AMGCL Documentation

Release 1.0.0.post69

Denis Demidov

Sep 26, 2017

Contents

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 *[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).

AMGCL builds the AMG hierarchy on a CPU and then transfers it to 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 library source code is available under MIT license at https://github.com/ddemidov/amgcl.

CHAPTER 1

Contents:

Getting started

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*.

Solving Poisson's equation

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:

```
#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 *Assembling matrix for Poisson's equation* for the source code of the `poisson()` function:

```
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:

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

Now we can construct the solver for our system matrix. We use the convenient adapter for boost tuples here and just tie together the matrix size and its CRS components:

```
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;

Solver solve( boost::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:

```
std::vector<double> x(n, 0.0);
int    iters;
double error;
boost::tie(iters, error) = solve(rhs, x);
```

That's it! Vector `x` contains the solution of our problem now.

Input formats

We used STL vectors to store the matrix components in the above axample. 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:

```
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.

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( boost::tie(n, ptr, col, val), prm );
```

Parameters may also be initialized with a boost::property_tree::ptree. This is especially convenient when *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( boost::tie(n, ptr, col, val), prm );
```

The `make_solver` class

template <**class** Precond, **class** IterativeSolver>
class `amgcl::make_solver`

Convenience class that bundles together a preconditioner and an iterative solver.

Public Functions

template <**class** Matrix>
make_solver(**const** Matrix &*A*, **const** *params* &*prm* = params (), **const** backend_params &*bprm* = backend_params())

Sets up the preconditioner and creates the iterative solver.

template <**class** Matrix, **class** Vec1, **class** Vec2>
boost::tuple<size_t, scalar_type> **operator()**(Matrix **const** &*A*, Vec1 **const** &*rhs*, Vec2 &&*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 `boost::tuple`. The solution vector `x` provides initial approximation in 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]*.

template <**class** Vec1, **class** Vec2>
boost::tuple<size_t, scalar_type> **operator()**(Vec1 **const** &*rhs*, Vec2 &&*x*) **const**

Computes the solution for the given right-hand side `rhs`. Returns the number of iterations made and the achieved residual as a `boost::tuple`. The solution vector `x` provides initial approximation in input and holds the computed solution on output.

template <**class** Vec1, **class** Vec2>
void **apply**(**const** Vec1 &*rhs*, Vec2 &&*x*) **const**

Acts as a preconditioner. That is, applies the solver to the right-hand side `rhs` to get the solution `x` with zero initial approximation. Iterative methods usually use estimated residual for exit condition. For some problems the value of the estimated residual can get too far from the true residual due to round-off errors. Nesting iterative solvers in this way may allow to shave the last bits off the error. The method should not be used directly but rather allows nesting *`make_solver`* classes as in the following example:

```
typedef amgcl::make_solver<
  amgcl::make_solver<
    amgcl::amg<
      Backend, amgcl::coarsening::smoothed_aggregation,␣
↪amgcl::relaxation::spai0
```

```
    >,
  amgcl::solver::cg<Backend>
  >,
amgcl::solver::cg<Backend>
> NestedSolver;
```

const Precond &**precond**() **const**
Returns reference to the constructed preconditioner.

const IterativeSolver &**solver**() **const**
Returns reference to the constructed iterative solver.

Precond::matrix **const** &**system_matrix**() **const**
Returns the system matrix in the backend format.

void **get_params**(boost::property_tree::ptree &*p*) **const**
Stores the parameters used during construction into the property tree `p`.

size_t **size**() **const**
Returns the size of the system matrix.

struct params
Combined parameters of the bundled preconditioner and the iterative solver.

Public Members

template<>
Precond::params **precond**
Preconditioner parameters.

template<>
IterativeSolver::params **solver**
Iterative solver parameters.

Components

Matrix adapters

Matrix adapters in AMGCL allow 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 access to the nonzero matrix values. An example of creating an adapter is provided in *Adapting custom matrix class*.

Boost tuple adapter

`#include` <amgcl/adapter/crs_tuple.hpp>

The Boost tuple adapter allows to use a `boost::tuple` of a matrix size and its three CRS format components (row pointer array, column indices array, and values array) as input matrix to AMGCL solvers. The arrays are allowed to be in any format recognized by the Boost.Range library as a random access range. Common examples are STL vectors and Boost iterator ranges.

Example:

```
// boost::tie creates a tuple of references, which avoids copying.
Solver solve( boost::tie(n, ptr, col, val) );

// A (cheap) copy is required when iterator ranges are created on the fly:
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())
    ) );
```

Boost.uBLAS adapter

`#include <amgcl/adapter/ublas.hpp>`

The Boost.uBLAS adapter allows to use uBLAS sparse matrices as input to AMGCL solvers. It also allows to use uBLAS dense vectors with *amgcl::backend::builtin*.

Example:

```
namespace ublas = boost::numeric::ublas;

ublas::compressed_matrix<double> A;
...
Solver solve(A);

ublas::vector<double> rhs, x;
...
solve(rhs, x);
```

Zero copy adapter

`#include <amgcl/adapter/zero_copy.hpp>`

In general, AMGCL copies the adapted input matrix into its internal structures, so that the matrix may be safely destroyed or reused as soon as the solver setup is complete. However, the memory overhead of the copying may be too large, especially for large problems that eat up almost all of available RAM. The zero copy adapter allows to use raw pointers to CRS arrays as input matrix for MAGCL solvers. The data from the arrays is never copied during setup, and the user has to make sure the arrays stay alive long enough. However, unless the backend used is *amgcl::backend::builtin*, the input matrix will be copied into the backend structures when the setup is finished. This would still allow to save some memory in case of GPGPU backends.

The one requirement is that the integer types stored in row pointers and column indices arrays have to be binary compatible with `ptrdiff_t`, and the value type has to be the value type of the backend.

Example:

```
Solver solve( amgcl::adapter::zero_copy(n, &ptr[0], &col[0], &val[0]) );
```

Backends

A backend in AMGCL is a class that defines matrix and vector types together with several operations on them, such as creation, matrix-vector products, elementwise vector operations, inner products etc. The <amgcl/backend/interface.hpp> file defines an interface that each backend should extend. The AMG hierarchy is

moved to the specified backend upon construction. The solution phase then uses types and operations defined in the backend. This enables transparent acceleration of the solution phase with OpenMP, OpenCL, CUDA, or any other technologies.

In order to use a backend, user must include its definition from the corresponding file inside amgcl/backend folder. On the user side of things, only the types of the right-hand side and the solution vectors should be affected by the choice of AMGCL backend. Here is an example of using the *builtin* backend. First, we need to include the appropriate header:

```
#include <amgcl/backend/builtin.hpp>
```

Then, we need to construct the solver and apply it to the vector types supported by the backend:

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

typedef amgcl::make_solver<
    amgcl::amg<Backend, amgcl::coarsening::aggregation, amgcl::relaxation::spai0>,
    amgcl::solver::gmres<Backend>
    > Solver;

Solver solve(A);

std::vector<double> rhs, x; // Initialized elsewhere

solve(rhs, x);
```

Now, if we want to switch to a different backend, for example, in order to accelerate the solution phase with a powerful GPU, we just need to include another backend header, and change the definitions of `Backend`, `rhs`, and `x`. Here is an example of what needs to be done to use the *VexCL* backend.

Include the correct header:

```
#include <amgcl/backend/builtin.hpp>
```

Change the definition of `Backend`:

```
typedef amgcl::backend::vexcl<double> Backend;
```

Change the definition of the vectors:

```
vex::vector<double> rhs, x;
```

That's it! Well, almost. In case the backend requires some parameters, we also need to provide those. In particular, the VexCL backend should know what VexCL context to use:

```
// Initialize VexCL context on a single GPU:
vex::Context ctx(vex::Filter::GPU && vex::Filter::Count(1));

// Create backend parameters:
Backend::params backend_prm;
backend_prm.q = ctx;

// Pass the parameters to the solver constructor:
Solver solve(A, Solver::params(), backend_prm);
```

Builtin

`#include <amgcl/backend/builtin.hpp>`
template <typename *ValueType*>
struct `amgcl::backend::builtin`

The builtin backend does not have any dependencies except for the Boost libraries, and uses OpenMP for parallelization. Matrices are stored in the CRS format, and vectors are instances of `std::vector<value_type>`. There is no usual overhead of moving the constructed hierarchy to the builtin backend, since the backend is used internally during setup.

Public Types

typedef amgcl::detail::empty_params **params**

The backend has no parameters.

struct **provides_row_iterator**

Inherits from true_type

VexCL

`#include <amgcl/backend/vexcl.hpp>`
template <typename real, **class** DirectSolver = solver::vexcl_skyline_lu<real>>
struct `amgcl::backend::vexcl`

The backend uses the VexCL library for accelerating solution on the modern GPUs and multicore processors with the help of OpenCL or CUDA technologies. The VexCL backend stores the system matrix as `vex::SpMat<real>` and expects the right hand side and the solution vectors to be instances of the `vex::vector<real>` type.

struct **params**

The VexCL backend parameters.

Public Members

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

Command queues that identify compute devices to use with VexCL.

template<>
bool **fast_matrix_setup**

Do CSR to ELL conversion on the GPU side.

This will result in faster setup, but will require more GPU memory.

struct **provides_row_iterator**

Inherits from false_type

Preconditioners

Iterative solvers

AMGCL provides several iterative solvers, but it should be easy to use AMGCL preconditioners with a user-provided solver as well. Each solver in AMGCL is a class template. Its single template parameter specifies the backend to use.

This allows to preallocate necessary resources at class construction. Obviously, the solver backend has to coincide with the preconditioner backend.

Each of the solvers in AMGCL provides two overloads for the `operator()`:

boost::tuple<size_t, scalar_type> **operator()**(**const** Matrix &*A*, **const** Precond &*P*, **const** Vec1 &*rhs*, Vec2 &&*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 `boost::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]*.

boost::tuple<size_t, scalar_type> **operator()**(**const** Precond &*P*, **const** Vec1 &*rhs*, Vec2 &&*x*) **const**

Computes the solution for the given right-hand side `rhs`. The system matrix is the same that was used for the setup of the preconditioner `P`. Returns the number of iterations made and the achieved residual as a `boost::tuple`. The solution vector `x` provides initial approximation on input and holds the computed solution on output.

Conjugate Gradient

`#include <amgcl/solver/cg.hpp>`
template <**class** Backend, **class** InnerProduct = detail::default_inner_product>
class `amgcl::solver::cg`

Conjugate Gradients method. An effective method for symmetric positive definite systems *[Barr94]*.

Public Functions

cg(size_t *n*, **const** *params* &*prm* = params (), **const** backend_params &*backend_prm* = backend_params(), **const** InnerProduct &*inner_product* = InnerProduct())

Preallocates necessary data structures for the system of size `n`.

struct **params**

Solver parameters.

Public Members

template<>
size_t **maxiter**

Maximum number of iterations.

template<>
scalar_type **tol**

Target relative residual error.

template<>
scalar_type **abstol**

Target absolute residual error.

BiConjugate Gradient Stabilized (BiCGSTAB)

`#include <amgcl/solver/bicgstab.hpp>`

template <**class** Backend, **class** InnerProduct = detail::default_inner_product>
class `amgcl::solver::bicgstab`
BiConjugate Gradient Stabilized (BiCGSTAB) method. The BiConjugate Gradient Stabilized method (BiCGSTAB) was developed to solve nonsymmetric linear systems while avoiding the often irregular convergence patterns of the Conjugate Gradient *[Barr94]*.

Public Functions

bicgstab(size_t *n*, **const** *params* &*prm* = params (), **const** backend_params &*backend_prm* = backend_params(), **const** InnerProduct &*inner_product* = InnerProduct())
Preallocates necessary data structures for the system of size `n`.

struct **params**
Solver parameters.

Public Members

template<>
size_t **maxiter**
Maximum number of iterations.

template<>
scalar_type **tol**
Target relative residual error.

template<>
scalar_type **abstol**
Target absolute residual error.

BiCGSTAB(L)

`#include <amgcl/solver/bicgstabl.hpp>`
template <**class** Backend, **class** InnerProduct = detail::default_inner_product>
class `amgcl::solver::bicgstabl`
BiCGStab(L) method. Generalization of BiCGStab method *[SlDi93]*.

Public Functions

bicgstabl(size_t *n*, **const** *params* &*prm* = params (), **const** backend_params &*backend_prm* = backend_params(), **const** InnerProduct &*inner_product* = InnerProduct())
Preallocates necessary data structures for the system of size `n`.

struct **params**
Solver parameters.

Public Members

template<>
int **L**
Order of the method.

template<>
size_t **maxiter**
Maximum number of iterations.

template<>
scalar_type **tol**
Target relative residual error.

template<>
scalar_type **abstol**
Target absolute residual error.

GMRES

`#include <amgcl/solver/gmres.hpp>`
template <**class** Backend, **class** InnerProduct = detail::default_inner_product>
class `amgcl::solver::gmres`
Generalized Minimal Residual (GMRES) method. The Generalized Minimal Residual method is an extension of MINRES (which is only applicable to symmetric systems) to unsymmetric systems *[Barr94]*.

Public Functions

gmres(size_t *n*, **const** *params* &*prm* = params(), **const** backend_params &*backend_prm* = backend_params(), **const** InnerProduct &*inner_product* = InnerProduct())
Preallocates necessary data structures for the system of size `n`.

struct **params**
Solver parameters.

Public Members

template<>
unsigned **M**
Number of iterations before restart.

template<>
unsigned **maxiter**
Maximum number of iterations.

template<>
scalar_type **tol**
Target relative residual error.

template<>
scalar_type **abstol**
Target absolute residual error.

LGMRES

`#include <amgcl/solver/lgmres.hpp>`
template <**class** Backend, **class** InnerProduct = detail::default_inner_product>
class `amgcl::solver::lgmres`
"Loose" GMRES. The LGMRES algorithm *[BaJM05]* is designed to avoid some problems in the convergence in restarted GMRES, and often converges in fewer iterations.

Public Functions

lgmres (size_t *n*, **const** *params* &*prm* = params (), **const** backend_params &*bprm* = backend_params(), **const** InnerProduct &*inner_product* = InnerProduct())
Preallocates necessary data structures for the system of size `n`.

struct params
Solver parameters.

Public Members

template<>
precond::type **pside**
Preconditioning kind (left/right).

template<>
unsigned **M**
Number of inner GMRES iterations per each outer iteration.

template<>
unsigned **K**
Number of vectors to carry between inner GMRES iterations.

According to [BaJM05], good values are in the range of 1...3. However, note that if you want to use the additional vectors to accelerate solving multiple similar problems, larger values may be beneficial.

template<>
bool **always_reset**
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.

template<>
bool **store_Av**
Whether LGMRES should store also A*v in addition to vectors `v`.

template<>
size_t **maxiter**
Maximum number of iterations.

template<>
scalar_type **tol**
Target relative residual error.

template<>
scalar_type **abstol**
Target absolute residual error.

FGMRES

`#include <amgcl/solver/fgmres.hpp>`
template <**class** Backend, **class** InnerProduct = detail::default_inner_product>
class `amgcl::solver::fgmres`
Flexible GMRES method. Flexible version of the GMRES method *[Saad03]*.

Public Functions

fgmres (size_t *n*, **const** *params* &*prm* = params (), **const** backend_params &*bprm* = backend_params(), **const** InnerProduct &*inner_product* = InnerProduct())
Preallocates necessary data structures for the system of size `n`.

struct params
Solver parameters.

Public Members

template<>
unsigned **M**
Number of inner GMRES iterations per each outer iteration.

template<>
unsigned **maxiter**
Maximum number of iterations.

template<>
scalar_type **tol**
Target relative residual error.

template<>
scalar_type **abstol**
Target absolute residual error.

Runtime interface

The compile-time configuration of the solvers used in AMGCL is not always convenient, especially if the solvers are used inside a software package or another library. That is why AMGCL provides runtime interface, which allows to postpone the configuration until, well, runtime. The classes inside `amgcl::runtime` namespace correspond to their compile-time alternatives, but the only template parameter they have is the backend to use.

Since there is no way of knowing 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. The runtime interface provides some enumerations for this purpose. For example, to select smoothed aggregation for coarsening, we could do this:

```
boost::property_tree::ptree prm;
prm.put("precond.coarsening.type", amgcl::runtime::coarsening::smoothed_aggregation);
```

The enumerations provide functions for converting to/from strings, so the following would work as well:

```
prm.put("precond.coarsening.type", "smoothed_aggregation");
```

Here is an example of using a runtime-configurable solver:

```
#include <amgcl/backend/builtin.hpp>
#include <amgcl/runtime.hpp>

...

boost::property_tree::ptree prm;
```

```
prm.put("precond.coarsening.type", amgcl::runtime::coarsening::smoothed_aggregation);
prm.put("precond.relaxation.type", amgcl::runtime::relaxation::spai0);
prm.put("solver.type",             amgcl::runtime::solver::gmres);

amgcl::make_solver<
    amgcl::runtime::amg<Backend>,
    amgcl::runtime::iterative_solver<Backend>
    > solve(A, prm);
```

Classes

AMG preconditioner

template <class Backend>
class `amgcl::runtime::amg`

Runtime-configurable AMG preconditioner.

Inherits from noncopyable

Public Functions

template <class Matrix>
amg(**const** Matrix &*A*, params *prm* = params(), **const** backend_params &*backend_prm* = backend_params())

Constructs the AMG hierarchy for the system matrix `A`. `prm` is an instance of `boost::property_tree::ptree` class. The property tree may contain parameters "coarsening.type" and "relax.type". Default values are `amgcl::runtime::coarsening::smoothed_aggregation` and `runtime::relaxation::spai0`. The rest of the property tree should copy the structure of the corresponding `amgcl::amg::params` struct. For example, when smoothed aggregation is selected for coarsening, one could:

```
prm.put("coarsening.aggr.eps_strong", 1e-2);
```

Note: Any parameters that are not relevant to the selected AMG components are silently ignored.

template <class Vec1, **class** Vec2>
void **cycle**(**const** Vec1 &*rhs*, Vec2 &*x*) **const**

Performs single V-cycle for the given right-hand side `rhs` and solution `x`.

template <class Vec1, **class** Vec2>
void **apply**(**const** Vec1 &*rhs*, Vec2 &*x*) **const**

Performs single V-cycle for the given right-hand side `rhs` after clearing `x`. This is intended for use as a preconditioning procedure.

const matrix &**system_matrix**() **const**

Returns the system matrix in the backend format

size_t **size**() **const**

Returns the problem size at the finest level.

Friends

std::ostream &**operator<<**(std::ostream &*os*, **const** amg &*a*)
Prints some info about the AMG hierarchy to the output stream.

enum amgcl::runtime::coarsening::**type**
Values:

ruge_stuben
Ruge-Stueben coarsening.

aggregation
Aggregation.

smoothed_aggregation
Smoothed aggregation.

smoothed_aggr_emin
Smoothed aggregation with energy minimization.

enum amgcl::runtime::relaxation::**type**
Relaxation schemes.

Values:

gauss_seidel
Gauss-Seidel smoothing.

ilu0
Incomplete LU with zero fill-in.

iluk
Level-based incomplete LU.

ilut
Incomplete LU with thresholding.

damped_jacobi
Damped Jacobi.

spai0
Sparse approximate inverse of 0th order.

spai1
Sparse approximate inverse of 1st order.

chebyshev
Chebyshev relaxation.

Iterative solver

template <**class** Backend, **class** InnerProduct = amgcl::solver::detail::default_inner_product>
class amgcl::runtime::**iterative_solver**
This is runtime wrapper around AMGCL iterative solver types. Allows to select the actual solver at runtime.

Public Functions

iterative_solver(size_t *n*, params *prm* = params(), **const** backend_params &*bprm* = backend_params(), **const** InnerProduct &*inner_product* = InnerProduct())
Constructs the iterative solver for the problem size `n`. The property tree `solver_prm` may con-

tain "type" entry that would determine the actual type of the iterative solver. Default value: `amgcl::runtime::solver::bicgstab`.

Note: Any parameters that are not relevant to the selected solver are silently ignored.

template <class Matrix, **class** Precond, **class** Vec1, **class** Vec2>
boost::tuple<size_t, scalar_type> **operator()** (Matrix **const** &*A*, Precond **const** &*P*, Vec1 **const** &*rhs*, Vec2 &&*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 `boost::tuple`. The solution vector `x` provides initial approximation in 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]*.

template <class Precond, **class** Vec1, **class** Vec2>
boost::tuple<size_t, scalar_type> **operator()** (Precond **const** &*P*, Vec1 **const** &*rhs*, Vec2 &&*x*) **const**

Computes the solution for the given right-hand side `rhs`. The system matrix is the same that was used for the setup of the preconditioner `P`. Returns the number of iterations made and the achieved residual as a `boost::tuple`. The solution vector `x` provides initial approximation in input and holds the computed solution on output.

enum `amgcl::runtime::solver::`**type**

Values:

cg
Conjugate gradients method.

bicgstab
BiConjugate Gradient Stabilized.

bicgstabl
BiCGStab(ell)

gmres
GMRES.

lgmres
LGMRES.

fgmres
FGMRES.

idrs
IDR(s)

Subdomain deflation

The capability to solve large and sparse systems of equations is a cornerstone of modern numerical methods, sparse linear systems of equations being ubiquitous in engineering and physics. Direct techniques, despite their attractiveness, simply become not viable beyond a certain size, typically of the order of the few millions of unknowns, due to their intrinsic memory requirements and shear computational cost.

Hence, preconditioned iterative methods become the only feasible alternative in addressing such large scale problems. In practice it is customary to blend the capability of multigrid and Krylov techniques so to preserve some of the advantages of each. A number of successful (distributed memory) libraries exist implementing different flavors of

such blending. Even though such implementation were proven to be weakly scalable, their strong scalability remains questionable. This fact motivated the renowned interest in Domain Decomposition (DD) methods as well as the rise of a new class of approaches named *deflation techniques*.

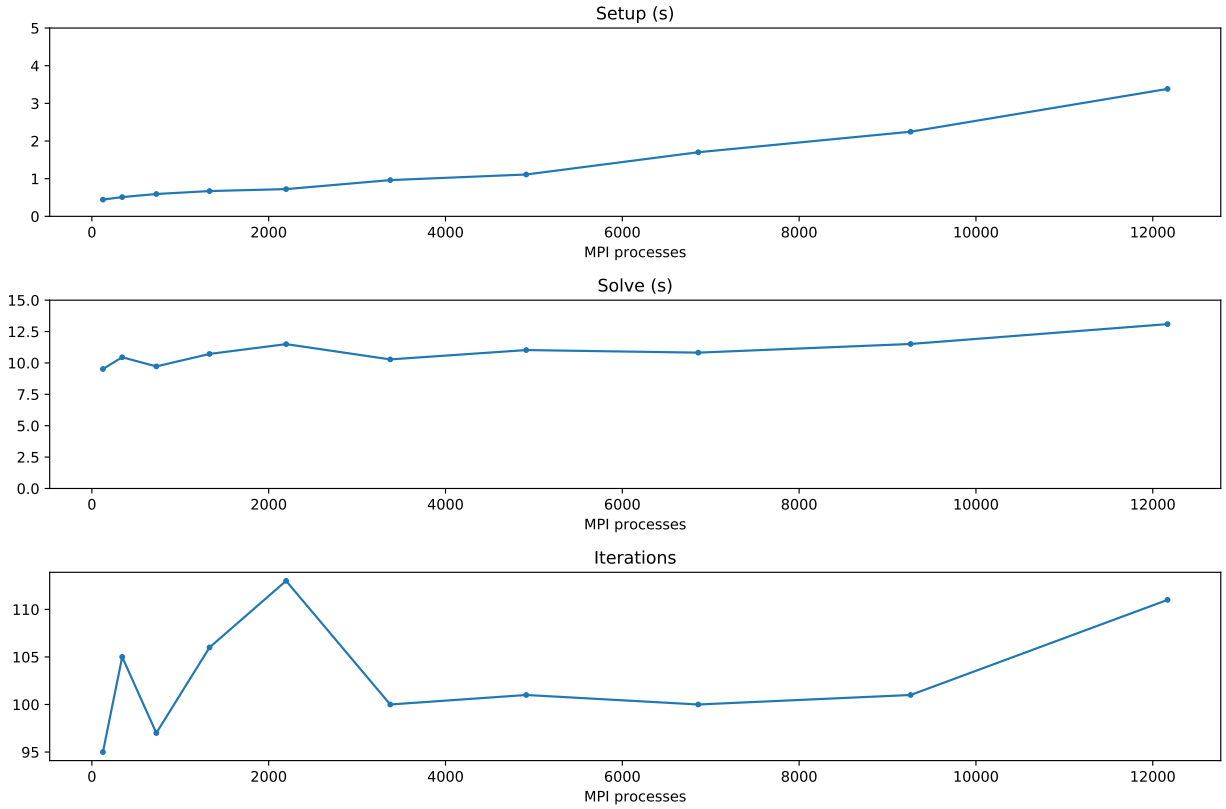
Essentially all of the recent efforts in this area take on the idea of constructing a multi level decomposition of the solution space, understood as a very aggressive coarsening of the original problem. The use of such coarser basis is proved sufficient to guarantee good weak scaling properties, while implying a minimal additional computational complexity and thus good strong scaling properties. An additional advantage of such approaches is the ease in combining them *in a modular way* with local preconditioners. AMGCL allows to combine the subdomain deflation approach with any of the shared-memory preconditioners implemented in the library, thus providing good weak and strong scalability properties.

The figures below demonstrate scalability of the subdomain deflation approach combined with AMG used as a local preconditioner. We solve the classical 3D Poisson problem in a unit cube $\Omega = [0, 1]^3$:

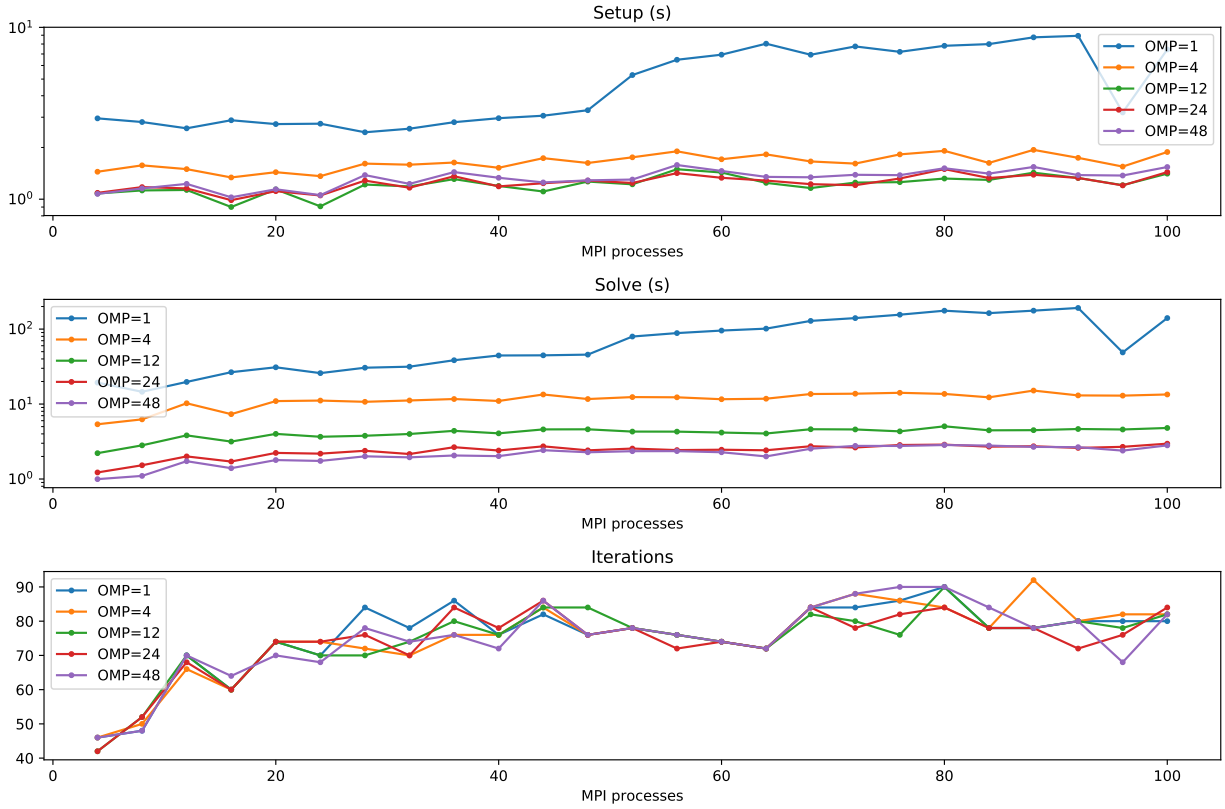
$$-\Delta u = 1,\ u \in \Omega \quad u = 0,\ u \in \partial\Omega$$

The first figure shows weak scaling for the problem in case of linear deflation Here the problem size grows proportionally with number of MPI processes used to solve the problem, so that a subdomain on each node corresponds to about 64^3 unknowns. The top two subplots show the setup and the solution cost of the algorithm (in wall-clock time, seconds). The bottom plot shows the number of iterations required to achieve convergence. It is clear from the figure that the setup time constitutes only a small fraction of the total cost of the method, and hence the most time is spent on actually solving the problem.

We can see that the scalability of the setup phase begins to noticeably suffer at about 5000 MPI processes. The main reason here is that the direct solver used to solve the coarse deflated problem becomes more and more expensive to setup. One possible solution to this problem would be to use more scalable direct solver. Another possibility is to reduce the size of the coarse problem by employing parallelism available within each of the compute nodes.



The next figure shows the results of the second approach. Here we study how the problem scales with increasing number of OpenMP threads withing each node, while allocating single MPI process per node. We can observe good scalability for up to 24 OpenMP threads, and even though going from 24 to 48 threads does not yield immediate improvement in solution time, it should allow us to gain advantage at extreme scale, where the size of the coarse system will be a limiting factor.



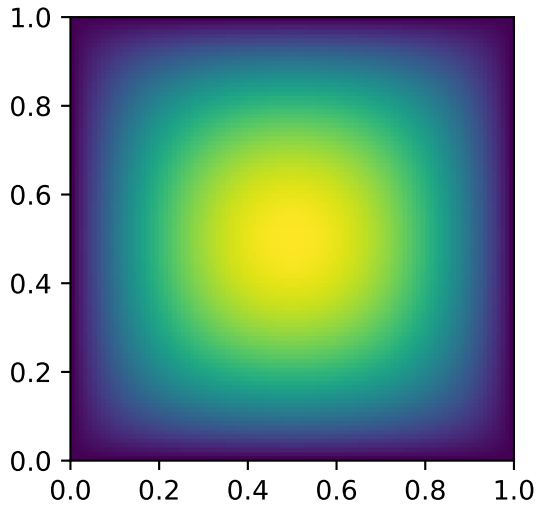
Examples

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 $\Omega = [0, 1] \times [0, 1]$:

$$-\Delta u = 1, \; u \in \Omega \quad u = 0, \; u \in \partial\Omega$$

The solution to the problem looks like this:



Here is how the problem may be discretized on a uniform $n \times n$ grid:

```
#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());
        }
    }

    return n2;
}
```

Adapting custom matrix class

This example shows how to adapt a custom matrix class for use with AMGCL solvers. Let's say the user application declares the following class to store its sparse matrices:

```
class sparse_matrix {
    public:
        typedef std::map<int, double> sparse_row;

        sparse_matrix(int n, int m) : _n(n), _m(m), _rows(n) { }

        int nrows()    const { return _n; }
        int ncols()    const { return _m; }
        int nonzeros() const {
            int nnz = 0;
            for(auto &row : _rows) nnz += row.size();
            return nns;
        }

        // Get a value at row i and column j
        double operator()(int i, int j) const {
            sparse_row::const_iterator elem = _rows[i].find(j);
            return elem == _rows[i].end() ? 0.0 : elem->second;
        }

        // Get reference to a value at row i and column j
        double& operator()(int i, int j) { return _rows[i][j]; }

        // Access the whole row
        const sparse_row& operator[](int i) const { return _rows[i]; }
    private:
        int _n, _m;
        std::vector<sparse_row> _rows;
```

```
};
```

Using `std::map` to store sparse rows is probably not the best idea performance-wise, but it may be convenient during assembly phase. Also, AMGCL will copy the matrix to internal structures during construction, so setup performance should not be affected by the choice of the input matrix type too much.

In order to make the above class work as input matrix for AMGCL, we have to specialize a few templates inside `amgcl::backend` namespace (the templates are declared inside <amgcl/backend/interface.hpp>). First we need to let AMGCL know the value type of the matrix:

```
namespace amgcl {
namespace backend {

template <> struct value_type<sparse_matrix> {
    typedef double type;
};
```

We also need to tell how to get the dimensions of the matrix and the number of its nonzero elements:

```
// Number of rows in the matrix
template<> struct rows_impl<sparse_matrix> {
    static int get(const sparse_matrix &A) { return A.nrows(); }
};

// Number of cols in the matrix
template<> struct cols_impl<sparse_matrix> {
    static int get(const sparse_matrix &A) { return A.ncols(); }
};

// Number of nonzeros in the matrix. This may be just a rough estimate.
template<> struct nonzeros_impl<sparse_matrix> {
    static int get(const sparse_matrix &A) { return A.nonzeros(); }
};
```

The last and the most involved part is providing a row iterator for the custom matrix type. In order to do this we need to define a `row_iterator<sparse_matrix>::type` class and specialize `row_begin_impl<sparse_matrix>` template that would return the iterator over the given row of the matrix. Here goes:

```
// Here we define row_iterator<sparse_matrix>::type
template<> struct row_iterator<sparse_matrix> {
    struct iterator {
        sparse_matrix::sparse_row::const_iterator _it, _end;

        // Take the matrix and the row number:
        iterator(const sparse_matrix &A, int row)
            : _it(A[row].begin()), _end(A[row].end()) { }

        // Check if the iterator is valid:
        operator bool() const {
            return _it != _end;
        }

        // Advance to the next nonzero element.
        iterator& operator++() {
            ++_it;
            return *this;
        }
```

```
        // Column number of the current nonzero element.
        int col() const { return _it->first; }

        // Value of the current nonzero element.
        double value() const { return _it->second; }
    };

    typedef iterator type;
};

// Provide a way to obtain the row iterator for the given matrix row:
template<> struct row_begin_impl<sparse_matrix> {
    typedef typename row_iterator<sparse_matrix>::type iterator;
    static iterator get(const sparse_matrix &A, int row) {
        return iterator(A, row);
    }
};

} // namespace backend
} // namespace amgcl
```

After this, we can directly use our matrix type to create an AMGCL solver:

```
// Discretize a 1D Poisson problem
const int n = 10000;

sparse_matrix A(n, n);
for(int i = 0; i < n; ++i) {
    if (i == 0 || i == n - 1) {
        // Dirichlet boundary condition
        A(i,i) = 1.0;
    } else {
        // Internal point.
        A(i, i-1) = -1.0;
        A(i, i)   =  2.0;
        A(i, i+1) = -1.0;
    }
}

// Create an AMGCL solver for the problem.
typedef amgcl::backend::builtin<double> Backend;

amgcl::make_solver<
    amgcl::amg<
        Backend,
        amgcl::coarsening::aggregation,
        amgcl::relaxation::spai0
        >,
    amgcl::solver::cg<Backend>
    > solve( A );
```

Note: The complete source code of the example may be found at examples/custom_adapter.cpp.

Bibliography

Indices and tables

- genindex
- search

Bibliography

[AnCD15] Anzt, Hartwig, Edmond Chow, and Jack Dongarra. "Iterative sparse triangular solves for preconditioning." European Conference on Parallel Processing. Springer Berlin Heidelberg, 2015.

[Barr94] Barrett, Richard, et al. Templates for the solution of linear systems: building blocks for iterative methods. Vol. 43. Siam, 1994.

[BaJM05] Baker, A. H., Jessup, E. R., & Manteuffel, T. (2005). A technique for accelerating the convergence of restarted GMRES. SIAM Journal on Matrix Analysis and Applications, 26(4), 962-984.

[BrGr02] Bröker, Oliver, and Marcus J. Grote. "Sparse approximate inverse smoothers for geometric and algebraic multigrid." Applied numerical mathematics 41.1 (2002): 61-80.

[CaGP73] Caretto, L. S., et al. "Two calculation procedures for steady, three-dimensional flows with recirculation." Proceedings of the third international conference on numerical methods in fluid mechanics. Springer Berlin Heidelberg, 1973.

[DeSh12] Demidov, D. E., and Shevchenko, D. V. "Modification of algebraic multigrid for effective GPGPU-based solution of nonstationary hydrodynamics problems." Journal of Computational Science 3.6 (2012): 460-462.

[FrVu01] Frank, Jason, and Cornelis Vuik. "On the construction of deflation-based preconditioners." SIAM Journal on Scientific Computing 23.2 (2001): 442-462.

[GiSo11] Van Gijzen, Martin B., and Peter Sonneveld. "Algorithm 913: An elegant IDR (s) variant that efficiently exploits biorthogonality properties." ACM Transactions on Mathematical Software (TOMS) 38.1 (2011): 5.

[Saad03] Saad, Yousef. Iterative methods for sparse linear systems. Siam, 2003.

[SaTu08] Sala, Marzio, and Raymond S. Tuminaro. "A new Petrov-Galerkin smoothed aggregation preconditioner for nonsymmetric linear systems." SIAM Journal on Scientific Computing 31.1 (2008): 143-166.

[SlDi93] Sleijpen, Gerard LG, and Diederik R. Fokkema. "BiCGstab (l) for linear equations involving unsymmetric matrices with complex spectrum." Electronic Transactions on Numerical Analysis 1.11 (1993): 2000.

[Stue99] Stüben, Klaus. Algebraic multigrid (AMG): an introduction with applications. GMD-Forschungszentrum Informationstechnik, 1999.

[Stue07] Stüben, Klaus, et al. "Algebraic multigrid methods (AMG) for the efficient solution of fully implicit formulations in reservoir simulation." SPE Reservoir Simulation Symposium. Society of Petroleum Engineers, 2007.

[TrOS01] Trottenberg, U., Oosterlee, C., and Schüller, A. Multigrid. Academic Press, London, 2001.

Index

A

O