AMGCL Documentation

Release 1.2.0.post189

Denis Demidov

Nov 13, 2018

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:

1.1 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*.

1.1.1 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( 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:

```
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.

1.1.2 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( std::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.

1.1.3 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 *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 );
```

1.1.4 The `make_solver` class

template <**class** Precond, **class** IterativeSolver>
class **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>
std::tuple<size_t, scalar_type> **operator()**(**const** Matrix &*A*, **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 `std::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>
std::tuple<size_t, scalar_type> **operator()**(**const** Vec1 &*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 `std::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.

std::shared_ptr<**typename** Precond::matrix> **system_matrix_ptr**() **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.

1.2 Components

1.2.1 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 `std::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:

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

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

1.2.2 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;
```

(continues on next page)

(continued from previous page)

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

Builtin

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

template <typename *ValueType*>
struct builtin

The builtin backend does not have any dependencies, 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.

VexCL

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

template <**typename** real, **class** DirectSolver = solver::vexcl_skyline_lu<real>>
struct 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.

1.2.3 Preconditioners

1.2.4 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()`:

std::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 `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]*.

std::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 `std::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 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 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<>
preconditioner::side::type **pside**

Preconditioning kind (left/right).

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 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.

GMRES

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

template <**class** Backend, **class** InnerProduct = detail::default_inner_product>
class 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<>
preconditioner::side::type **pside**

Preconditioning kind (left/right).

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 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<>
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<>
preconditioner::side::type **pside**
Preconditioning kind (left/right).

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 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.

1.3 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);
```

1.3.1 Classes

AMG preconditioner

Warning: doxygenclass: Cannot find class "amgcl::runtime::amg" in doxygen xml output for project "AMGCL" from directory: xml

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

Warning: doxygenclass: Cannot find class "amgcl::runtime::iterative_solver" in doxygen xml output for project "AMGCL" from directory: xml

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)

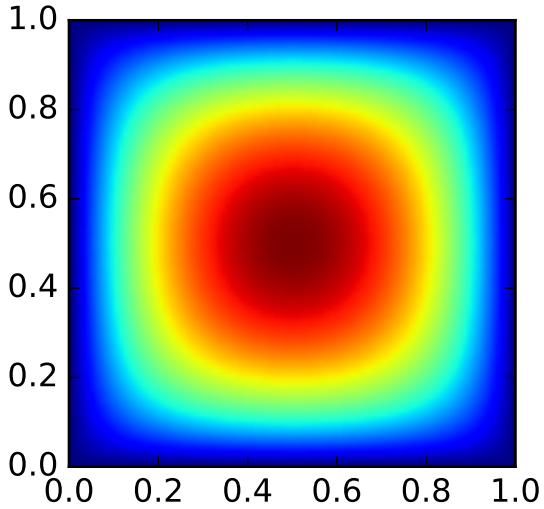
1.4 Examples

1.4.1 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:

```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());
        }
    }

    return n2;
}
```

1.4.2 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
```

(continues on next page)

(continued from previous page)

```
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.

1.5 Benchmarks

The performance of the library on shared and distributed memory systems was tested on two example problems in a three dimensional space: simple scalar Poisson problem and a non-scalar Navier-Stokes problem. The source code for the benchmarks is available at https://github.com/ddemidov/amgcl_benchmarks.

The first example we consider 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 dicretized 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

$$\begin{pmatrix} \mathbf{K} & \mathbf{G} \\ \mathbf{D} & \mathbf{S} \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \mathbf{p} \end{pmatrix} = \begin{pmatrix} \mathbf{b}_u \\ \mathbf{b}_p \end{pmatrix}$$

where each of the matrix subblocks is a large sparse matrix, and the blocks $\mathbf{G}$ and $\mathbf{D}$ are non-square. The overall system matrix for the problem was assembled in the Kratos multi-physics package developed in CIMNE, Barcelona.

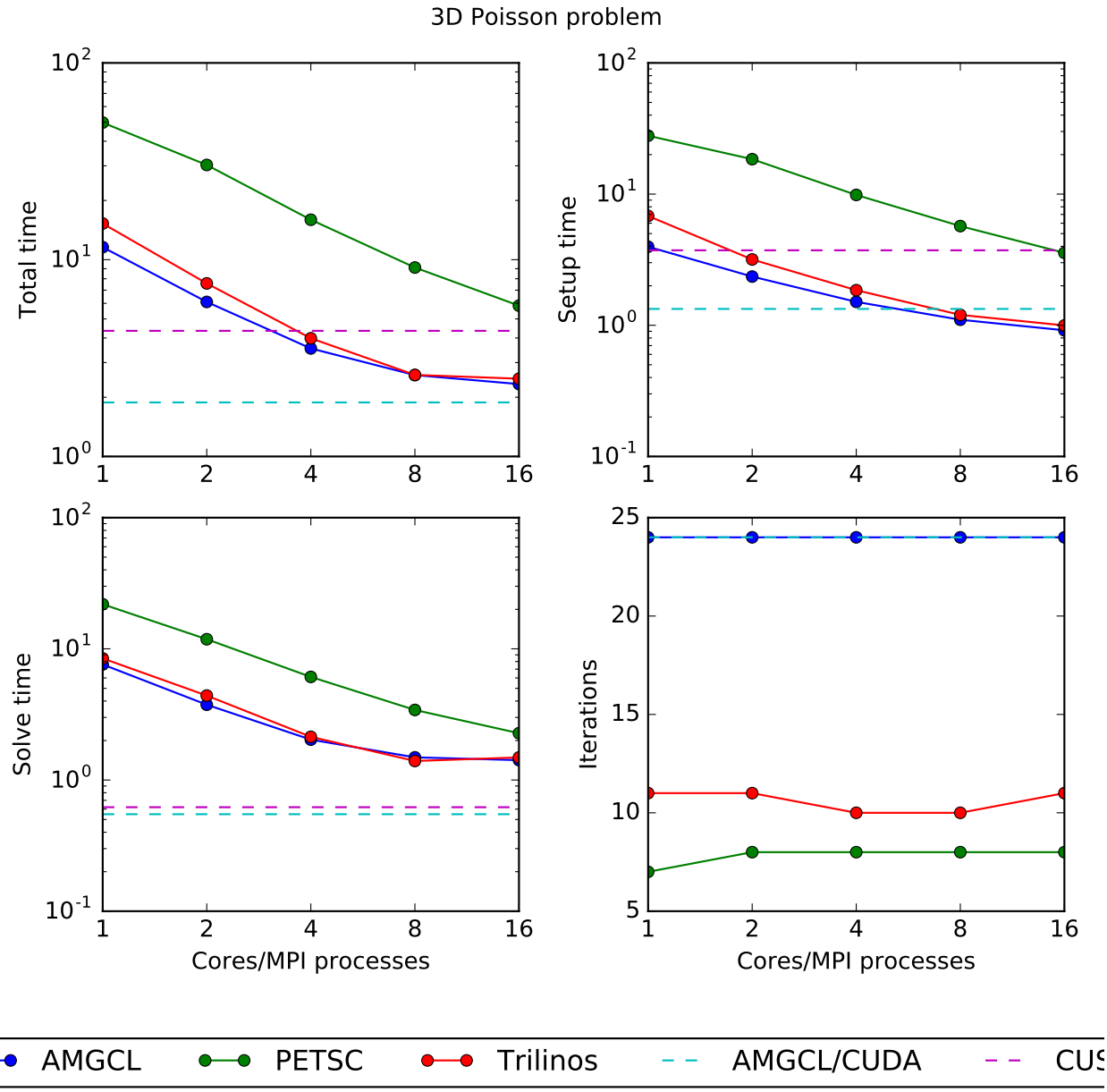
1.5.1 Shared memory benchmarks

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 dicretized 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.

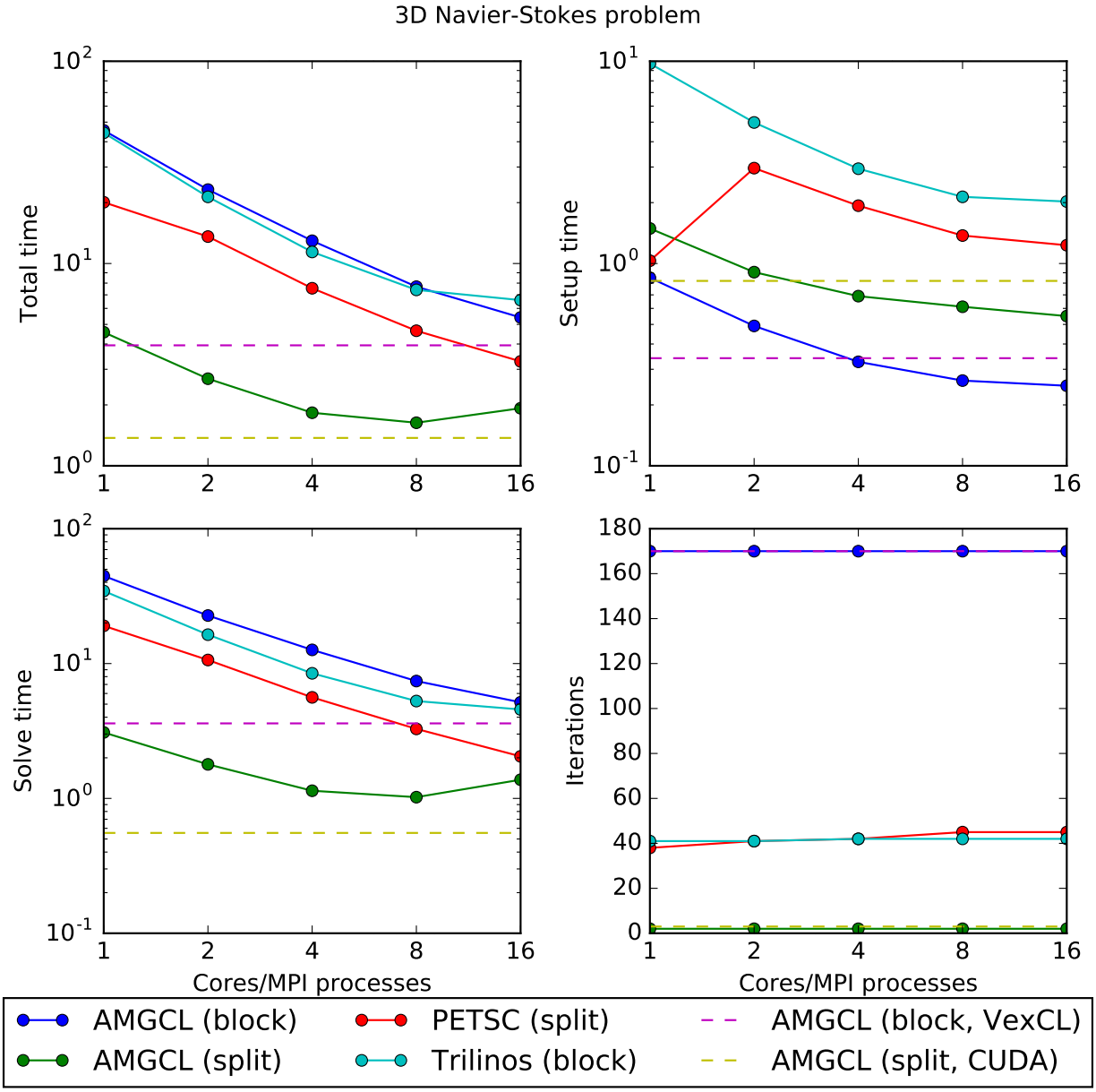
On the GPU, AMGCL performs slightly better than CUSP. If we consider the solution time (without setup), then both libraries are able to outperform CPU-based versions by a factor of 3-4. The total solution time of AMGCL with CUDA backend is only 30% better than that of either AMGCL with OpenMP backend 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.

There are at least two ways to solve the system. First, one can treat the system as a monolythic 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 monolythic 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.

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 monolythic system, and 'split' results correspond to the field-split approach.



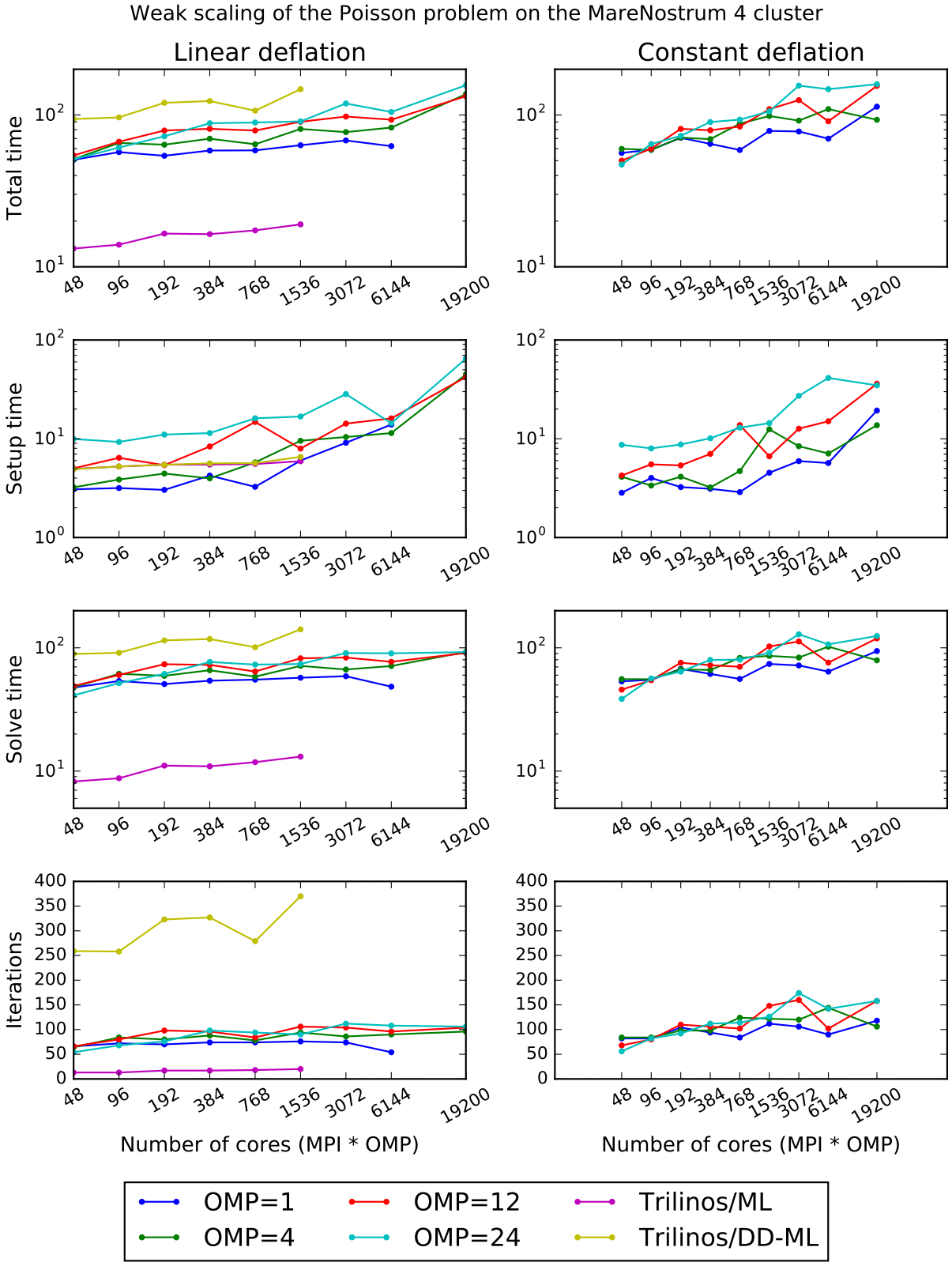
1.5.2 Distributed memory benchmarks

Support for distributed memory systems in AMGCL is implemented using the subdomin deflation method. Here we demonstrate performance and scalability of the approach 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-known Trilinos ML package. The benchmarks were run on MareNostrum 4 and PizDaint 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.

3D Poisson problem

The AMGCL implementation uses a BiCGStab(2) iterative solver preconditioned with subdomain deflation, as it 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:

Cores	Setup		Solve	Iterations
	Total	Factorize E		
Linear deflation, OMP=1				
384	4.23	0.02	54.08	74
1536	6.01	0.64	57.19	76
6144	13.92	8.41	48.40	54
Constant deflation, OMP=1				
384	3.11	0.00	61.41	94
1536	4.52	0.01	73.98	112
6144	5.67	0.16	64.13	90
Linear deflation, OMP=12				
384	8.35	0.00	72.68	96
1536	7.95	0.00	82.22	106
6144	16.08	0.03	77.00	96
19200	42.09	1.76	90.74	104
Constant deflation, OMP=12				
384	7.02	0.00	72.25	106
1536	6.64	0.00	102.53	148
6144	15.02	0.00	75.82	102
19200	36.08	0.03	119.25	158

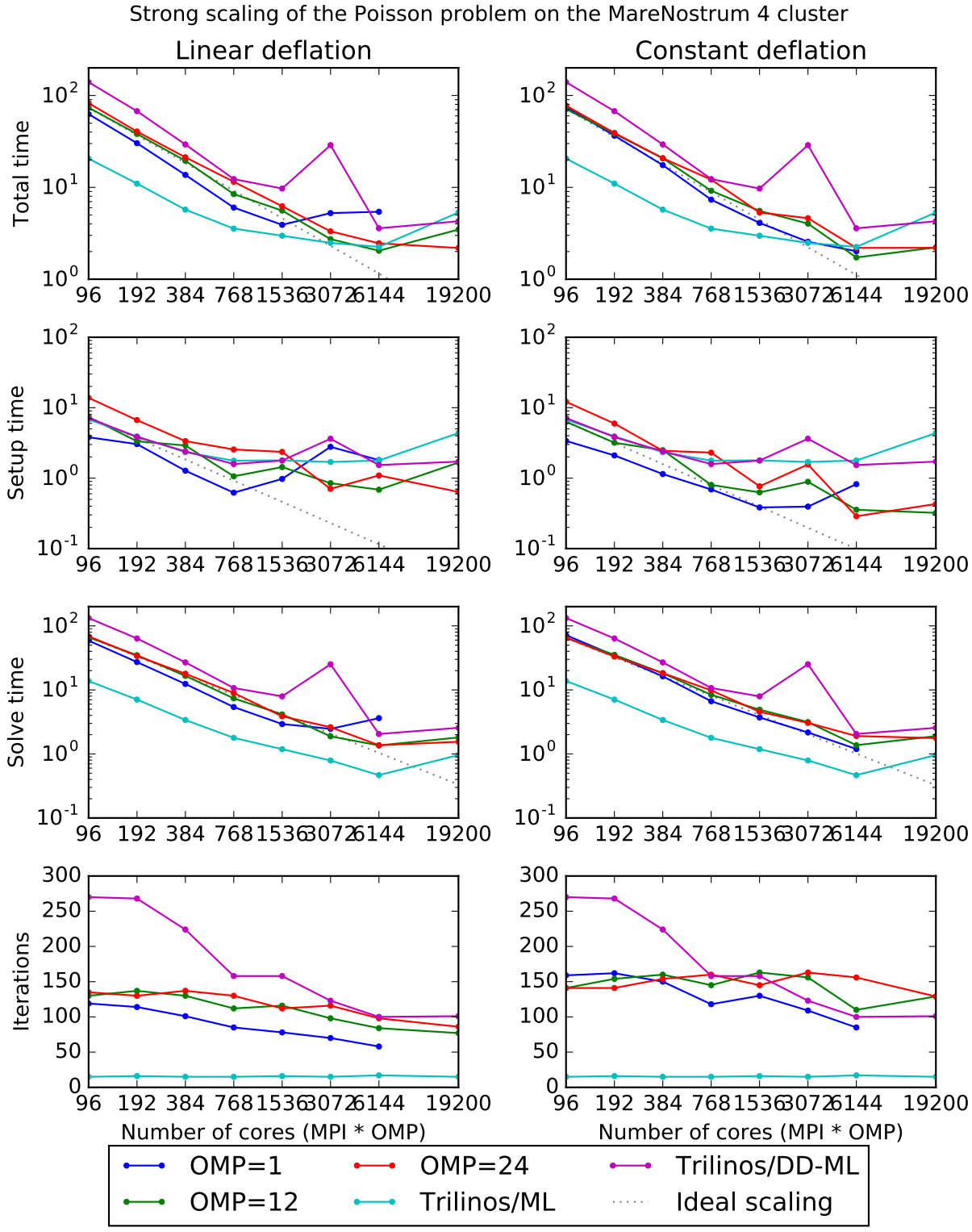
The table presents the profiling data for the solution of the Poisson problem on the MareNostrum 4 cluster. The first two columns show time spent on the setup of the preconditioner and the solution of the problem; the third column shows the number of iterations required for convergence. The 'Setup' column 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 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.

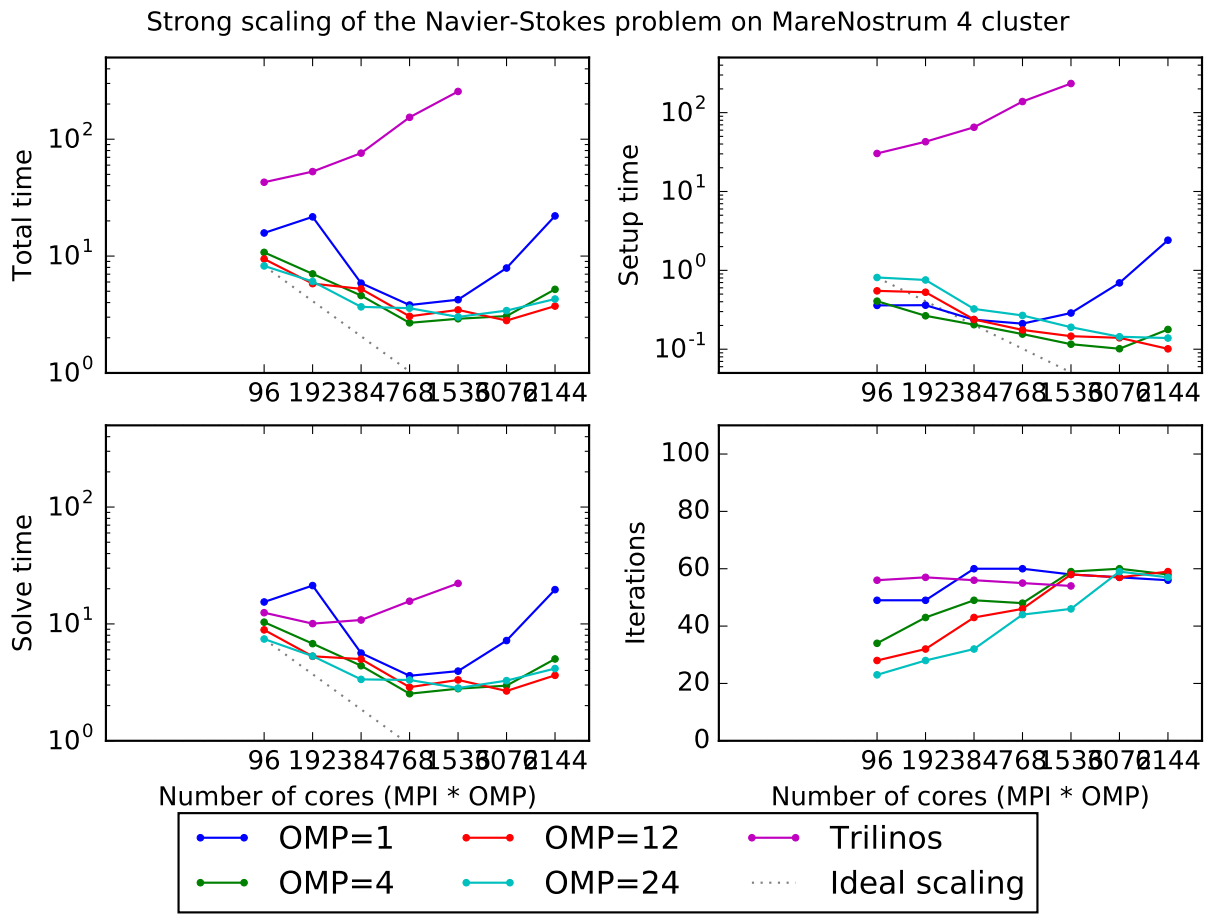
Cores	Setup		Solve	Iterations
	Total	Factorize E		
Linear deflation, OMP=1				
384	1.27	0.02	12.39	101
1536	0.97	0.45	2.93	78
6144	9.09	8.44	3.61	58
Constant deflation, OMP=1				
384	1.14	0.00	16.30	150
1536	0.38	0.01	3.71	130
6144	0.82	0.16	1.19	85
Linear deflation, OMP=12				
384	2.90	0.00	16.57	130
1536	1.43	0.00	4.15	116
6144	0.68	0.03	1.35	84
19200	1.66	1.29	1.80	77
Constant deflation, OMP=12				
384	2.49	0.00	18.25	160
1536	0.62	0.00	4.91	163
6144	0.35	0.00	1.37	110
19200	0.32	0.02	1.89	129

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'.

3D Navier-Stokes problem

The system matrix in these tests contains 4773588 unknowns and 281089456 nonzeros. 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 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 context the use of OpenMP within each domain pays off and allows delivering a greater level of scalability.

1.6 Bibliography

1.7 Indices and tables

- genindex
- search

Bibliography

[Adam98] Adams, Mark. “A parallel maximal independent set algorithm”, in Proceedings 5th copper mountain conference on iterative methods, 1998.

[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.

[Fokk96] Fokkema, Diederik R. “Enhanced implementation of BiCGstab (l) for solving linear systems of equations.” Universiteit Utrecht. Mathematisch Instituut, 1996.

[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.

[VaMB96] Vaněk, Petr, Jan Mandel, and Marian Brezina. “Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems.” Computing 56.3 (1996): 179-196.

Index

A

O