</td>
</tr>
</tbody>
</table>

Iterations: 197
Error: 8.76043e-09

Profile: 5.525 s (100.00%)
Reading: 2.170 s (39.28%)
Setup: 0.173 s (3.14%)
Solve: 3.180 s (57.56%)

However, since this is an elasticity problem and we know the coordinates for the discretization mesh, we can compute the rigid body modes and provide them as the near null-space vectors for the smoothed aggregation AMG method. AMGCL has a convenience function `amgcl::coarsening::rigid_body_modes()` that takes the 2D or 3D coordinates and converts them into the rigid body modes. The examples/solver utility allows to specify the file con-
In the 3D case we get 6 near null-space vectors corresponding to the rigid body modes. Note that this makes the preconditioner more expensive memory-wise: the memory footprint of the preconditioner has increased to 132M from 70M in the simplest case and 92M in the case using the block structure of the matrix. But this pays up in terms of performance: the number of iterations dropped from 197 to 63 and the solution time decreased from 3.2 seconds to 1.2 seconds.

In principle, it is also possible to approximate the near null-space vectors by solving the homogeneous system $Ax = 0$, starting with a random initial solution $x$. We may use the computed $x$ as a near-null space vector, solve the homogeneous system again from a different random start, and do this until we have enough near null-space vectors. The examples/ns_search.cpp example shows how to do this. However, this process is quite expensive, because we need to solve the system multiple times, starting with a badly tuned solver at that. It is probably only worth the time in case one needs to solve the same system efficiently for multiple right-hand side vectors. Below is an example of searching for the 6 near null-space vectors:

```
$ ns_search -A A.mtx -f b.mtx solver.type=cg solver.maxiter=1000 \
   precond.coarsening.aggr.eps_strong=0 -n6 -o N6.mtx
```

(continues on next page)
Preconditioner
==============
Number of levels: 2
Operator complexity: 1.01
Grid complexity: 1.02
Memory footprint: 62.79 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>81657</td>
<td>3171111</td>
<td>60.56 M (98.58%)</td>
</tr>
<tr>
<td>1</td>
<td>1284</td>
<td>45576</td>
<td>2.24 M (1.42%)</td>
</tr>
</tbody>
</table>

Iterations: 932
Error: 8.66233e-09

--- Searching for vector 1 ---
Solver
=====
Type: CG
Unknowns: 81657
Memory footprint: 2.49 M

Preconditioner
==============
Number of levels: 2
Operator complexity: 1.01
Grid complexity: 1.02
Memory footprint: 62.79 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>81657</td>
<td>3171111</td>
<td>60.56 M (98.58%)</td>
</tr>
<tr>
<td>1</td>
<td>1284</td>
<td>45576</td>
<td>2.24 M (1.42%)</td>
</tr>
</tbody>
</table>

Iterations: 750
Error: 9.83476e-09

--- Searching for vector 2 ---
Solver
=====
Type: CG
Unknowns: 81657
Memory footprint: 2.49 M

Preconditioner
==============
Number of levels: 2
Operator complexity: 1.06
Grid complexity: 1.03
Memory footprint: 76.72 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>81657</td>
<td>3171111</td>
<td>60.56 M (98.58%)</td>
</tr>
<tr>
<td>1</td>
<td>1284</td>
<td>45576</td>
<td>2.24 M (1.42%)</td>
</tr>
</tbody>
</table>
Iterations: 528
Error: 8.74633e-09

-- Searching for vector 3

Solver
======
Type: CG
Unknowns: 81657
Memory footprint: 2.49 M

Preconditioner
--------------
Number of levels: 3
Operator complexity: 1.13
Grid complexity: 1.05
Memory footprint: 84.87 M

level  unknowns  nonzeros  memory
---------------------------------------------
0  81657     3171111  77.41 M (88.49%)
1  3852      410184   7.42 M (11.45%)
2   72       2394     31.36 K (0.07%)

Iterations: 391
Error: 9.04425e-09

-- Searching for vector 4

Solver
======
Type: CG
Unknowns: 81657
Memory footprint: 2.49 M

Preconditioner
--------------
Number of levels: 3
Operator complexity: 1.23
Grid complexity: 1.06
Memory footprint: 99.01 M

level  unknowns  nonzeros  memory
---------------------------------------------
0  81657     3171111  85.84 M (81.22%)
1  5136      729216   13.11 M (18.68%)
2   96       4256     55.00 K (0.11%)

Iterations: 238
Error: 9.51092e-09
-- Searching for vector 5
-------------------------
Solver
======
Type: CG
Unknowns: 81657
Memory footprint: 2.49 M

Preconditioner
==============
Number of levels: 3
Operator complexity: 1.36
Grid complexity: 1.08
Memory footprint: 114.78 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>81657</td>
<td>3171111</td>
<td>94.27 M (73.45%)</td>
</tr>
<tr>
<td>1</td>
<td>6420</td>
<td>1139400</td>
<td>20.42 M (26.39%)</td>
</tr>
<tr>
<td>2</td>
<td>120</td>
<td>6650</td>
<td>85.24 K (0.15%)</td>
</tr>
</tbody>
</table>

Iterations: 175
Error: 9.43207e-09

-------------------------
-- Solving the system
-------------------------
Solver
======
Type: CG
Unknowns: 81657
Memory footprint: 2.49 M

Preconditioner
==============
Number of levels: 3
Operator complexity: 1.52
Grid complexity: 1.10
Memory footprint: 132.15 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>81657</td>
<td>3171111</td>
<td>102.70 M (65.77%)</td>
</tr>
<tr>
<td>1</td>
<td>7704</td>
<td>1640736</td>
<td>29.33 M (34.03%)</td>
</tr>
<tr>
<td>2</td>
<td>144</td>
<td>9576</td>
<td>122.07 K (0.20%)</td>
</tr>
</tbody>
</table>

Iterations: 100
Error: 8.14427e-09

[Profile: 48.503 s] (100.00%)
[ apply: 2.373 s] ( 4.89%)
[ setup: 0.422 s] ( 0.87%)
[ solve: 1.949 s] ( 4.02%)
[ read: 2.113 s] ( 4.36%)
[ search: 43.713 s] ( 90.12%)
[ vector 0: 12.437 s] (25.64%)
[ setup: 0.101 s] ( 0.21%)

(continues on next page)
Note that the number of iterations required to find the next vector is gradually decreasing, as the quality of the solver increases. The 6 orthogonalized vectors are saved to the output file N6.mtx and are also used to solve the original system. We can also use the file with the examples/solver:

```
$ solver -A A.mtx -f b.mtx solver.type=cg precond.coarsening.aggr.eps_strong=0 -N N6.mtx
```

Solver
======
Type: CG
Unknowns: 81,657
Memory footprint: 2.49 M

Preconditioner
==============
Number of levels: 3
Operator complexity: 1.52
Grid complexity: 1.10
Memory footprint: 132.15 M

```
<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>81,657</td>
<td>3,171,111</td>
<td>102.70 M</td>
</tr>
<tr>
<td>1</td>
<td>7,704</td>
<td>164,073</td>
<td>29.33 M</td>
</tr>
<tr>
<td>2</td>
<td>144</td>
<td>9,576</td>
<td>122.07 K</td>
</tr>
</tbody>
</table>
```

Iterations: 100
Error: 8.14427e-09

```
[Profile: 4.736 s] (100.00%)
[ reading: 2.407 s] (50.83%)
[ setup: 0.354 s] (7.47%)
[ solve: 1.974 s] (41.69%)
```

This is an improvement with respect to the version that only uses the blockwize structure of the matrix, but is about 50% less effective than the version using the grid coordinates in order to compute the rigid body modes.

The listing below shows the complete source code computing the near null-space vectors from the mesh coordinates and using the vectors in order to improve the quality of the preconditioner. We include the `<amgcl/coarsening/rigid_body_modes.hpp>` header to bring the definition of the
amgcl::coarsening::rigid_body_modes() function in line 9, and use the function to convert the 3D coordinates into the 6 near null-space vectors (rigid body modes) in lines 65–66. In lines 37–38 we check that the coordinate file has the correct dimensions (since each grid node has three displacement components associated with the node, the coordinate file should have three times less rows than the system matrix). The rest of the code should be quite familiar.

Listing 2.15: The solution of the connecting rod problem using the near null-space vectors.

```cpp
#include <vector>
#include <iostream>

#include <amgcl/backend/builtin.hpp>
#include <amgcl/adapter/crs_tuple.hpp>
#include <amgcl/make_solver.hpp>
#include <amgcl/amg.hpp>
#include <amgcl/coarsening/smoothed_aggregation.hpp>
#include <amgcl/coarsening/rigid_body_modes.hpp>
#include <amgcl/relaxation/spai0.hpp>
#include <amgcl/solver/cg.hpp>

#include <amgcl/io/mm.hpp>
#include <amgcl/profiler.hpp>

int main(int argc, char *argv[]) {
    // The command line should contain the matrix, the RHS, and the coordinate files:
    if (argc < 4) {
        std::cerr << "Usage: " << argv[0] << " <A.mtx> <b.mtx> <coo.mtx>" << std::endl;
        return 1;
    }

    // The profiler:
    amgcl::profiler<> prof("Nullspace");

    // Read the system matrix, the RHS, and the coordinates:
    ptrdiff_t rows, cols, ndim, ncoo;
    std::vector<ptrdiff_t> ptr, col;
    std::vector<double> val, rhs, coo;

    prof.tic("read");
    std::tie(rows, rows) = amgcl::io::mm_reader(argv[1])(ptr, col, val);
    std::tie(rows, cols) = amgcl::io::mm_reader(argv[2])(rhs);
    std::tie(ncoo, ndim) = amgcl::io::mm_reader(argv[3])(coo);
    prof.toc("read");

    amgcl::precondition(ncoo * ndim == rows && (ndim == 2 || ndim == 3),
        "The coordinate file has wrong dimensions");

    std::cout << "Matrix " << argv[1] << ": " " <rows << "x" << rows " std::endl;
    std::cout << "RHS " << argv[2] << ": " " <rows << "x" << cols " std::endl;
    std::cout << "Coords " << argv[3] << ": " " <ncoo << "x" << ndim " std::endl;

    // Declare the solver type
    typedef amgcl::backend::builtin<double> SBackend; // the solver backend
    typedef amgcl::backend::builtin<float> PBackend; // the preconditioner backend

    typedef amgcl::make_solver<
```

(continues on next page)
amgcl::amg<
    PBackend,
    amgcl::coarsening::smoothed_aggregation,
    amgcl::relaxation::spai0
>
Solver;

// Solver parameters:
Solver::params prm;
prm.precond.coarsening.aggr.eps_strong = 0;

// Convert the coordinates to the rigid body modes.
// The function returns the number of near null-space vectors
// (3 in 2D case, 6 in 3D case) and writes the vectors to the
// std::vector<double> specified as the last argument:
prm.precond.coarsening.nullspace.cols = amgcl::coarsening::rigid_body_modes(
    ndim, coo, prm.precond.coarsening.nullspace.B);

// We use the tuple of CRS arrays to represent the system matrix.
auto A = std::tie(rows, ptr, col, val);

// Initialize the solver with the system matrix.
prof.tic("setup");
Solver solve(A, prm);
prof.toc("setup");

// Show the mini-report on the constructed solver:
std::cout << solve << std::endl;

// Solve the system with the zero initial approximation:
int iters;
double error;
std::vector<double> x(rows, 0.0);

prof.tic("solve");
std::tie(iters, error) = solve(A, rhs, x);
prof.toc("solve");

// Output the number of iterations, the relative error,
// and the profiling data:
std::cout << "Iters: " << iters << std::endl
    << "Error: " << error << std::endl
    << prof << std::endl;
}

The output of the compiled program is shown below:

$ ./nullspace A.mtx b.mtx C.mtx
Matrix A.mtx: 81657x81657
RHS b.mtx: 81657x1
Coords C.mtx: 27219x3
Solver
======
Type:        CG
Unknowns:    81657
Memory footprint: 2.49 M
Preconditioner
===============
Number of levels: 3
Operator complexity: 1.52
Grid complexity: 1.10
Memory footprint: 98.76 M

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>81657</td>
<td>3171111</td>
<td>76.73 M (65.77%)</td>
</tr>
<tr>
<td>1</td>
<td>7704</td>
<td>1640736</td>
<td>21.97 M (34.03%)</td>
</tr>
<tr>
<td>2</td>
<td>144</td>
<td>9576</td>
<td>61.60 K (0.20%)</td>
</tr>
</tbody>
</table>

Iters: 63
Error: 8.46024e-09

[Nullspace: 3.653 s] (100.00%)
[ read: 2.173 s] ( 59.48%)
[ setup: 0.326 s] ( 8.94%)
[ solve: 1.150 s] ( 31.48%)

### 2.4.8 Using near null-space vectors (MPI version)

Let us look at how to use the near null-space vectors in the MPI version of the solver for the elasticity problem (see Using near null-space vectors). The following points need to be kept in mind:

- The near null-space vectors need to be partitioned (and reordered) similar to the RHS vector.
- Since we are using coordinates of the discretization grid nodes for the computation of the rigid body modes, in order to be able to do this locally we need to partition the system in such a way that DOFs from a single grid node are owned by the same MPI process. In this case this means we need to do a block-wise partitioning with a $3 \times 3$ blocks.
- It is more convenient to partition the coordinate matrix and then to compute the rigid body modes.

The listing below shows the complete source code for the MPI elasticity solver (tutorial/5.Nullspace/nullspace_mpi.cpp)

```
#include <vector>
#include <iostream>

#include <amgcl/backend/builtin.hpp>
#include <amgcl/adapter/crs_tuple.hpp>
#include <amgcl/coarsening/rigid_body_modes.hpp>

#include <amgcl/mpi/distributed_matrix.hpp>
#include <amgcl/mpi/make_solver.hpp>
#include <amgcl/mpi/amg.hpp>
#include <amgcl/mpi/coarsening/smoothed_aggregation.hpp>
#include <amgcl/mpi/relaxation/spai0.hpp>
#include <amgcl/mpi/solver/cg.hpp>

#include <amgcl/io/binary.hpp>
```

Listing 2.16: The MPI solution of the elasticity problem
```cpp
#include <amgcl/profiler.hpp>

#if defined(AMGCL_HAVE_PARMETIS)
#include <amgcl/mpi/partition/parmetis.hpp>
#else defined(AMGCL_HAVE_SCOTCH)
#include <amgcl/mpi/partition/ptscotch.hpp>
#endif

int main(int argc, char *argv[]) {
  // The command line should contain the matrix, the RHS, and the coordinate files:
  if (argc < 4) {
    std::cerr << "Usage: " << argv[0] << " <A.bin> <b.bin> <coo.bin>" << std::endl;
    return 1;
  }

  amgcl::mpi::init mpi(&argc, &argv);
  amgcl::mpi::communicator world(MPI_COMM_WORLD);

  // The profiler:
  amgcl::profiler<> prof("Nullspace");

  // Read the system matrix, the RHS, and the coordinates:
  prof.tic("read");

  // Get the global size of the matrix:
  ptrdiff_t rows = amgcl::io::crs_size<ptrdiff_t>(argv[1]);

  // Split the matrix into approximately equal chunks of rows, and
  // make sure each chunk size is divisible by 3.
  ptrdiff_t chunk = (rows + world.size - 1) / world.size;
  if (chunk % 3) chunk += 3 - chunk % 3;

  ptrdiff_t row_beg = std::min(rows, chunk * world.rank);
  ptrdiff_t row_end = std::min(rows, row_beg + chunk);
  chunk = row_end - row_beg;

  // Read our part of the system matrix, the RHS and the coordinates.
  std::vector<ptrdiff_t> ptr, col;
  std::vector<double> val, rhs, coo;
  amgcl::io::read_crs(argv[1], rows, ptr, col, val, row_beg, row_end);

  ptrdiff_t n, m;
  amgcl::io::read_dense(argv[2], n, m, rhs, row_beg, row_end);
  amgcl::precondition(n == rows && m == 1, "The RHS file has wrong dimensions");
  amgcl::io::read_dense(argv[3], n, m, coo, row_beg / 3, row_end / 3);
  amgcl::precondition(n * 3 == rows && m == 3, "The coordinate file has wrong dimensions");

  prof.toc("read");

  if (world.rank == 0) {
    std::cout << "Matrix " << argv[1] << ": " << rows << "x" << rows << "x" << cols << std::endl
              << "RHS " << argv[2] << ": " << rows << "x1" << "x1" << std::endl
              << "Coords " << argv[3] << ": " << rows / 3 << "x3" << std::endl;
  }
}
```

(continues on next page)
// Declare the backends and the solver type
typedef amgcl::backend::builtin<double> SBackend; // the solver backend
typedef amgcl::backend::builtin<float> PBackend; // the preconditioner backend

typedef amgcl::mpi::make_solver<
    amgcl::mpi::amg<
        PBackend,
        amgcl::mpi::coarsening::smoothed_aggregation<PBackend>,
        amgcl::mpi::relaxation::spai0<PBackend>
    >,
    amgcl::mpi::solver::cg<PBackend>
> Solver;

// The distributed matrix
auto A = std::make_shared<amgcl::mpi::distributed_matrix<SBBackend>>(
    world, std::tie(chunk, ptr, col, val));

// Partition the matrix, the RHS vector, and the coordinates.
// If neither ParMETIS not PT-SCOTCH are not available,
// just keep the current naive partitioning.
#if defined(AMGCL_HAVE_PARMETIS) || defined(AMGCL_HAVE_SCOTCH)
  #if defined(AMGCL_HAVE_PARMETIS)
    typedef amgcl::mpi::partition::parmetis<SBBackend> Partition;
  #elif defined(AMGCL_HAVE_SCOTCH)
    typedef amgcl::mpi::partition::ptscotch<SBBackend> Partition;
  #endif
#endif

if (world.size > 1) {
    auto t = prof.scoped_tic("partition");
    Partition part;

    // par(A) returns the distributed permutation matrix.
    // Keep the DOFs belonging to the same grid nodes together
    // (use block-wise partitioning with block size 3).
    auto P = part(*A, 3);
    auto R = transpose(*P);

    // Reorder the matrix:

    // Reorder the RHS vector and the coordinates:
    R->move_to_backend();
    std::vector<double> new_rhs(R->loc_rows());
    std::vector<double> new_coo(R->loc_rows());
    amgcl::backend::spmv(1, *R, rhs, 0, new_rhs);
    amgcl::backend::spmv(1, *R, coo, 0, new_coo);
    rhs.swap(new_rhs);
    coo.swap(new_coo);

    // Update the number of the local rows
    // (it may have changed as a result of permutation).
    chunk = A->loc_rows();
}
#endif

// Solver parameters:
Solver::params prm;
In lines 44–49 we split the system into approximately equal chunks of rows, while making sure the chunk sizes are divisible by 3 (the number of DOFs per grid node). This is a naive partitioning that will be improved a bit later:

We read the parts of the system matrix, the RHS vector, and the grid node coordinates that belong to the current MPI process in lines 52–61. The backends for the iterative solver and the preconditioner and the solver type are declared in lines 72–82. In lines 85–86 we create the distributed version of the matrix from the local CRS arrays. After that, we are ready to partition the system using AMGCL wrapper for either ParMETIS or PT-SCOTCH libraries (lines 91–123).

Note that we are reordering the coordinate matrix coo in the same way the RHS vector is reordered, even though the coordinate matrix has three times less rows than the system matrix. We can do this because the coordinate matrix is stored in the row-major order, and each row of the matrix has three coordinates, which means the total number of elements in the matrix is equal to the number of elements in the RHS vector, and we can apply our block-wise partitioning to the coordinate matrix.

The coordinates for the current MPI domain are converted into the rigid body modes in lines 135–136, after which we are ready to setup the solver (line 140) and solve the system (line 152). Below is the output of the compiled program:

```
$ export OMP_NUM_THREADS=1
$ mpirun -np 4 nullspace_mpi A.bin b.bin C.bin
Matrix A.bin: 81657x81657
```
RHS b.bin: 81657x1
Coords C.bin: 27219x3
Partitioning[ParMETIS] 4 -> 4
Type: CG
Unknowns: 19965
Memory footprint: 311.95 K

Number of levels: 3
Operator complexity: 1.53
Grid complexity: 1.10

<table>
<thead>
<tr>
<th>level</th>
<th>unknowns</th>
<th>nonzeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>81657</td>
<td>3171111  (65.31%)  [4]</td>
</tr>
<tr>
<td>1</td>
<td>7824</td>
<td>1674144  (34.48%)  [4]</td>
</tr>
<tr>
<td>2</td>
<td>144</td>
<td>10224    (0.21%)    [4]</td>
</tr>
</tbody>
</table>

Iters: 104
Error: 9.26388e-09

[Nullspace: 2.833 s] (100.00%)
[ self: 0.070 s] (2.48%)
[ partition: 0.230 s] (8.10%)
[ read: 0.009 s] (0.32%)
[ setup: 1.081 s] (38.15%)
[ solve: 1.443 s] (50.94%)

### 2.5 Examples

#### 2.5.1 Solving Poisson’s equation

The easiest way to solve a problem with AMGCL is to use the `amgcl::make_solver` class. It has two template parameters: the first one specifies a preconditioner to use, and the second chooses an iterative solver. The class constructor takes the system matrix in one of supported formats and parameters for the chosen algorithms and for the backend.

Let us consider a simple example of Poisson’s equation in a unit square. Here is how the problem may be solved with AMGCL. We will use BiCGStab solver preconditioned with smoothed aggregation multigrid with SPAI(0) for relaxation (smoothing). First, we include the necessary headers. Each of those brings in the corresponding component of the method:

```cpp
#include <amgcl/make_solver.hpp>
#include <amgcl/solver/bicgstab.hpp>
#include <amgcl/amg.hpp>
#include <amgcl/coarsening/smoothed_aggregation.hpp>
#include <amgcl/relaxation/spai0.hpp>
#include <amgcl/adapter/crs_tuple.hpp>
```

Next, we assemble sparse matrix for the Poisson’s equation on a uniform 1000x1000 grid. See below for the definition of the `poisson()` function:

```cpp
std::vector<int> ptr, col;
std::vector<double> val, rhs;
int n = poisson(1000, ptr, col, val, rhs);
```
For this example, we select the `builtin` backend with double precision numbers as value type:

```cpp
typedef amgcl::backend::builtin<double> Backend;
```

Now we can construct the solver for our system matrix. We use the convenient adapter for `std::tuple` here and just tie together the matrix size and its CRS components:

```cpp
typedef amgcl::make_solver<
    // Use AMG as preconditioner:
    amgcl::amg<
        Backend,
        amgcl::coarsening::smoothed_aggregation,
        amgcl::relaxation::spai0
    >,
    // And BiCGStab as iterative solver:
    amgcl::solver::bicgstab<Backend>
> Solver;
```

```cpp
Solver solve( std::tie(n, ptr, col, val) );
```

Once the solver is constructed, we can apply it to the right-hand side to obtain the solution. This may be repeated multiple times for different right-hand sides. Here we start with a zero initial approximation. The solver returns a boost tuple with number of iterations and norm of the achieved residual:

```cpp
std::vector<double> x(n, 0.0);
int iters;
double error;
std::tie(iters, error) = solve(rhs, x);
```

That's it! Vector \( x \) contains the solution of our problem now.

### 2.5.2 Input formats

We used STL vectors to store the matrix components in the above example. This may seem too restrictive if you want to use AMGCL with your own types. But the `crs_tuple` adapter will take anything that the Boost.Range library recognizes as a random access range. For example, you can wrap raw pointers to your data into a `boost::iterator_range`:

```cpp
Solver solve( boost::make_tuple( 
    n, 
    boost::make_iterator_range(ptr.data(), ptr.data() + ptr.size()),
    boost::make_iterator_range(col.data(), col.data() + col.size()),
    boost::make_iterator_range(val.data(), val.data() + val.size())
) );
```

Same applies to the right-hand side and the solution vectors. And if that is still not general enough, you can provide your own adapter for your matrix type. See `Matrix Adapters` for further information on this.

### 2.5.3 Setting parameters

Any component in AMGCL defines its own parameters by declaring a `param` subtype. When a class wraps several subclasses, it includes parameters of its children into its own `param`. For example, parameters for the `amgcl::make_solver<Precond, Solver>` are declared as
struct params {
    typename Precond::params precond;
    typename Solver::params solver;
};

Knowing that, we can easily set the parameters for individual components. For example, we can set the desired tolerance for the iterative solver in the above example like this:

Solver::params prm;
prm.solver.tol = 1e-3;
Solver solve( std::tie(n, ptr, col, val), prm );

Parameters may also be initialized with a boost::property_tree::ptree. This is especially convenient when the runtime interface is used, and the exact structure of the parameters is not known at compile time:

boost::property_tree::ptree prm;
prm.put("solver.tol", 1e-3);
Solver solve( std::tie(n, ptr, col, val), prm );

### 2.5.4 Assembling matrix for Poisson’s equation

The section provides an example of assembling the system matrix and the right-hand side for a Poisson’s equation in a unit square $\Omega = [0,1] \times [0,1]$:

$$-\Delta u = 1, \quad u = 0, \quad u \in \partial \Omega$$

The solution to the problem looks like this:

Here is how the problem may be discretized on a uniform $n \times n$ grid:

```cpp
#include <vector>

// Assembles matrix for Poisson’s equation with homogeneous
// boundary conditions on a n x n grid.
// Returns number of rows in the assembled matrix.
// The matrix is returned in the CRS components ptr, col, and val.
// The right-hand side is returned in rhs.
int poisson(
    int n,
    std::vector<int> &ptr,
    std::vector<int> &col,
    std::vector<double> &val,
    std::vector<double> &rhs
)
{
    int n2 = n * n; // Number of points in the grid.
    double h = 1.0 / (n - 1); // Grid spacing.

    ptr.clear(); ptr.reserve(n2 + 1); ptr.push_back(0);
    col.clear(); col.reserve(n2 * 5); // We use 5-point stencil, so the matrix
    val.clear(); val.reserve(n2 * 5); // will have at most n2 * 5 nonzero elements.

    rhs.resize(n2);

    for(int j = 0, k = 0; j < n; ++j) {
        for(int i = 0; i < n; ++i, ++k) {
```
if (i == 0 || i == n - 1 || j == 0 || j == n - 1) {
    // Boundary point. Use Dirichlet condition.
    col.push_back(k);
    val.push_back(1.0);
    rhs[k] = 0.0;
} else {
    // Interior point. Use 5-point finite difference stencil.
    col.push_back(k - n);
    val.push_back(-1.0 / (h * h));
    col.push_back(k - 1);
    val.push_back(-1.0 / (h * h));
    col.push_back(k);
    val.push_back(4.0 / (h * h));
    col.push_back(k + 1);
    val.push_back(-1.0 / (h * h));
    col.push_back(k + n);
    val.push_back(-1.0 / (h * h));
    rhs[k] = 1.0;
}
ptr.push_back(col.size());
}

2.6 Benchmarks

The performance of the shared memory and the distributed memory versions of AMGCL algorithms was tested on two example problems in a three dimensional space. The source code for the benchmarks is available at https://github.com/ddemidov/amgcl_benchmarks.

The first example is the classical 3D Poisson problem. Namely, we look for the solution of the problem

\[-\Delta u = 1,
\]

in the unit cube \( \Omega = [0, 1]^3 \) with homogeneous Dirichlet boundary conditions. The problem is discretized with the finite difference method on a uniform mesh.

The second test problem is an incompressible 3D Navier-Stokes problem discretized on a non uniform 3D mesh with a finite element method:

\[
\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} + \nabla p = \mathbf{b},
\]

\[
\nabla \cdot \mathbf{u} = 0.
\]

The discretization uses an equal-order tetrahedral Finite Elements stabilized with an ASGS-type (algebraic subgrid-
scale) approach. This results in a linear system of equations with a block structure of the type

\[
\begin{pmatrix}
K & G \\
D & S
\end{pmatrix}
\begin{pmatrix}
u \\
p
\end{pmatrix}
=
\begin{pmatrix}
b_u \\
b_p
\end{pmatrix}
\]

where each of the matrix subblocks is a large sparse matrix, and the blocks \(G\) and \(D\) are non-square. The overall system matrix for the problem was assembled in the Kratos multi-physics package developed in CIMNE, Barcelona.

### 2.6.1 Shared Memory Benchmarks

In this section we test performance of the library on a shared memory system. We also compare the results with PETSC and Trilinos ML distributed memory libraries and CUSP GPGPU library. The tests were performed on a dual socket system with two Intel Xeon E5-2640 v3 CPUs. The system also had an NVIDIA Tesla K80 GPU installed, which was used for testing the GPU based versions.

#### 3D Poisson problem

The Poisson problem is discretized with the finite difference method on a uniform mesh, and the resulting linear system contained 3375000 unknowns and 23490000 nonzeros.

The figure below presents the multicore scalability of the problem. Here AMGCL uses the `builtin` OpenMP backend, while PETSC and Trilinos use MPI for parallelization. We also show results for the CUDA backend of AMGCL library compared with the CUSP library. All libraries use the Conjugate Gradient iterative solver preconditioned with a smoothed aggregation AMG. Trilinos and PETSC use default options for smoothers (symmetric Gauss-Seidel and damped Jacobi accordingly) on each level of the hierarchy, AMGCL uses SPAI0, and CUSP uses Gauss-Seidel smoother.

The CPU-based results show that AMGCL performs on par with Trilinos, and both of the libraries outperform PETSC by a large margin. Also, AMGCL is able to setup the solver about 20–100% faster than Trilinos, and 4–7 times faster than PETSC. This is probably due to the fact that both Trilinos and PETSC target distributed memory machines and hence need to do some complicated bookkeeping under the hood. PETSC shows better scalability than both Trilinos and AMGCL, which scale in a similar fashion.

On the GPU, AMGCL performs slightly better than CUSP. If we consider the solution time (without setup), then both libraries are able to outperform CPU-based versions by a factor of 3-4. The total solution time of AMGCL with CUDA backend is only 30% better than that of either AMGCL with OpenMP backend or Trilinos ML. This is due to the fact that the setup step in AMGCL is always performed on the CPU and in case of the CUDA backend has an additional overhead of moving the constructed hierarchy into the GPU memory.

#### 3D Navier-Stokes problem

The system matrix resulting from the problem discretization has block structure with blocks of 4-by-4 elements, and contains 713456 unknowns and 41277920 nonzeros. The assembled problem is available to download at https://doi.org/10.5281/zenodo.1231818.

There are at least two ways to solve the system. First, one can treat the system as a monolithic one, and provide some minimal help to the preconditioner in form of near null space vectors. Second option is to employ the knowledge about the problem structure, and to combine separate preconditioners for individual fields (in this particular case, for pressure and velocity). In case of AMGCL both options were tested, where the monolithic system was solved with static 4x4 matrices as value type, and the field-split approach was implemented using the `schur_pressure_correction` preconditioner. Trilinos ML only provides the first option; PETSC implement both options, but we only show results for the second, superior option here. CUSP library does not provide field-split preconditioner and does not allow to specify near null space vectors, so it was not tested for this problem.
3D Poisson problem

- Total time
- Setup time
- Solve time
- Iterations

Cores/MPI processes: 1, 2, 4, 8, 16

Graphs showing the performance of AMGCL, PETSC, Trilinos, AMGCL/CUDA, and CUSP for 3D Poisson problem.
The figure below shows multicore scalability results for the Navier-Stokes problem. Lines labelled with ‘block’ correspond to the cases when the problem is treated as a monolithic system, and ‘split’ results correspond to the field-split approach.

2.6.2 Distributed Memory Benchmarks

Here we demonstrate performance and scalability of the distributed memory algorithms provided by AMGCL on the example of a Poisson problem and a Navier-Stokes problem in a three dimensional space. To provide a reference, we compare performance of the AMGCL library with that of the well-established Trilinos ML package. The benchmarks were run on MareNostrum 4, PizDaint, and SuperMUC clusters which we gained access to via PRACE program (project 2010PA4058). The MareNostrum 4 cluster has 3456 compute nodes, each equipped with two 24 core Intel Xeon Platinum 8160 CPUs, and 96 GB of RAM. The peak performance of the cluster is 6.2 Petaflops. The PizDaint cluster has 5320 hybrid compute nodes, where each node has one 12 core Intel Xeon E5-2690 v3 CPU with 64
GB RAM and one NVIDIA Tesla P100 GPU with 16 GB RAM. The peak performance of the PizDaint cluster is 25.3 Petaflops. The SuperMUC cluster allowed us to use 512 compute nodes, each equipped with two 14 core Intel Haswell Xeon E5-2697 v3 CPUs, and 64 GB of RAM.

### 3D Poisson problem

The figure below shows weak scaling of the solution on the SuperMUC cluster. Here the problem size is chosen to be proportional to the number of CPU cores with about $100^3$ unknowns per core. Both AMGCL and Trilinos implementations use a CG iterative solver preconditioned with smoothed aggregation AMG. AMGCL uses SPAI(0) for the smoother, and Trilinos uses ILU(0), which are the corresponding defaults for the libraries. The plots in the figure show total computation time, time spent on constructing the preconditioner, solution time, and the number of iterations. The AMGCL library results are labelled ‘OMP=n’, where n=1,14,28 corresponds to the number of OpenMP threads controlled by each MPI process. The Trilinos library uses single-threaded MPI processes.

Next figure shows strong scaling results for smoothed aggregation AMG preconditioned on the SuperMUC cluster. The problem size is fixed to $256^3$ unknowns and ideally the compute time should decrease as we increase the number of CPU cores. The case of ideal scaling is depicted for reference on the plots with thin gray dotted lines.

The AMGCL implementation uses a BiCGStab(2) iterative solver preconditioned with subdomain deflation, as it...
Strong scaling of the Poisson problem on the SuperMUC cluster

AMGCL, omp=1
AMGCL, omp=14
AMGCL, omp=28
Trilinos

Ideal

2.6. Benchmarks
showed the best behaviour in our tests. Smoothed aggregation AMG is used as the local preconditioner. The Trilinos implementation uses a CG solver preconditioned with smoothed aggregation AMG with default ‘SA’ settings, or domain decomposition method with default ‘DD-ML’ settings.

The figure below shows weak scaling of the solution on the MareNostrum 4 cluster. Here the problem size is chosen to be proportional to the number of CPU cores with about $100^3$ unknowns per core. The rows in the figure from top to bottom show total computation time, time spent on constructing the preconditioner, solution time, and the number of iterations. The AMGCL library results are labelled ‘OMP=n’, where n=1,4,12,24 corresponds to the number of OpenMP threads controlled by each MPI process. The Trilinos library uses single-threaded MPI processes. The Trilinos data is only available for up to 1536 MPI processes, which is due to the fact that only 32-bit version of the library was available on the cluster. The AMGCL data points for 19200 cores with ‘OMP=1’ are missing because factorization of the deflated matrix becomes too expensive for this configuration. AMGCL plots in the left and the right columns correspond to the linear deflation and the constant deflation correspondingly. The Trilinos and Trilinos/DD-ML lines correspond to the smoothed AMG and domain decomposition variants accordingly and are depicted both in the left and the right columns for convenience.

In the case of ideal scaling the timing plots on this figure would be strictly horizontal. This is not the case here: instead, we see that both AMGCL and Trilinos loose about 6-8% efficiency whenever the number of cores doubles. The AMGCL algorithm performs about three times worse that the AMG-based Trilinos version, and about 2.5 times better than the domain decomposition based Trilinos version. This is mostly governed by the number of iterations each version needs to converge.

We observe that AMGCL scalability becomes worse at the higher number of cores. We refer to the following table for the explanation:

<table>
<thead>
<tr>
<th>Cores</th>
<th>Setup</th>
<th>Solve</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Factorize E</td>
<td></td>
</tr>
<tr>
<td>Linear deflation, OMP=1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>384</td>
<td>4.23</td>
<td>0.02</td>
<td>54.08</td>
</tr>
<tr>
<td>1536</td>
<td>6.01</td>
<td>0.64</td>
<td>57.19</td>
</tr>
<tr>
<td>6144</td>
<td>13.92</td>
<td>8.41</td>
<td>48.40</td>
</tr>
<tr>
<td>Constant deflation, OMP=1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>384</td>
<td>3.11</td>
<td>0.00</td>
<td>61.41</td>
</tr>
<tr>
<td>1536</td>
<td>4.52</td>
<td>0.01</td>
<td>73.98</td>
</tr>
<tr>
<td>6144</td>
<td>5.67</td>
<td>0.16</td>
<td>64.13</td>
</tr>
<tr>
<td>Linear deflation, OMP=12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>384</td>
<td>8.35</td>
<td>0.00</td>
<td>72.68</td>
</tr>
<tr>
<td>1536</td>
<td>7.95</td>
<td>0.00</td>
<td>82.22</td>
</tr>
<tr>
<td>6144</td>
<td>16.08</td>
<td>0.03</td>
<td>77.00</td>
</tr>
<tr>
<td>19200</td>
<td>42.09</td>
<td>1.76</td>
<td>90.74</td>
</tr>
<tr>
<td>Constant deflation, OMP=12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>384</td>
<td>7.02</td>
<td>0.00</td>
<td>72.25</td>
</tr>
<tr>
<td>1536</td>
<td>6.64</td>
<td>0.00</td>
<td>102.53</td>
</tr>
<tr>
<td>6144</td>
<td>15.02</td>
<td>0.00</td>
<td>75.82</td>
</tr>
<tr>
<td>19200</td>
<td>36.08</td>
<td>0.03</td>
<td>119.25</td>
</tr>
</tbody>
</table>

The table presents the profiling data for the solution of the Poisson problem on the MareNostrum 4 cluster. The first two columns show time spent on the setup of the preconditioner and the solution of the problem; the third column shows the number of iterations required for convergence. The ‘Setup’ column is further split into subcolumns detailing the total setup time and the time required for factorization of the coarse system. It is apparent from the table that factorization of the coarse (deflated) matrix starts to dominate the setup phase as the number of subdomains (or MPI processes) grows, since we use a sparse direct solver for the coarse problem. This explains the fact that the constant deflation scales better, since the deflation matrix is four times smaller than for a corresponding linear deflation case.

The advantage of the linear deflation is that it results in a better approximation of the problem on a coarse scale and
Weak scaling of the Poisson problem on the MareNostrum 4 cluster

Linear deflation

Constant deflation

Total time

Setup time

Solve time

Iterations

Number of cores (MPI * OMP)

Number of cores (MPI * OMP)

OMP=1
OMP=4
OMP=12
OMP=24
Trilinos/ML
Trilinos/DD-ML
hence needs less iterations for convergence and performs slightly better within its scalability limits, but the constant deflation eventually outperforms linear deflation as the scale grows.

Next figure shows weak scaling of the Poisson problem on the PizDaint cluster. The problem size here is chosen so that each node owns about $200^3$ unknowns. On this cluster we are able to compare performance of the OpenMP and CUDA backends of the AMGCL library. Intel Xeon E5-2690 v3 CPU is used with the OpenMP backend, and NVIDIA Tesla P100 GPU is used with the CUDA backend on each compute node. The scaling behavior is similar to the MareNostrum 4 cluster. We can see that the CUDA backend is about 9 times faster than OpenMP during solution phase and 4 times faster overall. The discrepancy is explained by the fact that the setup phase in AMGCL is always performed on the CPU, and in the case of CUDA backend it has the additional overhead of moving the generated hierarchy into the GPU memory. It should be noted that this additional cost of setup on a GPU (and the cost of setup in general) often can amortized by reusing the preconditioner for different right-hand sides. This is often possible for non-linear or time dependent problems. The performance of the solution step of the AMGCL version with the CUDA backend here is on par with the Trilinos ML package. Of course, this comparison is not entirely fair to Trilinos, but it shows the advantages of using CUDA technology.

The following figure shows strong scaling results for the MareNostrum 4 cluster. The problem size is fixed to $512^3$ unknowns and ideally the compute time should decrease as we increase the number of CPU cores. The case of ideal scaling is depicted for reference on the plots with thin gray dotted lines.

Here, AMGCL demonstrates scalability slightly better than that of the Trilinos ML package. At 384 cores the AMGCL solution for OMP=1 is about 2.5 times slower than Trilinos/AMG, and 2 times faster than Trilinos/DD-ML. As is expected for a strong scalability benchmark, the drop in scalability at higher number of cores for all versions of the tests is explained by the fact that work size per each subdomain becomes too small to cover both setup and communication costs.

The profiling data for the strong scaling case is shown in the table below, and it is apparent that, as in the weak scaling scenario, the deflated matrix factorization becomes the bottleneck for the setup phase performance.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Setup</th>
<th>Solve</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Factorize E</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linear deflation, OMP=1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>384</td>
<td>1.27</td>
<td>0.02</td>
<td>12.39</td>
</tr>
<tr>
<td>1536</td>
<td>0.97</td>
<td>0.45</td>
<td>2.93</td>
</tr>
<tr>
<td>6144</td>
<td>9.09</td>
<td>8.44</td>
<td>3.61</td>
</tr>
<tr>
<td>Constant deflation, OMP=1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>384</td>
<td>1.14</td>
<td>0.00</td>
<td>16.30</td>
</tr>
<tr>
<td>1536</td>
<td>0.38</td>
<td>0.01</td>
<td>3.71</td>
</tr>
<tr>
<td>6144</td>
<td>0.82</td>
<td>0.16</td>
<td>1.19</td>
</tr>
<tr>
<td>Linear deflation, OMP=12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>384</td>
<td>2.90</td>
<td>0.00</td>
<td>16.57</td>
</tr>
<tr>
<td>1536</td>
<td>1.43</td>
<td>0.00</td>
<td>4.15</td>
</tr>
<tr>
<td>6144</td>
<td>0.68</td>
<td>0.03</td>
<td>1.35</td>
</tr>
<tr>
<td>19200</td>
<td>1.66</td>
<td>1.29</td>
<td>1.80</td>
</tr>
<tr>
<td>Constant deflation, OMP=12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>384</td>
<td>2.49</td>
<td>0.00</td>
<td>18.25</td>
</tr>
<tr>
<td>1536</td>
<td>0.62</td>
<td>0.00</td>
<td>4.91</td>
</tr>
<tr>
<td>6144</td>
<td>0.35</td>
<td>0.00</td>
<td>1.37</td>
</tr>
<tr>
<td>19200</td>
<td>0.32</td>
<td>0.02</td>
<td>1.89</td>
</tr>
</tbody>
</table>

An interesting observation is that convergence of the method improves with growing number of MPI processes. In other words, the number of iterations required to reach the desired tolerance decreases with as the number of subdomains grows, since the deflated system is able to describe the main problem better and better. This is especially apparent from the strong scalability results, where the problem size remains fixed, but is also observable in the weak scaling case for ‘OMP=1’.
Weak scaling of the Poisson problem on PizDaint cluster

**Linear deflation**

- Total time
- Setup time
- Solve time
- Iterations

**Constant deflation**

- Total time
- Setup time
- Solve time
- Iterations

Compute nodes: 2, 8, 32, 128, 512, 2048

Iterations: 0, 50, 100, 150

GPU, CPU (OMP=12), Trilinos

**2.6. Benchmarks**
2.6. Benchmarks

Strong scaling of the Poisson problem on the MareNostrum 4 cluster

- Linear deflation
- Constant deflation

**Total time**

- OMP=1
- OMP=12
- OMP=24
- Trilinos/ML
- Trilinos/DD-ML
- Ideal scaling
3D Navier-Stokes problem

The system matrix in these tests contains 4773588 unknowns and 281089456 nonzeros. The assembled system is available to download at https://doi.org/10.5281/zenodo.1231961. AMGCL library uses field-split approach with the mpi::schur_pressure_correction preconditioner. Trilinos ML does not provide field-split type preconditioners, and uses the nonsymmetric smoothed aggregation variant (NSSA) applied to the monolithic problem. Default NSSA parameters were employed in the tests.

The figure below shows scalability results for the Navier-Stokes problem on the SuperMUC cluster. In case of AMGCL, the pressure part of the system is preconditioned with a smoothed aggregation AMG. Since we are solving a fixed-size problem, this is essentially a strong scalability test.

![Strong scaling of the Navier-Stokes problem on the SuperMUC cluster](image)

The next figure shows scalability results for the Navier-Stokes problem on the MareNostrum 4 cluster. Since we are solving a fixed-size problem, this is essentially a strong scalability test.

Both AMGCL and ML preconditioners deliver a very flat number of iterations with growing number of MPI processes. As expected, the field-split preconditioner pays off and performs better than the monolithic approach in the solution of the problem. Overall the AMGCL implementation shows a decent, although less than optimal parallel scalability. This is not unexpected since the problem size quickly becomes too little to justify the use of more parallel resources (note that at 192 processes, less than 25000 unknowns are assigned to each MPI subdomain). Unsurprisingly, in this
Strong scaling of the Navier-Stokes problem on MareNostrum 4 cluster

Total time

Setup time

Solve time

Iterations

Number of cores (MPI * OMP)

Number of cores (MPI * OMP)

96 192 384 768 1536 3072 6144

96 192 384 768 1536 3072 6144

100

101

102

2.6. Benchmarks
context the use of OpenMP within each domain pays off and allows delivering a greater level of scalability.

2.7 Bibliography
Bibliography


[Alex00] A. Alexandrescu, Modern C++ design: generic programming and design patterns applied, AddisonWesley, 2001.


amgcl::adapter::reorder (C++ class), 14
amgcl::amg (C++ class), 22
amgcl::amg::backend_params (C++ type), 22
amgcl::amg::coarsening_type (C++ type), 22
amgcl::amg::params (C++ class), 22
amgcl::amg::params::coarse_enough (C++ member), 23
amgcl::amg::params::coarsening (C++ member), 23
amgcl::amg::params::coarsening_params (C++ type), 23
amgcl::amg::params::direct_coarse (C++ member), 23
amgcl::amg::params::max_levels (C++ member), 23
amgcl::amg::params::ncycle (C++ member), 23
amgcl::amg::params::npost (C++ member), 23
amgcl::amg::params::npre (C++ member), 23
amgcl::amg::params::pre_cycles (C++ member), 23
amgcl::amg::params::relax (C++ member), 23
amgcl::amg::params::relax_params (C++ type), 23
amgcl::amg::relax_type (C++ type), 22
amgcl::amg::scalar_type (C++ type), 22
amgcl::amg::value_type (C++ type), 22
amgcl::backend::blaze (C++ class), 10
amgcl::backend::blaze::params (C++ class), 10
amgcl::backend::builtin (C++ class), 9
amgcl::backend::builtin::params (C++ class), 9
amgcl::backend::cuda (C++ class), 9
amgcl::backend::cuda::params (C++ class), 9
amgcl::backend::cuda::params::cusparse_handle (C++ member), 9
amgcl::backend::eigen (C++ class), 10
amgcl::backend::eigen::params (C++ class), 10
amgcl::backend::vexcl (C++ class), 9
amgcl::backend::vexcl::params (C++ class), 9
amgcl::backend::vexcl::params::fast_matrix_setup (C++ member), 9
amgcl::backend::vexcl::params::q (C++ member), 9
amgcl::backend::viennacl (C++ class), 10
amgcl::backend::viennacl::params (C++ class), 10
amgcl::coarsening::aggregation (C++ class), 33
amgcl::coarsening::aggregation::params (C++ class), 33
amgcl::coarsening::aggregation::params::nullspace (C++ member), 33
amgcl::coarsening::aggregation::params::over_interp (C++ member), 33
amgcl::coarsening::nullspace_params (C++ class), 33
amgcl::coarsening::nullspace_params::B (C++ member), 33
amgcl::coarsening::nullspace_params::cols (C++ member), 33
amgcl::coarsening::pointwise_aggregates (C++ class), 33
amgcl::coarsening::pointwise_aggregates::params (C++ class), 33
amgcl::coarsening::pointwise_aggregates::params::b (C++ member), 33
amgcl::coarsening::pointwise_aggregates::params::eps_strong (C++ member), 33
amgcl::coarsening::ruge_stuben (C++ class), 32
amgcl::coarsening::ruge_stuben::params (C++ class), 32
amgcl::coarsening::ruge_stuben::params::do_trunc (C++ member), 32
Index 139

B
  block_matrix (C++ function), 13

C
  constructor (C++ function), 14

O
  operator() (C++ function), 14, 15

S
  scaled_diagonal (C++ function), 13

Z
  zero_copy (C++ function), 12