AMGCL Documentation

Release 1.4.0

Denis Demidov

Nov 26, 2020

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 [BrMH85], [Stue99], [TrOS01]. The method can be used as a black-box solver for various computational problems, since it does not require any information about the underlying geometry. AMG is often used not as a standalone solver but as a preconditioner within an iterative solver (e.g. Conjugate Gradients, BiCGStab, or GMRES).

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

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

CHAPTER 1

Referencing

D. Demidov. AMGCL: An efficient, flexible, and extensible algebraic multigrid implementation. Lobachevskii Journal of Mathematics, 40(5):535–546, May 2019. doi1 pdf1

```
@Article{Demidov2019,
    author="Demidov, D.",
    title="AMGCL: An Efficient, Flexible, and Extensible Algebraic Multigrid␣
↪Implementation",
    journal="Lobachevskii Journal of Mathematics",
    year="2019",
    month="May",
    day="01",
    volume="40",
    number="5",
    pages="535--546",
    issn="1818-9962",
    doi="10.1134/S1995080219050056",
    url="https://doi.org/10.1134/S1995080219050056"
}
```

D. Demidov. AMGCL – A C++ library for efficient solution of large sparse linear systems. Software Impacts, 6:100037, November 2020. doi2

```
@article{Demidov2020,
    author = "Denis Demidov",
    title = "AMGCL -- A C++ library for efficient solution of large sparse linear␣
↪systems",
    journal = "Software Impacts",
    volume = "6",
    pages = "100037",
    year = "2020",
    issn = "2665-9638",
    doi = "10.1016/j.simpa.2020.100037",
    url = "https://doi.org/10.1016/j.simpa.2020.100037"
}
```

CHAPTER 2

Contents:

2.1 Algebraic Multigrid

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

$$Au = f$$

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

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

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

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

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

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

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

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

  - Solve the corasest system directly: $u_L = A_L^{-1} f_L$.
  - **At each level of the grid hiearchy, coarsest-to-finest:**
    1. Update the current solution with the interpolated solution from the coarser level: $u_i = u_i + P_i u_{i+1}$.
    2. Apply a couple of smoothing iterations (*post-relaxation*) to the updated solution: $u_i = S_i(A_i, f_i, u_i)$.

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

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

2.2 Design Principles

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

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

- *Policy-based design* [Alex00] of public library classes such as `amgcl::make_solver` or `amgcl::amg` allows the library users to compose their own customized version of the iterative solver and preconditioner from the provided components and easily extend and customize the library by providing their own implementation of the algorithms.
- Preference for *free functions* as opposed to member functions [Meye05], combined with *partial template specialization* allows to extend the library operations onto user-defined datatypes and to introduce new algorithmic components when required.
- The *backend* system of the library allows expressing the algorithms such as Krylov iterative solvers or multigrid relaxation methods in terms of generic parallel primitives which facilitates transparent acceleration of the solution phase with OpenMP, OpenCL, or CUDA technologies.
- One level below the backends are *value types*: AMGCL supports systems with scalar, complex, or block value types both in single and double precision. Arithmetic operations necessary for the library work may also be extended onto the user-defined types using template specialization.

2.2.1 Policy-based design

Listing 2.1: Policy-based design illustration: creating customized solvers from AMGCL components

```
 1  // CG solver preconditioned with ILU0
 2  typedef amgcl::make_solver<
 3      amgcl::relaxation::as_preconditioner<
 4          amgcl::backend::builtin<double>,
 5          amgcl::relaxation::ilu0
 6          >,
 7      amgcl::solver::cg<
 8          amgcl::backend::builtin<double>
 9          >
10      > Solver1;
11
12  // GMRES solver preconditioned with AMG
13  typedef amgcl::make_solver<
14      amgcl::amg<
15          amgcl::backend::builtin<double>,
16          amgcl::coarsening::smoothed_aggregation,
17          amgcl::relaxation::spai0
18          >,
19      amgcl::solver::gmres<
20          amgcl::backend::builtin<double>
21          >
22      > Solver2;
```

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

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

Listing 2.2: Example of parameter declaration in AMGCL components

```
template <class P, class S>
struct make_solver {
    struct params {
        typename P::params precond;
        typename S::params solver;
    };
};
```

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

Listing 2.3: Setting parameters for AMGCL components

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

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

2.2.2 Free functions and partial template specialization

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

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

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

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

// Specialization of rows_impl template for a CRS tuple.
template < typename N, typename PRng, typename CRng, typename VRng >
```

(continues on next page)

(continued from previous page)

```
struct rows_impl< std::tuple<N, PRng, CRng, VRng> >
{
    static size_t get(const std::tuple<N, PRng, CRng, VRng> &A) {
        return std::get<0>(A);
    }
};
```

2.2.3 Backends

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

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

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

2.2.4 Value types

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

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

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

2.2.5 Runtime interface

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

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

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

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

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

boost::property_tree::ptree prm;

prm.put("solver.type", "bicgstab");
prm.put("solver.tol", 1e-3);
prm.put("solver.maxiter", 10);
prm.put("precond.coarsening.type", "smoothed_aggregation");
prm.put("precond.relax.type", "spai0");

Solver solve( std::tie(n, ptr, col, val), prm );
```

2.3 Tutorial

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

2.3.1 Poisson problem

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

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

Note: The examples/solver is convenient not only for testing the systems obtained elsewhere. You can also save your

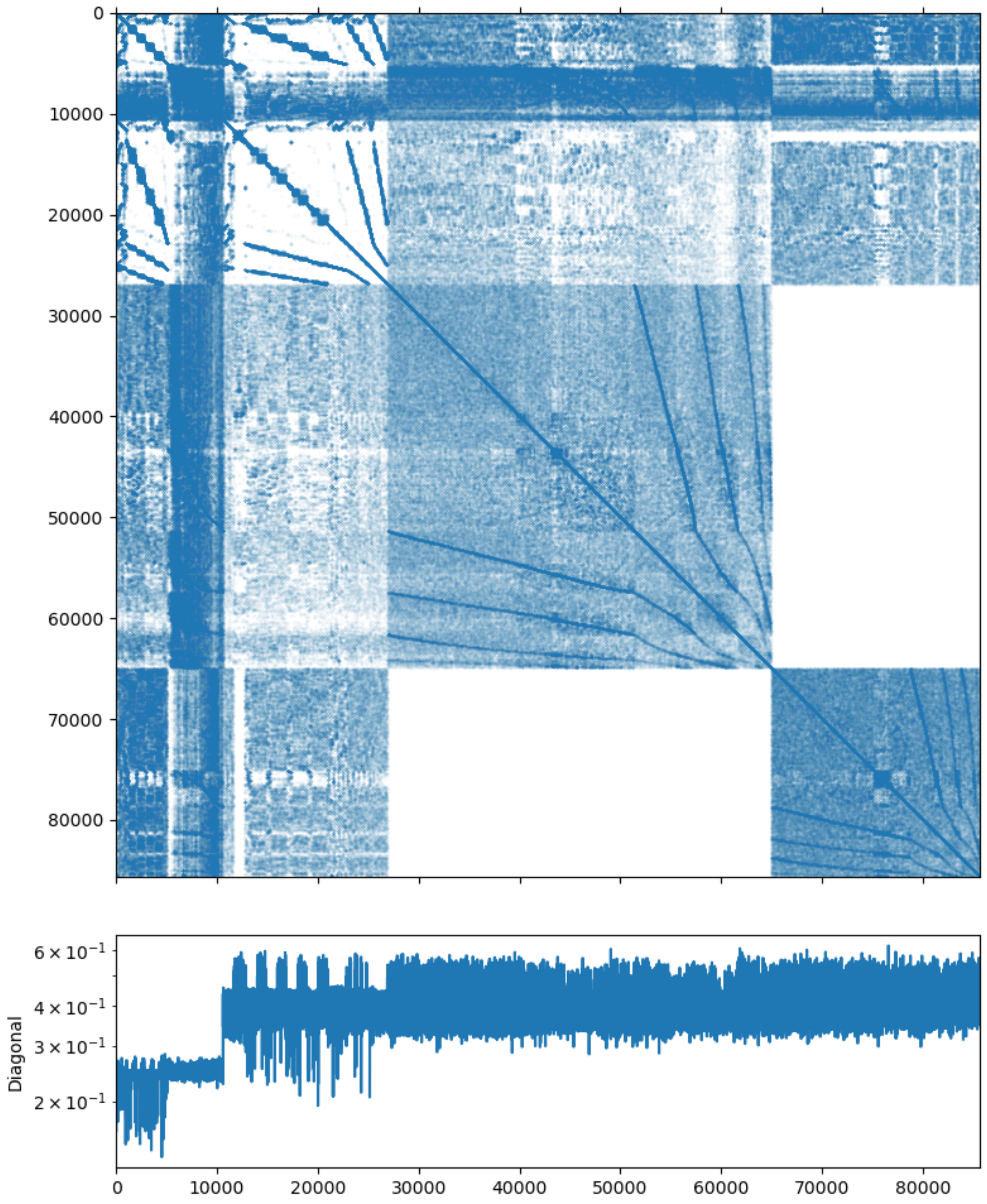


Fig. 2.1: Poisson3Db matrix portrait

own matrix and the RHS vector into the Matrix Market format with `amgcl::io::mm_write()` function. This way you can find the AMGCL options that work for your problem without the need to rewrite the code and recompile the program.

The default options of BiCGStab iterative solver preconditioned with a smoothed aggregation AMG (a simple diagonal SPAI(0) relaxation is used on each level of the AMG hierarchy) should work very well for a Poisson problem:

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

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

level     unknowns       nonzeros      memory
---------------------------------------------
    0        85623        2374949     50.07 M (83.20%)
    1         6361         446833      7.78 M (15.65%)
    2          384          32566      1.08 M ( 1.14%)

Iterations: 24
Error:      8.33789e-09

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

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

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

Preconditioner
==============
Number of levels:    3
Operator complexity: 1.20
Grid complexity:     1.08
```

(continues on next page)

(continued from previous page)

```
Memory footprint:    58.93 M

level     unknowns       nonzeros      memory
---------------------------------------------
    0        85623        2374949     50.07 M (83.20%)
    1         6361         446833      7.78 M (15.65%)
    2          384          32566      1.08 M ( 1.14%)

Iterations: 39
Error:      9.50121e-09

[Profile:       2.282 s] (100.00%)
[  reading:     1.612 s] ( 70.66%)
[  setup:       0.135 s] (  5.93%)
[  solve:       0.533 s] ( 23.38%)
```

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

```
$ solver -A poisson3Db.mtx -f poisson3Db_b.mtx precond.relax.type=ilu0
Solver
======
Type:             BiCGStab
Unknowns:         85623
Memory footprint: 4.57 M

Preconditioner
==============
Number of levels:    3
Operator complexity: 1.20
Grid complexity:     1.08
Memory footprint:    103.44 M

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

Iterations: 12
Error:      7.99207e-09

[Profile:       2.510 s] (100.00%)
[ self:         0.005 s] (  0.19%)
[  reading:     1.614 s] ( 64.30%)
[  setup:       0.464 s] ( 18.51%)
[  solve:       0.427 s] ( 17.01%)
```

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

```
$ solver -A poisson3Db.mtx -f poisson3Db_b.mtx precond.relax.type=chebyshev
Solver
======
Type:             BiCGStab
```

(continues on next page)

(continued from previous page)

```
Unknowns:          85623
Memory footprint: 4.57 M

Preconditioner
==============
Number of levels:    3
Operator complexity: 1.20
Grid complexity:     1.08
Memory footprint:    59.63 M

level     unknowns       nonzeros      memory
---------------------------------------------
    0        85623        2374949     50.72 M (83.20%)
    1         6361         446833      7.83 M (15.65%)
    2          384          32566      1.08 M ( 1.14%)

Iterations: 8
Error:      5.21588e-09

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
    std::vector<double> val, rhs;

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

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

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

    // Compose the solver type
    //   the solver backend:
    typedef amgcl::backend::builtin<double> SBackend;
    //   the preconditioner backend:
#ifdef MIXED_PRECISION
    typedef amgcl::backend::builtin<float> PBackend;
#else
    typedef amgcl::backend::builtin<double> PBackend;
#endif

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

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

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

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

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

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

In lines 4–10 we include the necessary AMGCL headers: the builtin backend uses the OpenMP threading model; the `crs_tuple` matrix adaper allows to use a `std::tuple` of CRS arrays as an input matrix; the `amgcl::make_solver` class binds together a preconditioner and an iterative solver; `amgcl::amg` class is the AMG preconditioner; `amgcl::coarsening::smoothed_aggregation` defines the smoothed aggreation coarsening strategy; `amgcl::relaxation::spai0` is the sparse approximate inverse relaxation used on each level of the AMG hierarchy; and `amgcl::solver::bicgstab` is the BiCGStab iterative solver. In lines 12–13 we include the Matrix Market reader and the AMGCL profiler.

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

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

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

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

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

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

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

level     unknowns       nonzeros      memory
---------------------------------------------
    0        85623        2374949     50.07 M (83.20%)
    1         6361         446833      7.78 M (15.65%)
    2          384          32566      1.08 M ( 1.14%)

Iters: 24
Error: 8.33789e-09

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

Looking at the output of the mixed precision version, it is apparent that it uses less memory for the preconditioner

(43.59M as opposed to 58.93M in the double-precision case), and is slightly faster during both the setup and the solution phases:

```
$ ./poisson3Db_mixed poisson3Db.mtx poisson3Db_b.mtx
Matrix poisson3Db.mtx: 85623x85623
RHS poisson3Db_b.mtx: 85623x1
Solver
======
Type:             BiCGStab
Unknowns:         85623
Memory footprint: 4.57 M

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

level     unknowns       nonzeros      memory
---------------------------------------------
    0        85623        2374949     37.23 M (83.20%)
    1         6361         446833      5.81 M (15.65%)
    2          384          32566    554.90 K ( 1.14%)

Iters: 24
Error: 7.33493e-09

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

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

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

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

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

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

```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

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

    // Initialize the solver with the system matrix:
```

(continues on next page)

(continued from previous page)

```
    prof.tic("setup");
    Solver solve(A, prm, bprm);
    prof.toc("setup");

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

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

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

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

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

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

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

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

Iters: 24
Error: 8.33789e-09

```

(continues on next page)

(continued from previous page)

```
[poisson3Db:     2.253 s] (100.00%)
[ self:          0.223 s] (  9.90%)
[  read:         1.676 s] ( 74.39%)
[  setup:        0.183 s] (  8.12%)
[  solve:        0.171 s] (  7.59%)
```

2.3.2 Poisson problem (MPI version)

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

Listing 2.7: The MPI solution of the poisson3Db problem

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

#include <amgcl/backend/builtin.hpp>
#include <amgcl/adapter/crs_tuple.hpp>

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

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

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

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

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

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

    // Read the system matrix and the RHS:
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
95
96          // Reorder the matrix:
97          A = product(*R, *product(*A, *P));
98
99          // and the RHS vector:
100         std::vector<double> new_rhs(R->loc_rows());
101         R->move_to_backend(typename DBackend::params());
102         amgcl::backend::spmv(1, *R, rhs, 0, new_rhs);
103         rhs.swap(new_rhs);
104
105         // Update the number of the local rows
106         // (it may have changed as a result of permutation):
107         chunk = A->loc_rows();
108         prof.toc("partition");
109     }
110 #endif
111
112     // Initialize the solver:
113     prof.tic("setup");
114     Solver solve(world, A);
115     prof.toc("setup");
116
117     // Show the mini-report on the constructed solver:
118     if (world.rank == 0)
119         std::cout << solve << std::endl;
120
121     // Solve the system with the zero initial approximation:
122     int iters;
123     double error;
124     std::vector<double> x(chunk, 0.0);
125
126     prof.tic("solve");
127     std::tie(iters, error) = solve(*A, rhs, x);
128     prof.toc("solve");
129
130     // Output the number of iterations, the relative error,
131     // and the profiling data:
132     if (world.rank == 0)
133         std::cout
134             << "Iters: " << iters << std::endl
135             << "Error: " << error << std::endl
136             << prof << std::endl;
137 }
```

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

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

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

The `amgcl::mpi::init` is a convenience RAII wrapper for `MPI_Init()`. It will call `MPI_Finalize()` in the destructor when its instance (`mpi`) goes out of scope at the end of the program. We don't have to use the wrapper, but it simply makes things easier. `amgcl::mpi::communicator` is an equally thin wrapper for `MPI_Comm`. `amgcl::mpi::communicator` and `MPI_Comm` may be used interchangeably both with the AMGCL MPI interface and the native MPI functions.

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

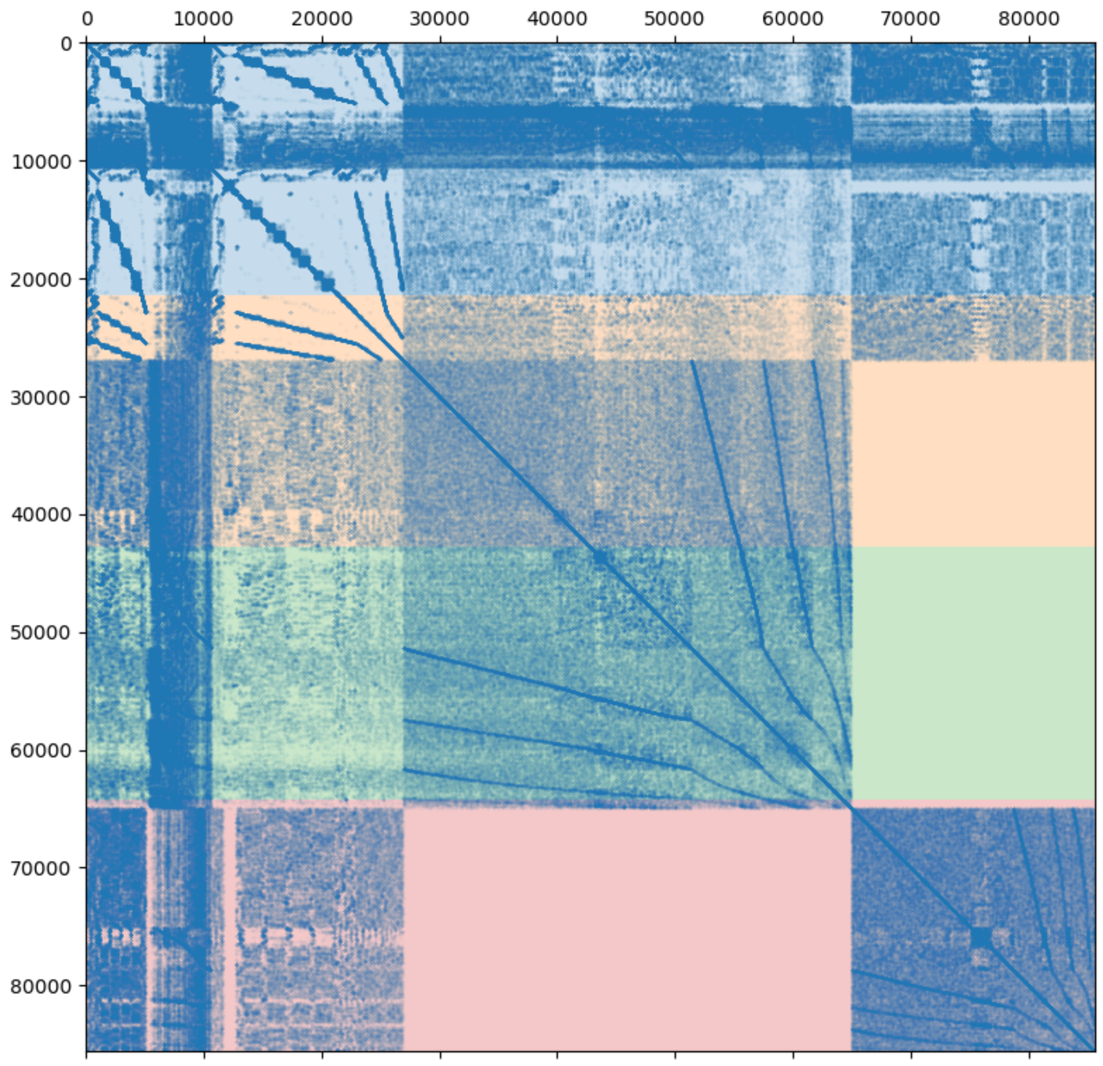


Fig. 2.2: Poisson3Db matrix partitioned between the 4 MPI processes

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

The figure shows the finite-diffrence discretization of a 2D Poisson problem on a 4×4 grid in a unit square. The nonzero pattern of the system matrix is presented on the lower left plot. If the grid nodes are numbered row-wise, then

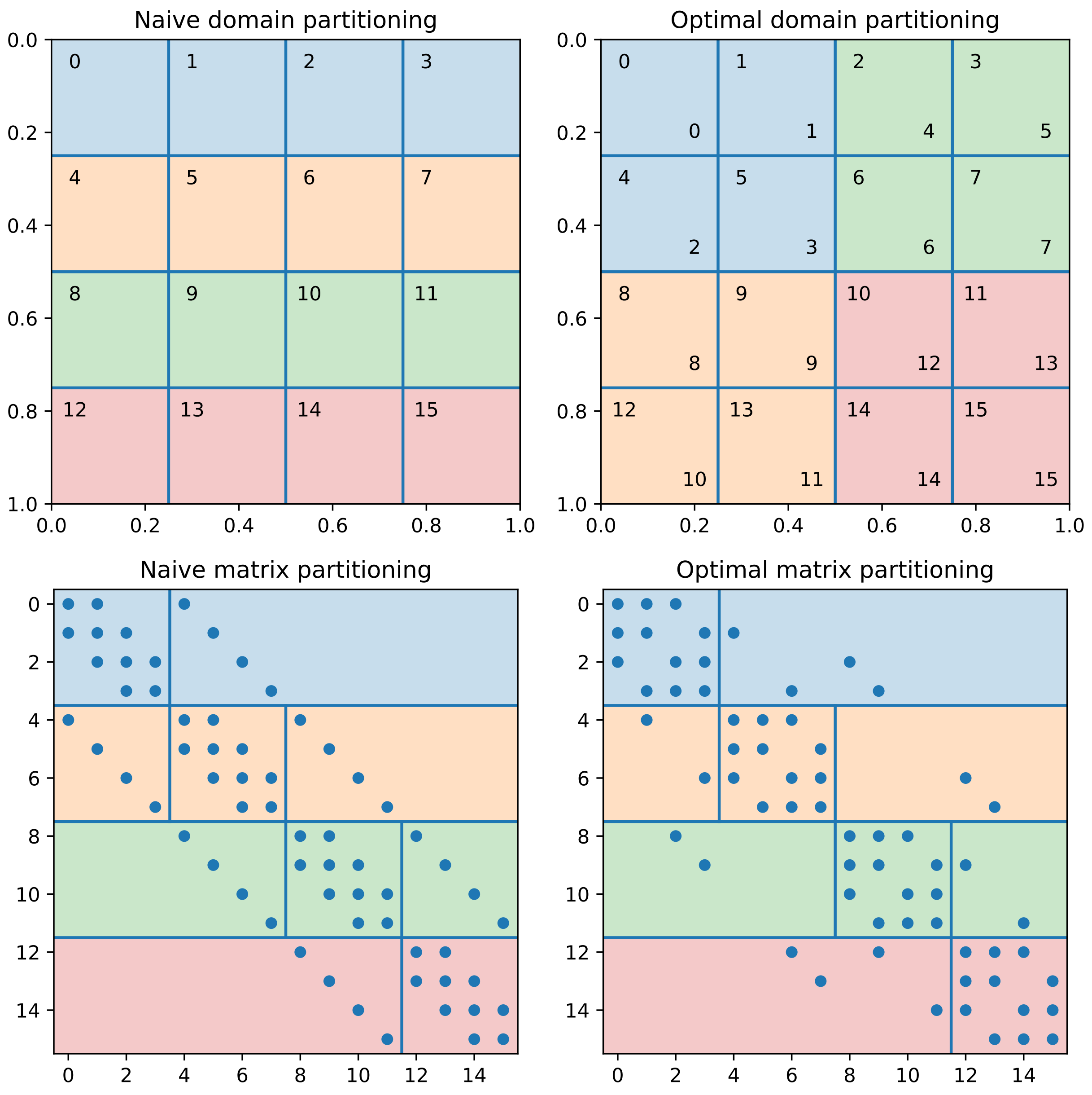

Naive domain partitioning
Optimal domain partitioning
Naive matrix partitioning
Optimal matrix partitioning

Fig. 2.3: Naive vs optimal partitioning of a 4×4 grid between 4 MPI processes.

the naive partitioning of the system matrix for the 4 MPI processes is shown on the upper left plot. The subdomains belonging to each of the MPI processes correspond to the continuous ranges of grid node indices and are elongated along the X axis. This results in high MPI communication traffic, as the number of the interface nodes is high relative to the number of interior nodes. The upper right plot shows the optimal partitioning of the domain for the 4 MPI processes. In order to keep the rows owned by a single MPI process adjacent to each other (so that each MPI process owns a continuous range of rows, as required by AMGCL), the grid nodes have to be renumbered. The labels in the top left corner of each grid node show the original numbering, and the lower-rigth labels show the new numbering. The renumbering of the matrix may be expressed as the permutation matrix P, where $P_{ij} = 1$ if the j-th unknown in the original ordering is mapped to the i-th unknown in the new ordering. The reordered system may be written as

$$P^T APy = P^T f$$

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

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

```
37      // Read the system matrix and the RHS:
38      prof.tic("read");
39      // Get the global size of the matrix:
40      ptrdiff_t rows = amgcl::io::crs_size<ptrdiff_t>(argv[1]);
41      ptrdiff_t cols;
42
43      // Split the matrix into approximately equal chunks of rows
44      ptrdiff_t chunk = (rows + world.size - 1) / world.size;
45      ptrdiff_t row_beg = std::min(rows, chunk * world.rank);
46      ptrdiff_t row_end = std::min(rows, row_beg + chunk);
47      chunk = row_end - row_beg;
48
49      // Read our part of the system matrix and the RHS.
50      std::vector<ptrdiff_t> ptr, col;
51      std::vector<double> val, rhs;
52      amgcl::io::read_crs(argv[1], rows, ptr, col, val, row_beg, row_end);
53      amgcl::io::read_dense(argv[2], rows, cols, rhs, row_beg, row_end);
54      prof.toc("read");
```

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

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

```
62      // Compose the solver type
63      typedef amgcl::backend::builtin<double> DBackend;
64      typedef amgcl::backend::builtin<float>  FBackend;
65      typedef amgcl::mpi::make_solver<
66          amgcl::mpi::amg<
67              FBackend,
68              amgcl::mpi::coarsening::smoothed_aggregation<FBackend>,
69              amgcl::mpi::relaxation::spai0<FBackend>
70              >,
71          amgcl::mpi::solver::bicgstab<DBackend>
72          > Solver;
```

The structure of the solver is the same as in the shared memory case in the *Poisson problem* tutorial, but we are using the components from the `amgcl::mpi` namespace. Again, we are using the mixed-precision approach and the preconditioner backend is defined with a single-precision value type.

In lines 74–76 we create the distributed matrix from the local strips read by each of the MPI processes:

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

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

```
78      // Partition the matrix and the RHS vector.
79      // If neither ParMETIS not PT-SCOTCH are not available,
80      // just keep the current naive partitioning.
81  #if defined(AMGCL_HAVE_PARMETIS) || defined(AMGCL_HAVE_SCOTCH)
82  #   if defined(AMGCL_HAVE_PARMETIS)
83      typedef amgcl::mpi::partition::parmetis<DBackend> Partition;
84  #   elif defined(AMGCL_HAVE_SCOTCH)
85      typedef amgcl::mpi::partition::ptscotch<DBackend> Partition;
86  #   endif
87
88      if (world.size > 1) {
89          prof.tic("partition");
90          Partition part;
91
92          // part(A) returns the distributed permutation matrix:
93          auto P = part(*A);
94          auto R = transpose(*P);
95
96          // Reorder the matrix:
97          A = product(*R, *product(*A, *P));
98
99          // and the RHS vector:
100         std::vector<double> new_rhs(R->loc_rows());
101         R->move_to_backend(typename DBackend::params());
102         amgcl::backend::spmv(1, *R, rhs, 0, new_rhs);
103         rhs.swap(new_rhs);
104
105         // Update the number of the local rows
106         // (it may have changed as a result of permutation):
107         chunk = A->loc_rows();
108         prof.toc("partition");
109     }
110 #endif
```

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

```
112     // Initialize the solver:
113     prof.tic("setup");
114     Solver solve(world, A);
```

(continues on next page)

(continued from previous page)

```
115     prof.toc("setup");
116 
117     // Show the mini-report on the constructed solver:
118     if (world.rank == 0)
119         std::cout << solve << std::endl;
120 
121     // Solve the system with the zero initial approximation:
122     int iters;
123     double error;
124     std::vector<double> x(chunk, 0.0);
125 
126     prof.tic("solve");
127     std::tie(iters, error) = solve(*A, rhs, x);
128     prof.toc("solve");
```

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

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

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

level     unknowns       nonzeros
---------------------------------
    0        85623        2374949 (83.06%) [4]
    1         6377         450473 (15.75%) [4]
    2          401          34039 ( 1.19%) [4]

Iters: 24
Error: 6.09835e-09

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

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

```
1 #include <vector>
```

(continues on next page)

(continued from previous page)

```
#include <iostream>

#include <amgcl/backend/vexcl.hpp>
#include <amgcl/adapter/crs_tuple.hpp>

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

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

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

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

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

    // Create VexCL context. Use vex::Filter::Exclusive so that different MPI
    // processes get different GPUs. Each process gets a single GPU:
    vex::Context ctx(vex::Filter::Exclusive(vex::Filter::Count(1)));
    for(int i = 0; i < world.size; ++i) {
        // unclutter the output:
        if (i == world.rank)
            std::cout << world.rank << ": " << ctx.queue(0) << std::endl;
        MPI_Barrier(world);
    }

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

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

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

```

(continues on next page)

(continued from previous page)

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

    // Copy the RHS vector to the backend:
    vex::vector<double> f(ctx, rhs);

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

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

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

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

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

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

        // Reorder the matrix:
        A = product(*R, *product(*A, *P));

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

(continues on next page)

(continued from previous page)

```
        f.swap(new_rhs);

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

    // Initialize the solver:
    Solver::params prm;
    DBackend::params bprm;
    bprm.q = ctx;

    prof.tic("setup");
    Solver solve(world, A, prm, bprm);
    prof.toc("setup");

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

    // Solve the system with the zero initial approximation:
    int iters;
    double error;
    vex::vector<double> x(ctx, chunk);
    x = 0.0;

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

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

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

```
$ export OMP_NUM_THREADS=2
$ mpirun -np 2 ./poisson3Db_mpi_vexcl_cl poisson3Db.bin poisson3Db_b.bin
0: GeForce GTX 960 (NVIDIA CUDA)
1: GeForce GTX 1050 Ti (NVIDIA CUDA)
World size: 2
Matrix poisson3Db.bin: 85623x85623
RHS poisson3Db_b.bin: 85623x1
Partitioning[ParMETIS] 2 -> 2
Type:              BiCGStab
```

(continues on next page)

(continued from previous page)

```
Unknowns:          43255
Memory footprint: 2.31 M

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

level     unknowns       nonzeros
---------------------------------
    0        85623        2374949 (83.03%) [2]
    1         6381         451279 (15.78%) [2]
    2          396          34054 ( 1.19%) [2]

Iters: 24
Error: 9.14603e-09

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

2.3.3 Structural problem

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

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

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

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

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

level     unknowns       nonzeros      memory
---------------------------------------------
```

(continues on next page)

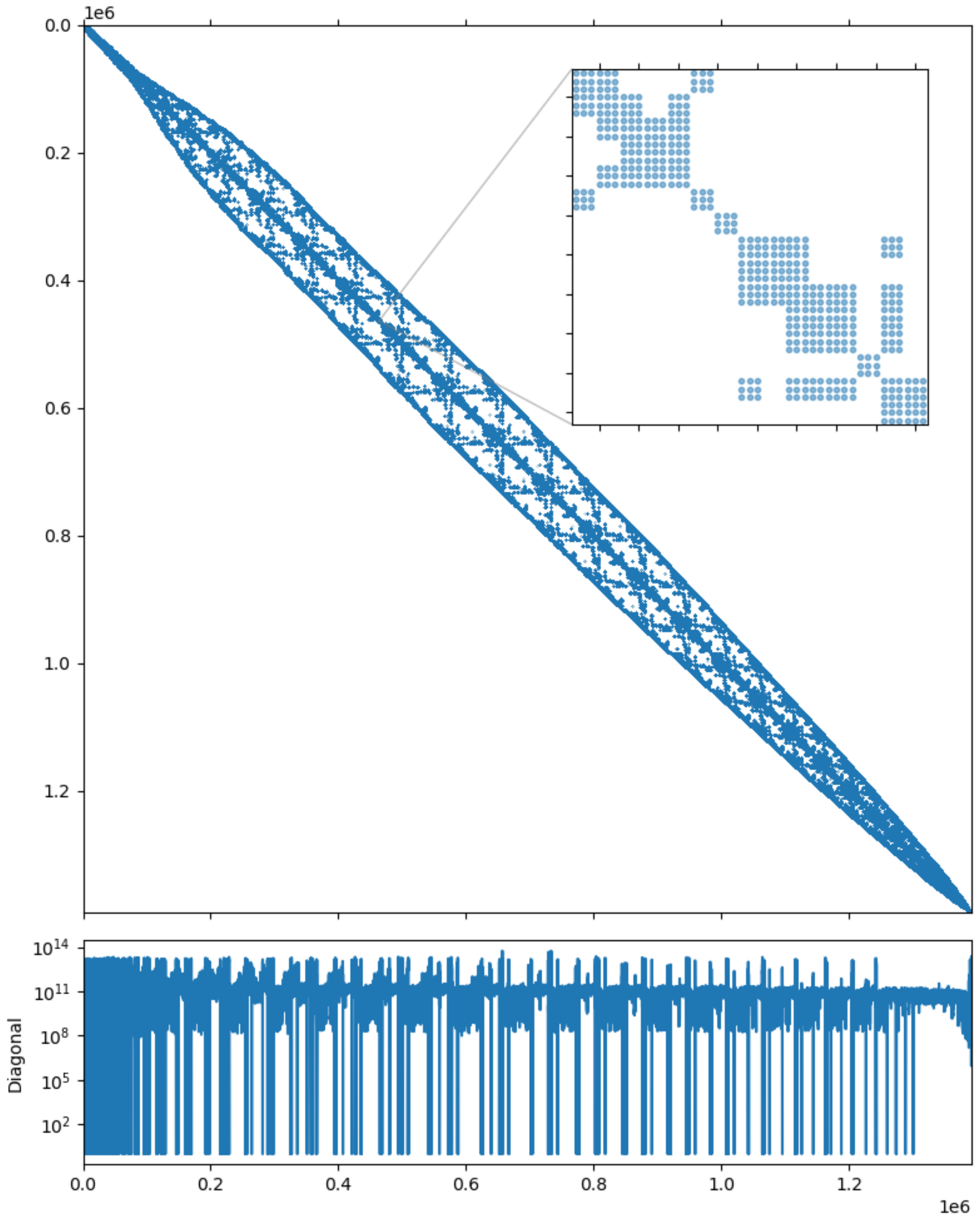


Fig. 2.4: Serena matrix portrait

(continued from previous page)

```
    0      1391349       64531701      1.22 G (82.01%)
    1        98824       13083884    218.40 M (16.63%)
    2         5721        1038749     16.82 M ( 1.32%)
    3          279          29151    490.75 K ( 0.04%)

Iterations: 100
Error:      0.000874761

[Profile:      74.102 s] (100.00%)
[  reading:    18.505 s] ( 24.97%)
[  setup:       2.101 s] (  2.84%)
[  solve:      53.489 s] ( 72.18%)
```

The system is quite large and just reading from the text-based Matrix Market format takes 18.5 seconds. No one has that amount of free time on their hands, so lets convert the matrix into the binary format with the examples/mm2bin utility. This should make the experiments slightly less painful:

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

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

```
$ solver -B -A Serena.bin solver.maxiter=1000
Solver
======
Type:             BiCGStab
Unknowns:         1391349
Memory footprint: 74.31 M

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

level     unknowns       nonzeros      memory
---------------------------------------------
    0      1391349       64531701      1.22 G (82.01%)
    1        98824       13083884    218.40 M (16.63%)
    2         5721        1038749     16.82 M ( 1.32%)
    3          279          29151    490.75 K ( 0.04%)

Iterations: 211
Error:      8.54558e-09

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

The input matrix is read much faster now, and the solver does converge, but the convergence rate is not great. Looking closer at the *Serena matrix portrait* figure, the matrix seems to have block structure with 3×3 blocks. This is usually the case when the system has been obtained via discretization of a system of coupled PDEs, or has vector unknowns. We have to guess here, but since the problem is described as “structural”, then each block probably corresponds to the 3D displacement vector of a single grid node. We can communicate this piece of information to AMGCL using the `block_size` parameter of the aggregation method:

```
$ solver -B -A Serena.bin solver.maxiter=1000 \
      precond.coarsening.aggr.block_size=3
Solver
======
Type:             BiCGStab
Unknowns:         1391349
Memory footprint: 74.31 M

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

level     unknowns       nonzeros      memory
---------------------------------------------
    0      1391349       64531701      1.50 G (76.48%)
    1       109764       17969220    316.66 M (21.30%)
    2         6291        1788507     29.51 M ( 2.12%)
    3          429          82719      1.23 M ( 0.10%)

Iterations: 120
Error:      9.73074e-09

[Profile:      73.296 s] (100.00%)
[  reading:     0.587 s] (  0.80%)
[  setup:       2.709 s] (  3.70%)
[  solve:      69.994 s] ( 95.49%)
```

This has definitely improved the convergence! We also know that the matrix is symmetric, so lets switch the solver from the default BiCGStab to the slightly less expensive CG:

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

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

level     unknowns       nonzeros      memory
---------------------------------------------
    0      1391349       64531701      1.50 G (76.48%)
    1       109764       17969220    316.66 M (21.30%)
    2         6291        1788507     29.51 M ( 2.12%)
    3          429          82719      1.23 M ( 0.10%)
```

(continues on next page)

(continued from previous page)

```

Iterations: 177
Error:      8.6598e-09

[Profile:      55.250 s] (100.00%)
[  reading:     0.550 s] (  1.00%)
[  setup:       2.801 s] (  5.07%)
[  solve:      51.894 s] ( 93.92%)
```

This reduces the solution time, even though the number of iterations has grown. Each iteration of BiCGStab costs about twice as much as a CG iteration, because BiCGStab does two matrix-vector products and preconditioner applications per iteration, while CG only does one.

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

```
$ solver -B -A Serena.bin -s \
      solver.type=cg solver.maxiter=200 \
      precond.coarsening.aggr.block_size=3
Solver
======
Type:             CG
Unknowns:         1391349
Memory footprint: 42.46 M

Preconditioner
==============
Number of levels:    4
Operator complexity: 1.29
Grid complexity:     1.08
Memory footprint:    1.82 G

level     unknowns       nonzeros      memory
---------------------------------------------
    0      1391349       64531701      1.51 G (77.81%)
    1       100635       16771185    294.81 M (20.22%)
    2         5643        1571157     25.92 M ( 1.89%)
    3          342          60264    802.69 K ( 0.07%)

Iterations: 112
Error:      9.84457e-09

[Profile:      36.021 s] (100.00%)
[ self:         0.204 s] (  0.57%)
[  reading:     0.564 s] (  1.57%)
[  setup:       2.684 s] (  7.45%)
[  solve:      32.568 s] ( 90.42%)
```

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

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

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

level     unknowns       nonzeros      memory
---------------------------------------------
    0       463783        7170189    891.53 M (78.80%)
    1        33052        1772434    159.22 M (19.48%)
    2         1722         151034     12.72 M ( 1.66%)
    3           98           5756    612.72 K ( 0.06%)

Iterations: 162
Error:      9.7497e-09

[Profile:      31.122 s] (100.00%)
[ self:         0.204 s] (  0.66%)
[  reading:     0.550 s] (  1.77%)
[  setup:       1.013 s] (  3.26%)
[  solve:      29.354 s] ( 94.32%)
```

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
72
73      typedef amgcl::make_solver<
74          amgcl::amg<
75              PBackend,
76              amgcl::coarsening::smoothed_aggregation,
77              amgcl::relaxation::spai0
78              >,
79          amgcl::solver::cg<SBackend>
80          > Solver;
81
82      // Solver parameters
83      Solver::params prm;
84      prm.solver.maxiter = 500;
85
86      // Initialize the solver with the system matrix.
87      // Use the block_matrix adapter to convert the matrix into
88      // the block format on the fly:
89      prof.tic("setup");
90      auto Ab = amgcl::adapter::block_matrix<dmat_type>(A);
91      Solver solve(Ab, prm);
92      prof.toc("setup");
93
94      // Show the mini-report on the constructed solver:
95      std::cout << solve << std::endl;
96
97      // Solve the system with the zero initial approximation:
98      int iters;
99      double error;
100     std::vector<double> x(rows, 0.0);
101
102     // Reinterpret both the RHS and the solution vectors as block-valued:
103     auto f_ptr = reinterpret_cast<dvec_type*>(f.data());
104     auto x_ptr = reinterpret_cast<dvec_type*>(x.data());
105     auto F = amgcl::make_iterator_range(f_ptr, f_ptr + rows / 3);
106     auto X = amgcl::make_iterator_range(x_ptr, x_ptr + rows / 3);
107
108     prof.tic("solve");
109     std::tie(iters, error) = solve(Ab, F, X);
110     prof.toc("solve");
111
112     // Output the number of iterations, the relative error,
113     // and the profiling data:
114     std::cout << "Iters: " << iters << std::endl
115               << "Error: " << error << std::endl
116               << prof << std::endl;
117 }
```

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

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

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

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

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

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

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

Here is the output of the program:

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

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

level     unknowns       nonzeros      memory
---------------------------------------------
    0       463783        7170189    490.45 M (78.80%)
    1        33052        1772434     87.58 M (19.48%)
    2         1722         151034      7.00 M ( 1.66%)
    3           98           5756    306.75 K ( 0.06%)

Iters: 162
Error: 9.74929e-09

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

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

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

Listing 2.9: The solution of the Serena problem with the VexCL backend.

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

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
108
109     // Show the mini-report on the constructed solver:
110     std::cout << solve << std::endl;
111
112     // Solve the system with the zero initial approximation:
113     int iters;
114     double error;
115     std::vector<double> x(rows, 0.0);
116
117     // Since we are using mixed precision, we have to transfer the system matrix to␣
    ↪the GPU:
118     prof.tic("GPU matrix");
119     auto A_gpu = SBackend::copy_matrix(
120             std::make_shared<amgcl::backend::crs<dmat_type>>(Ab), bprm);
121     prof.toc("GPU matrix");
122
123     // We reinterpret both the RHS and the solution vectors as block-valued,
124     // and copy them to the VexCL vectors:
125     auto f_ptr = reinterpret_cast<dvec_type*>(f.data());
126     auto x_ptr = reinterpret_cast<dvec_type*>(x.data());
127     vex::vector<dvec_type> F(ctx, rows / 3, f_ptr);
128     vex::vector<dvec_type> X(ctx, rows / 3, x_ptr);
129
130     prof.tic("solve");
131     std::tie(iters, error) = solve(*A_gpu, F, X);
132     prof.toc("solve");
133
134     // Output the number of iterations, the relative error,
135     // and the profiling data:
136     std::cout << "Iters: " << iters << std::endl
137               << "Error: " << error << std::endl
138               << prof << std::endl;
139 }
```

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

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

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

The output of the program is shown below:

```
$ ./serena_vexcl_cuda Serena.mtx
1. GeForce GTX 1050 Ti

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

```

(continues on next page)

(continued from previous page)

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

level     unknowns       nonzeros      memory
---------------------------------------------
    0       463783        7170189    490.45 M (78.80%)
    1        33052        1772434     87.58 M (19.48%)
    2         1722         151034      7.00 M ( 1.66%)
    3           98           5756    309.04 K ( 0.06%)

Iters: 162
Error: 9.74928e-09

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

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

2.3.4 Structural problem (MPI version)

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

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

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

Listing 2.10: The MPI solution of the Serena problem

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

#include <amgcl/backend/builtin.hpp>
#include <amgcl/value_type/static_matrix.hpp>
#include <amgcl/adapter/crs_tuple.hpp>
```

(continues on next page)

(continued from previous page)

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

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

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

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

// Block size
const int B = 3;

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```

    // Declare the backend and the solver types
    typedef amgcl::static_matrix<double, B, B> dmat_type;
    typedef amgcl::static_matrix<double, B, 1> dvec_type;
    typedef amgcl::static_matrix<float,  B, B> fmat_type;
    typedef amgcl::backend::builtin<dmat_type> DBackend;
    typedef amgcl::backend::builtin<fmat_type> FBackend;

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

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

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

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

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

```

(continues on next page)

(continued from previous page)

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

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

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

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

        // Reorder the matrix:
        A = product(*R, *product(*A, *P));

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
    prof.toc("solve");

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

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

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

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

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

```
    // Declare the backend and the solver types
    typedef amgcl::static_matrix<double, B, B> dmat_type;
    typedef amgcl::static_matrix<double, B, 1> dvec_type;
    typedef amgcl::static_matrix<float,  B, B> fmat_type;
    typedef amgcl::backend::builtin<dmat_type> DBackend;
    typedef amgcl::backend::builtin<fmat_type> FBackend;

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

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

```
    // We need to scale the matrix, so that it has the unit diagonal.
    // Since we only have the local rows for the matrix, and we may need the
    // remote diagonal values, it is more convenient to represent the scaling
    // with the matrix-matrix product (As = D^-1/2 A D^-1/2).
    prof.tic("scale");
    // Find the local diagonal values,
    // and form the CRS arrays for a diagonal matrix.
    std::vector<double> dia(chunk, 1.0);
```

(continues on next page)

(continued from previous page)

```
    std::vector<ptrdiff_t> d_ptr(chunk + 1), d_col(chunk);
    for(ptrdiff_t i = 0, I = row_beg; i < chunk; ++i, ++I) {
        d_ptr[i] = i;
        d_col[i] = I;
        for(ptrdiff_t j = ptr[i], e = ptr[i+1]; j < e; ++j) {
            if (col[j] == I) {
                dia[i] = 1 / sqrt(val[j]);
                break;
            }
        }
    }
    d_ptr.back() = chunk;

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

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

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

Here is the output from the compiled program:

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

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

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

Iterations: 80
```

(continues on next page)

(continued from previous page)

```
Error:      9.34355e-09

[Serena MPI:     24.840 s] (100.00%)
[  partition:     1.159 s] (  4.67%)
[  read:          0.265 s] (  1.07%)
[  scale:         0.583 s] (  2.35%)
[  setup:         0.811 s] (  3.26%)
[  solve:        22.017 s] ( 88.64%)
```

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

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

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

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

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

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

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

// Block size
const int B = 3;

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

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

    // Create VexCL context. Use vex::Filter::Exclusive so that different MPI
    // processes get different GPUs. Each process gets a single GPU:
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

    // Declare the backend and the solver types
    typedef amgcl::static_matrix<double, B, B> dmat_type;
    typedef amgcl::static_matrix<double, B, 1> dvec_type;
    typedef amgcl::static_matrix<float,  B, B> fmat_type;
    typedef amgcl::backend::vexcl<dmat_type> DBackend;
    typedef amgcl::backend::vexcl<fmat_type> FBackend;

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

```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
    typedef amgcl::mpi::partition::ptscotch<DBackend> Partition;
#   endif

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

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

        // Reorder the matrix:
        A = product(*R, *product(*A, *P));

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

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

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

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

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

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

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

Here is the output of the MPI version with the VexCL backend:

```
$ export OMP_NUM_THREADS=1
$ mpirun -np 2 ./serena_mpi_vexcl_cl Serena.bin
0: GeForce GTX 960 (NVIDIA CUDA)
1: GeForce GTX 1050 Ti (NVIDIA CUDA)
World size: 2
Matrix Serena.bin: 1391349x1391349
Partitioning[ParMETIS] 2 -> 2
Type:             BiCGStab
Unknowns:         231112
Memory footprint: 37.03 M

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

level     unknowns       nonzeros
---------------------------------
    0       463783        7170189 (79.01%) [2]
    1        32887        1754795 (19.34%) [2]
    2         1708         146064 ( 1.61%) [2]
    3           85           4059 ( 0.04%) [2]

Iterations: 83
Error:      9.80582e-09

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

2.3.5 Fully coupled poroelastic problem

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

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

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

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

Preconditioner
==============
Number of levels:    4
```

(continues on next page)

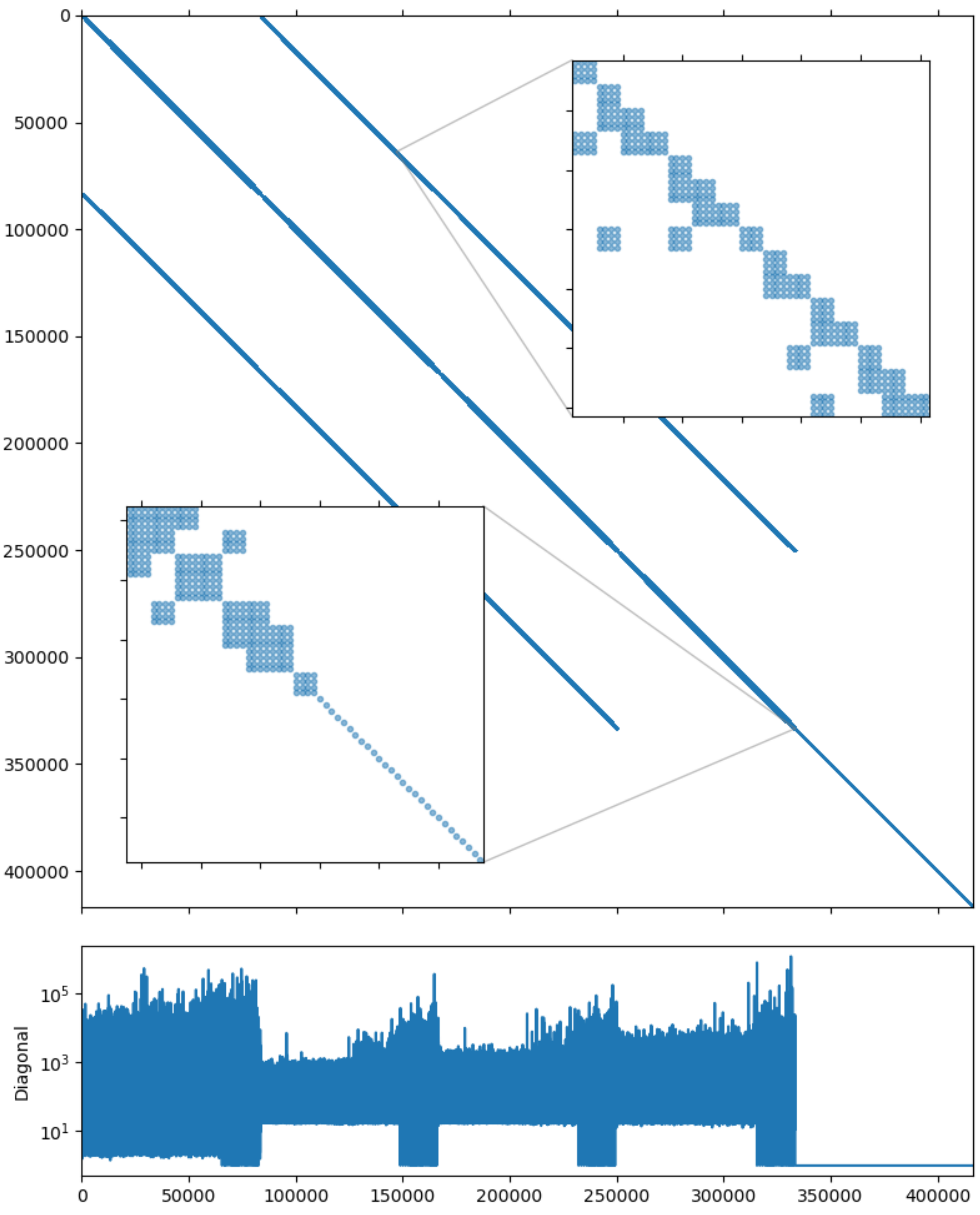


Fig. 2.5: CoupCons3D matrix portrait

(continued from previous page)

```
Operator complexity: 1.11
Grid complexity:     1.09
Memory footprint:    447.17 M

level     unknowns       nonzeros      memory
---------------------------------------------
    0       416800       22322336    404.08 M (90.13%)
    1        32140        2214998     38.49 M ( 8.94%)
    2         3762         206242      3.58 M ( 0.83%)
    3          522          22424      1.03 M ( 0.09%)

Iterations: 100
Error:      0.705403

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

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

```
$ solver -B -A CoupCons3D.bin precond.relax.type=ilu0
Solver
======
Type:              BiCGStab
Unknowns:          416800
Memory footprint: 22.26 M

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

level     unknowns       nonzeros      memory
---------------------------------------------
    0       416800       22322336    751.33 M (90.13%)
    1        32140        2214998     72.91 M ( 8.94%)
    2         3762         206242      6.85 M ( 0.83%)
    3          522          22424      1.03 M ( 0.09%)

Iterations: 47
Error:      4.8263e-09

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

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

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

(continues on next page)

(continued from previous page)

```
Type:             BiCGStab
Unknowns:         416800
Memory footprint: 22.26 M

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

level     unknowns       nonzeros      memory
---------------------------------------------
    0       416800       22322336    751.33 M (90.54%)
    1        32140        2214998     73.06 M ( 8.98%)
    2         2221         116339     10.12 M ( 0.47%)

Iterations: 11
Error:      9.79966e-09

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

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

```
$ solver -B -A CoupCons3D.bin precond.relax.type=ilu0 -s -b4
Solver
======
Type:             BiCGStab
Unknowns:         104200
Memory footprint: 22.26 M

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

level     unknowns       nonzeros      memory
---------------------------------------------
    0       104200        1395146    445.04 M (84.98%)
    1        10365         235821     70.07 M (14.36%)
    2          600          10792     10.57 M ( 0.66%)

Iterations: 4
Error:      2.90461e-09

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

(continues on next page)

(continued from previous page)

```
[   reading:      0.188 s] ( 13.84%)
[   setup:        0.478 s] ( 35.23%)
[   solve:        0.628 s] ( 46.30%)
```

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

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```

    // Reinterpret both the RHS and the solution vectors as block-valued:
    auto f_ptr = reinterpret_cast<dvec_type*>(f.data());
    auto x_ptr = reinterpret_cast<dvec_type*>(x.data());
    auto F = amgcl::make_iterator_range(f_ptr, f_ptr + rows / 4);
    auto X = amgcl::make_iterator_range(x_ptr, x_ptr + rows / 4);

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

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

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

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

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

level     unknowns       nonzeros      memory
---------------------------------------------
    0       104200        1395146    237.27 M (84.98%)
    1        10365         235821     37.27 M (14.36%)
    2          600          10792      5.29 M ( 0.66%)

Iters: 4
Error: 2.90462e-09

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

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

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

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

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

level     unknowns       nonzeros      memory
---------------------------------------------
    0       104200        1395146    238.79 M (84.98%)
    1        10365         235821     37.40 M (14.36%)
    2          600          10792      5.31 M ( 0.66%)

Iters: 5
Error: 6.30647e-09

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

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

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

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

[ U ]
```

(continues on next page)

(continued from previous page)

```
Solver
======
Type:             PreOnly
Unknowns:         83360
Memory footprint: 0.00 B

Preconditioner
==============
Number of levels:    4
Operator complexity: 1.21
Grid complexity:     1.23
Memory footprint:    206.09 M

level     unknowns       nonzeros      memory
---------------------------------------------
    0        83360        1082798    167.26 M (82.53%)
    1        14473         184035     28.49 M (14.03%)
    2         4105          39433      6.04 M ( 3.01%)
    3          605           5761      4.30 M ( 0.44%)


[ P ]
Solver
======
Type:             PreOnly
Unknowns:         83360
Memory footprint: 0.00 B

Preconditioner
==============
Relaxation as preconditioner
  Unknowns: 83360
  Nonzeros: 2332064
  Memory:   27.64 M


Iters: 7
Error: 5.0602e-09

[CoupCons3D:    14.427 s] (100.00%)
[  read:        13.010 s] ( 90.18%)
[  setup:        0.336 s] (  2.33%)
[  solve:        1.079 s] (  7.48%)
```

2.3.6 Stokes-like problem

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

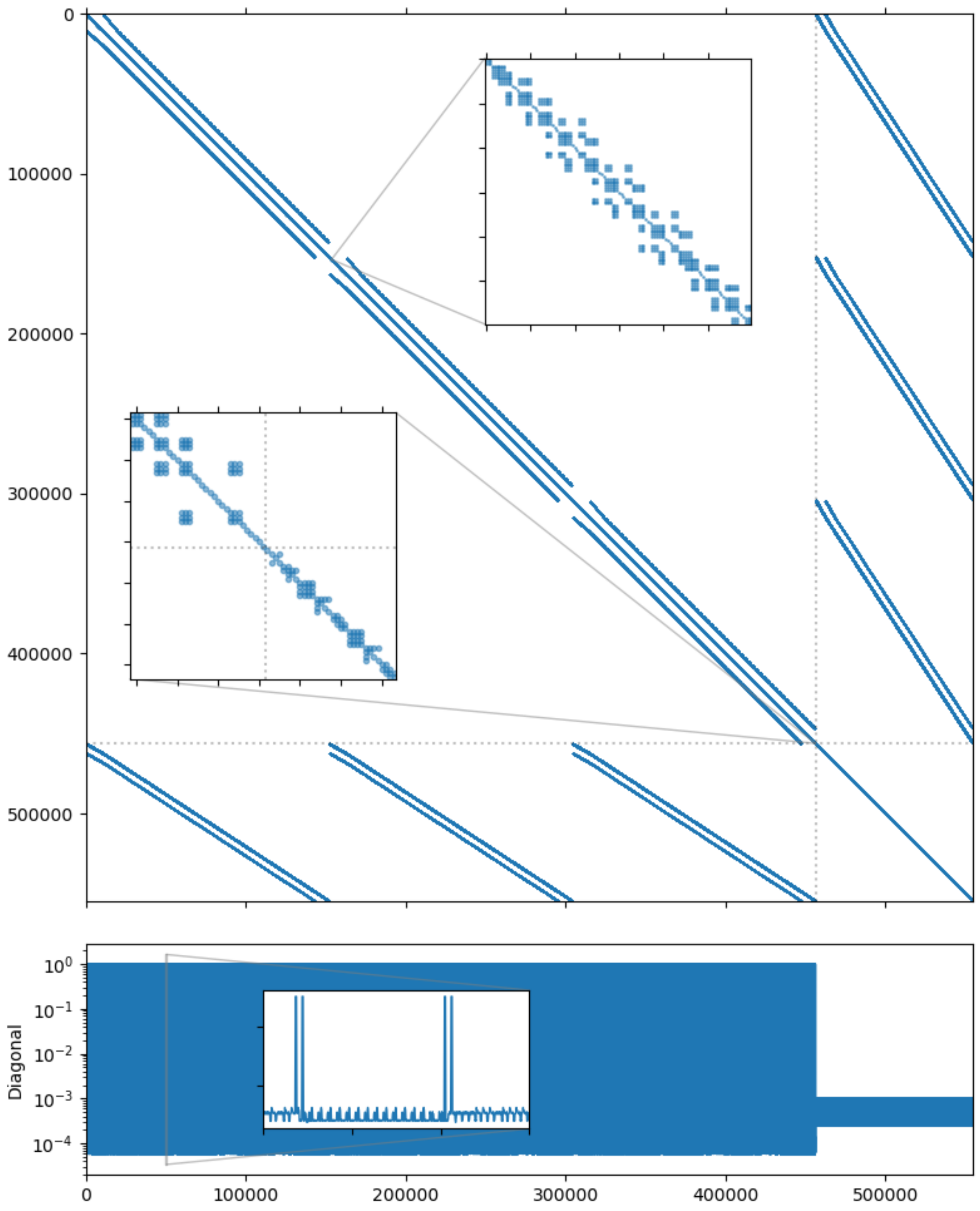


Fig. 2.6: UCube(4) matrix portrait

As with any Stokes-like problem, the system has a general block-wise structure:

$$\begin{bmatrix} A & B_1^T \\ B_2 & C \end{bmatrix} \begin{bmatrix} \mathbf{x}_u \\ \mathbf{x}_p \end{bmatrix} = \begin{bmatrix} \mathbf{b}_u \\ \mathbf{b}_p \end{bmatrix}$$

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

```
$ solver -B -A ucube_4_A.bin -f ucube_4_b.bin \
      solver.type=cg solver.maxiter=500 \
      precond.class=relaxation precond.type=ilu0
Solver
======
Type:             CG
Unknowns:         554496
Memory footprint: 16.92 M

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

Iterations: 270
Error:      6.84763e-09

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

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

$$\begin{aligned} S\mathbf{x}_p &= \mathbf{b}_p - B_2 A^{-1} \mathbf{b}_u, \\ A\mathbf{x}_u &= \mathbf{b}_u - B_1^T \mathbf{b}_p. \end{aligned} \tag{2.1}$$

Here S is the Schur complement $S = C - B_2 A^{-1} B_1^T$. Note that forming the Schur complement matrix explicitly is prohibitively expensive, and the following approximation is used to create the preconditioner for the first equation in (2.1):

$$\hat{S} = C - \operatorname{diag}\left(B_2 \operatorname{diag}(A)^{-1} B_1^T\right).$$

There is no need to solve the equations (2.1) exactly. It is enough to perform a single application of the corresponding preconditioner as an approximation to S^{-1} and A^{-1}. This means that the overall preconditioner is linear, and we may use a non-flexible iterative solver with it. The approximation matrix $\hat{S}$ has a simple band diagonal structure, and a diagonal SPAI(0) preconditioner should have reasonable performance.

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

```
$ schur_pressure_correction -B -A ucube_4_A.bin -f ucube_4_b.bin -m '>456192' \
    -p solver.type=cg solver.maxiter=200 \
       precond.simplec_dia=false \
       precond.usolver.solver.type=preonly \
       precond.usolver.precond.coarsening.type=aggregation \
       precond.usolver.precond.relax.type=ilu0 \
       precond.psolver.solver.type=preonly \
       precond.psolver.precond.class=relaxation \
       --ub 3
Solver
======
Type:              CG
Unknowns:          554496
Memory footprint: 16.92 M

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

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

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

level     unknowns       nonzeros      memory
---------------------------------------------
    0       152064         982416    188.13 M (80.25%)
    1        18654         197826     35.07 M (16.16%)
    2         2619          35991      6.18 M ( 2.94%)
    3          591           7953      3.69 M ( 0.65%)


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

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

(continues on next page)

(continued from previous page)

```
  Unknowns: 98304
  Nonzeros: 274472
  Memory:   5.69 M


Iterations: 35
Error:      8.57921e-09

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
    std::vector<ptrdiff_t> ptr, col;
    std::vector<double> val, rhs;

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

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

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

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

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

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

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

```

(continues on next page)

(continued from previous page)

```
92      // Show the mini-report on the constructed solver:
93      std::cout << solve << std::endl;
94
95      // Solve the system with the zero initial approximation:
96      int iters;
97      double error;
98      std::vector<double> x(rows, 0.0);
99      prof.tic("solve");
100     std::tie(iters, error) = solve(A, rhs, x);
101     prof.toc("solve");
102
103     // Output the number of iterations, the relative error,
104     // and the profiling data:
105     std::cout << "Iters: " << iters << std::endl
106               << "Error: " << error << std::endl
107               << prof << std::endl;
108 }
```

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

```
55      typedef amgcl::backend::builtin<double> SBackend; // the outer iterative solver␣
   ↪backend
56      typedef amgcl::backend::builtin<float> PBackend;  // the PSolver backend
57      typedef amgcl::backend::builtin<
58          amgcl::static_matrix<float,3,3>> UBackend;    // the USolver backend
```

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

```
60      typedef amgcl::make_solver<
61          amgcl::preconditioner::schur_pressure_correction<
62              amgcl::make_block_solver<
63                  amgcl::amg<
64                      UBackend,
65                      amgcl::coarsening::aggregation,
66                      amgcl::relaxation::ilu0
67                      >,
68                  amgcl::solver::preonly<UBackend>
69                  >,
70              amgcl::make_solver<
71                  amgcl::relaxation::as_preconditioner<
72                      PBackend,
73                      amgcl::relaxation::spai0
74                      >,
75                  amgcl::solver::preonly<PBackend>
76                  >
77              >,
78          amgcl::solver::cg<SBackend>
79          > Solver;
```

In the solver parameters we disable lumping of the matrix A diagonal for the Schur complement approimation (line

83) and fill the pressure mask to indicate which unknowns correspond to the pressure subsystem (lines 84–85):

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

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

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

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

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

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

level     unknowns       nonzeros      memory
---------------------------------------------
    0       152064         982416    105.64 M (80.25%)
    1        18654         197826     19.56 M (16.16%)
    2         2619          35991      3.44 M ( 2.94%)
    3          591           7953      1.85 M ( 0.65%)


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

Preconditioner
```

(continues on next page)

(continued from previous page)

```
==============
Relaxation as preconditioner
  Unknowns: 98304
  Nonzeros: 274472
  Memory:   4.27 M


Iters: 35
Error: 8.57996e-09

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

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

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

```
1  #include <iostream>
2  #include <string>
3
4  #include <amgcl/backend/vexcl.hpp>
5  #include <amgcl/backend/vexcl_static_matrix.hpp>
6  #include <amgcl/adapter/crs_tuple.hpp>
7  #include <amgcl/value_type/static_matrix.hpp>
8  #include <amgcl/adapter/block_matrix.hpp>
9  #include <amgcl/preconditioner/schur_pressure_correction.hpp>
10 #include <amgcl/make_solver.hpp>
11 #include <amgcl/make_block_solver.hpp>
12 #include <amgcl/amg.hpp>
13 #include <amgcl/solver/cg.hpp>
14 #include <amgcl/solver/preonly.hpp>
15 #include <amgcl/coarsening/aggregation.hpp>
16 #include <amgcl/relaxation/ilu0.hpp>
17 #include <amgcl/relaxation/spai0.hpp>
18 #include <amgcl/relaxation/as_preconditioner.hpp>
19
20 #include <amgcl/io/binary.hpp>
21 #include <amgcl/profiler.hpp>
22
23 //---------------------------------------------------------------------------
24 int main(int argc, char *argv[]) {
25     // The command line should contain the matrix and the RHS file names,
26     // and the number of unknowns in the flow subsytem:
27     if (argc < 4) {
28         std::cerr << "Usage: " << argv[0] << " <matrix.bin> <rhs.bin> <nu>" << std::endl;
29         return 1;
30     }
31
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
            >,
        amgcl::solver::cg<SBackend>
        > Solver;

    // Solver parameters
    Solver::params prm;
    prm.precond.simplec_dia = false;
    prm.precond.usolver.precond.relax.solve.iters = 4;
    prm.precond.pmask.resize(rows);
    for(ptrdiff_t i = 0; i < rows; ++i) prm.precond.pmask[i] = (i >= nu);

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

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

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

    // Since we are using mixed precision, we have to transfer the system matrix to␣
↪the GPU:
    prof.tic("GPU matrix");
    auto A_gpu = SBackend::copy_matrix(std::make_shared<amgcl::backend::crs<double>>
↪(A), bprm);
    prof.toc("GPU matrix");

    // Solve the system with the zero initial approximation:
    int iters;
    double error;
    vex::vector<double> f(ctx, rhs);
    vex::vector<double> x(ctx, rows);
    x = 0.0;

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

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

The output of the VexCL version is shown below. The solution phase is about twice as fast as the OpenMP version:

```
$ ./stokes_ucube_vexcl_cuda ucube_4_A.bin ucube_4_b.bin 456192
1. GeForce GTX 1050 Ti

Matrix ucube_4_A.bin: 554496x554496
RHS ucube_4_b.bin: 554496x1
Solver
======
```

(continues on next page)

(continued from previous page)

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

Preconditioner
==============
Schur complement (two-stage preconditioner)
  Unknowns: 554496(98304)
  Nonzeros: 14292884
  Memory:  399.66 M

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

Preconditioner
==============
Number of levels:    4
Operator complexity: 1.25
Grid complexity:     1.14
Memory footprint:    131.76 M

level     unknowns       nonzeros      memory
---------------------------------------------
    0       152064         982416    106.76 M (80.25%)
    1        18654         197826     19.68 M (16.16%)
    2         2619          35991      3.45 M ( 2.94%)
    3          591           7953      1.86 M ( 0.65%)


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

Preconditioner
==============
Relaxation as preconditioner
  Unknowns: 98304
  Nonzeros: 274472
  Memory:   4.27 M


Iters: 36
Error: 7.26253e-09

[UCube4 (VexCL):   1.858 s] (100.00%)
[ self:            0.004 s] (  0.20%)
[  GPU matrix:     0.213 s] ( 11.46%)
[  read:           0.128 s] (  6.87%)
[  setup:          0.519 s] ( 27.96%)
[  solve:          0.994 s] ( 53.52%)
```

2.4 Examples

2.4.1 Solving Poisson's equation

The easiest way to solve a problem with AMGCL is to use the `amgcl::make_solver` class. It has two template parameters: the first one specifies a preconditioner to use, and the second chooses an iterative solver. The class constructor takes the system matrix in one of supported formats and parameters for the chosen algorithms and for the backend.

Let us consider a simple example of Poisson's equation in a unit square. Here is how the problem may be solved with AMGCL. We will use BiCGStab solver preconditioned with smoothed aggregation multigrid with SPAI(0) for relaxation (smoothing). First, we include the necessary headers. Each of those brings in the corresponding component of the method:

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

Next, we assemble sparse matrix for the Poisson's equation on a uniform 1000x1000 grid. See below for the definition of the `poisson()` function:

```
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 `std::tuple` 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.

2.4.2 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( 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 adapters for further information on this.

2.4.3 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 is used, and the exact structure of the parameters is not known at compile time:

```
boost::property_tree::ptree prm;
prm.put("solver.tol", 1e-3);
Solver solve( std::tie(n, ptr, col, val), prm );
```

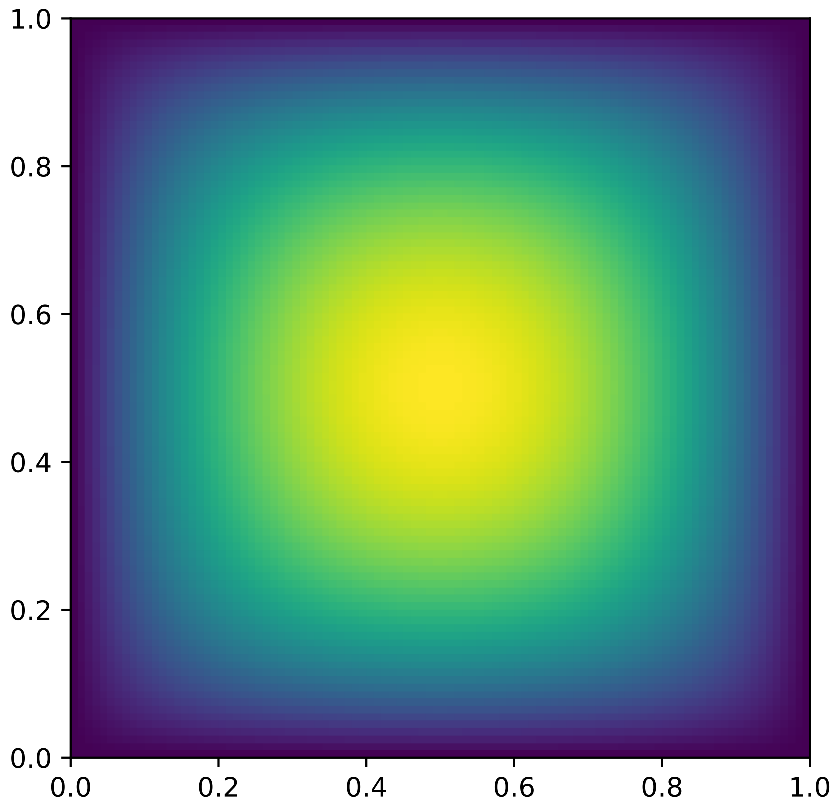
2.4.4 Assembling matrix for Poisson's equation

The section provides an example of assembling the system matrix and the right-hand side for a Poisson's equation in a unit square $\Omega = [0, 1] \times [0, 1]$:

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

The solution to the problem looks like this:

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


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

```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;
}
```

2.5 Distributed Memory Solvers

2.6 Benchmarks

The performance of the shared memory and the distributed memory versions of AMGCL algorithms was tested on two example problems in a three dimensional space. The source code for the benchmarks is available at https://github.com/ddemidov/amgcl_benchmarks.

The first example is the classical 3D Poisson problem. Namely, we look for the solution of the problem

$$-\Delta u = 1,$$

in the unit cube $\Omega = [0,1]^3$ with homogeneous Dirichlet boundary conditions. The problem is discretized with the finite difference method on a uniform mesh.

The second test problem is an incompressible 3D Navier-Stokes problem discretized on a non uniform 3D mesh with a finite element method:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} + \nabla p = \mathbf{b},$$
$$\nabla \cdot \mathbf{u} = 0.$$

The discretization uses an equal-order tetrahedral Finite Elements stabilized with an ASGS-type (algebraic subgrid-scale) approach. This results in a linear system of equations with a block structure of the type

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

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.

2.6.1 Shared Memory Benchmarks

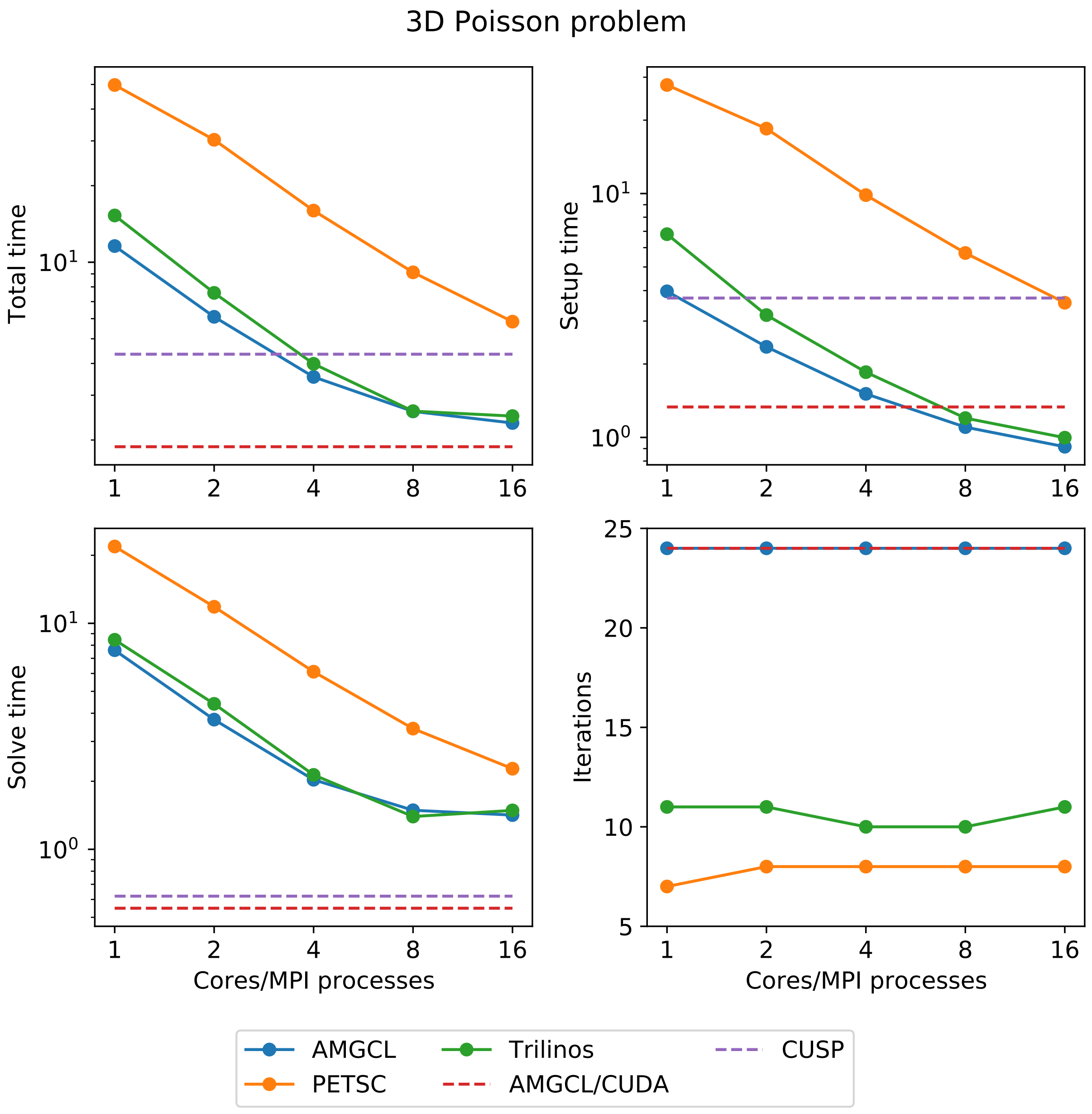
In this section we test performance of the library on a shared memory system. We also compare the results with PETSC and Trilinos ML distributed memory libraries and CUSP GPGPU library. The tests were performed on a dual socket system with two Intel Xeon E5-2640 v3 CPUs. The system also had an NVIDIA Tesla K80 GPU installed, which was used for testing the GPU based versions.

3D Poisson problem

The Poisson problem is discretized with the finite difference method on a uniform mesh, and the resulting linear system contained 3375000 unknowns and 23490000 nonzeros.

The figure below presents the multicore scalability of the problem. Here AMGCL uses the `builtin` OpenMP backend, while PETSC and Trilinos use MPI for parallelization. We also show results for the CUDA backend of AMGCL library compared with the CUSP library. All libraries use the Conjugate Gradient iterative solver preconditioned with a smoothed aggregation AMG. Trilinos and PETSC use default options for smoothers (symmetric Gauss-Seidel and damped Jacobi accordingly) on each level of the hierarchy, AMGCL uses SPAI0, and CUSP uses Gauss-Seidel smoother.

The CPU-based results show that AMGCL performs on par with Trilinos, and both of the libraries outperform PETSC by a large margin. Also, AMGCL is able to setup the solver about 20–100% faster than Trilinos, and 4–7 times faster than PETSC. This is probably due to the fact that both Trilinos and PETSC target distributed memory machines and hence need to do some complicated bookkeeping under the hood. PETSC shows better scalability than both Trilinos and AMGCL, which scale in a similar fashion.


3D Poisson problem
Total time
Setup time
Solve time
Iterations
Cores/MPI processes
1
2
4
8
16
10^1
10^0
5
10
15
20
25
AMGCL
PETSC
Trilinos
AMGCL/CUDA
CUSP

On the GPU, AMGCL performs slightly better than CUSP. If we consider the solution time (without setup), then both libraries are able to outperform CPU-based versions by a factor of 3-4. The total solution time of AMGCL with CUDA backend is only 30% better than that of either AMGCL with OpenMP backend or Trilinos ML. This is due to the fact that the setup step in AMGCL is always performed on the CPU and in case of the CUDA backend has an additional overhead of moving the constructed hierarchy into the GPU memory.

3D Navier-Stokes problem

The system matrix resulting from the problem discretization has block structure with blocks of 4-by-4 elements, and contains 713456 unknowns and 41277920 nonzeros. The assembled problem is available to download at https://doi.org/10.5281/zenodo.1231818.

There are at least two ways to solve the system. First, one can treat the system as a monolithic one, and provide some minimal help to the preconditioner in form of near null space vectors. Second option is to employ the knowledge about the problem structure, and to combine separate preconditioners for individual fields (in this particular case, for pressure and velocity). In case of AMGCL both options were tested, where the monolithic system was solved with static 4x4 matrices as value type, and the field-split approach was implemented using the `schur_pressure_correction` preconditioner. Trilinos ML only provides the first option; PETSC implement both options, but we only show results for the second, superior option here. CUSP library does not provide field-split preconditioner and does not allow to specify near null space vectors, so it was not tested for this problem.

The figure below shows multicore scalability results for the Navier-Stokes problem. Lines labelled with 'block' correspond to the cases when the problem is treated as a monolithic system, and 'split' results correspond to the field-split approach.

2.6.2 Distributed Memory Benchmarks

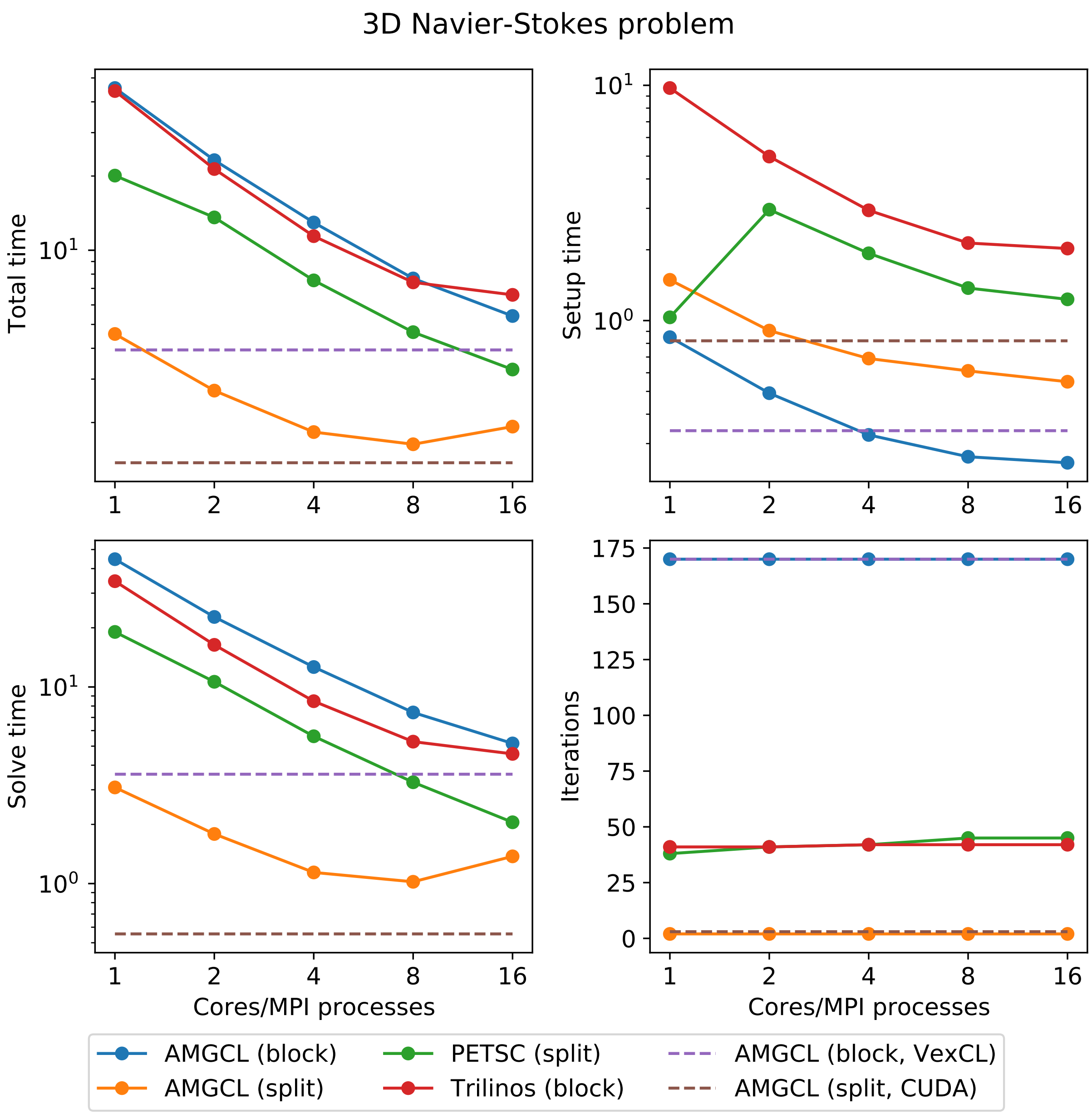
Here we demonstrate performance and scalability of the distributed memory algorithms provided by AMGCL on the example of a Poisson problem and a Navier-Stokes problem in a three dimensional space. To provide a reference, we compare performance of the AMGCL library with that of the well-established Trilinos ML package. The benchmarks were run on MareNostrum 4, PizDaint, and SuperMUC clusters which we gained access to via PRACE program (project 2010PA4058). The MareNostrum 4 cluster has 3456 compute nodes, each equipped with two 24 core Intel Xeon Platinum 8160 CPUs, and 96 GB of RAM. The peak performance of the cluster is 6.2 Petaflops. The PizDaint cluster has 5320 hybrid compute nodes, where each node has one 12 core Intel Xeon E5-2690 v3 CPU with 64 GB RAM and one NVIDIA Tesla P100 GPU with 16 GB RAM. The peak performance of the PizDaint cluster is 25.3 Petaflops. The SuperMUC cluster allowed us to use 512 compute nodes, each equipped with two 14 core Intel Haswell Xeon E5-2697 v3 CPUs, and 64 GB of RAM.

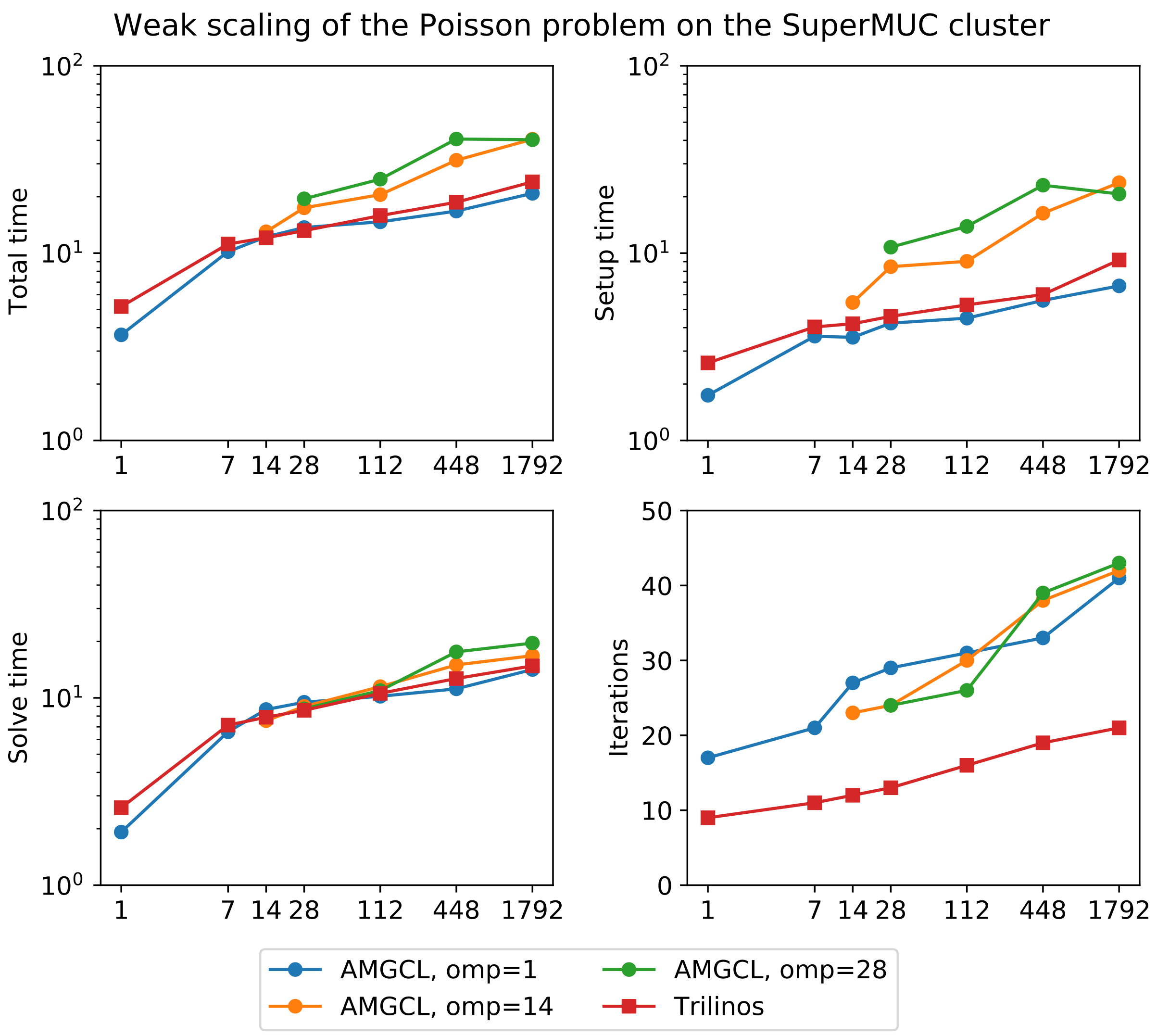
3D Poisson problem

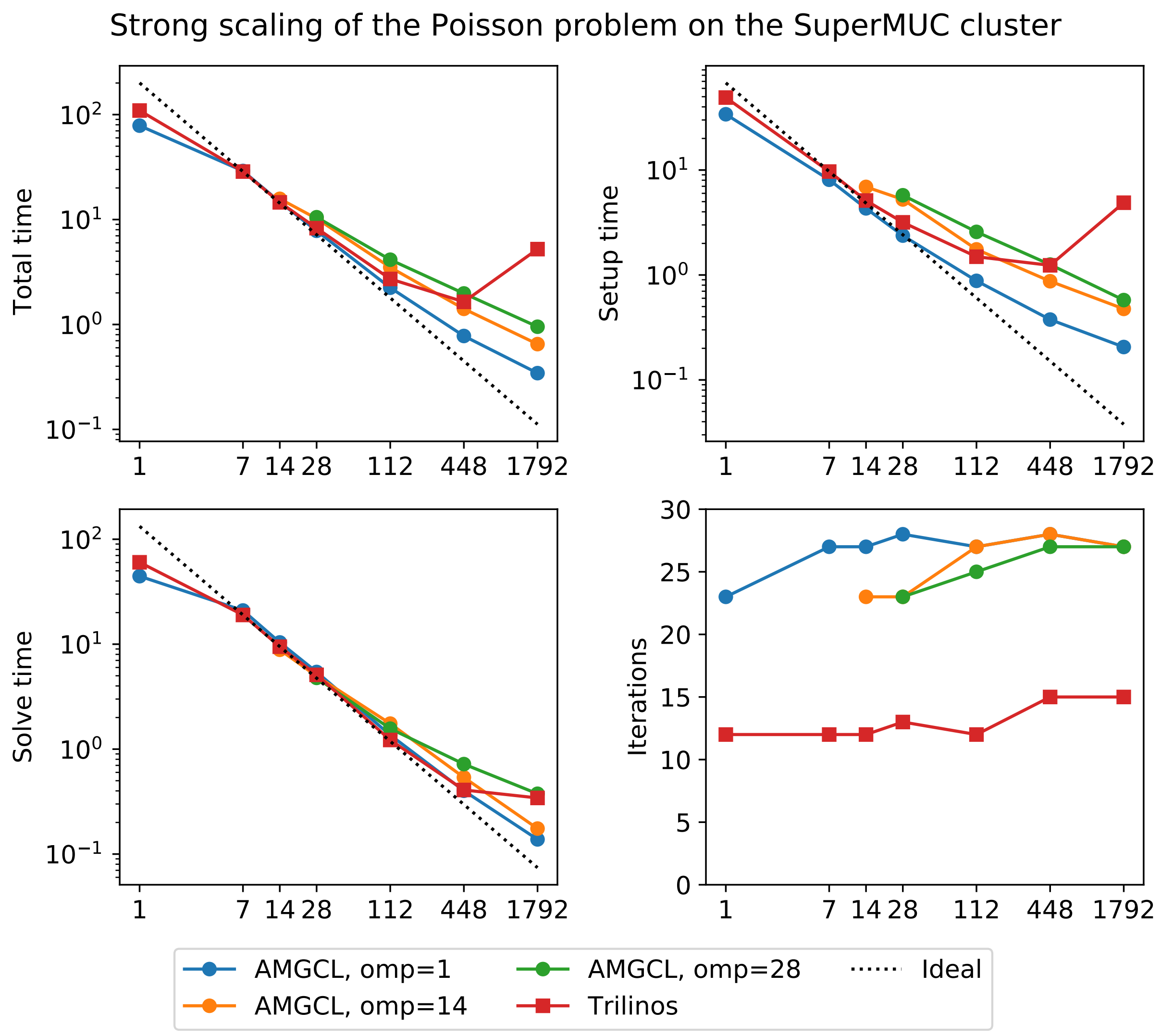
The figure below shows weak scaling of the solution on the SuperMUC cluster. Here the problem size is chosen to be proportional to the number of CPU cores with about 100^3 unknowns per core. Both AMGCL and Trilinos implementations use a CG iterative solver preconditioned with smoothed aggregation AMG. AMGCL uses SPAI(0) for the smoother, and Trilinos uses ILU(0), which are the corresponding defaults for the libraries. The plots in the figure show total computation time, time spent on constructing the preconditioner, solution time, and the number of iterations. The AMGCL library results are labelled 'OMP=n', where n=1,14,28 corresponds to the number of OpenMP threads controlled by each MPI process. The Trilinos library uses single-threaded MPI processes.

Next figure shows strong scaling results for smoothed aggregation AMG preconditioned on the SuperMUC cluster. The problem size is fixed to 256^3 unknowns and ideally the compute time should decrease as we increase the number of CPU cores. The case of ideal scaling is depicted for reference on the plots with thin gray dotted lines.

The AMGCL implementation uses a BiCGStab(2) iterative solver preconditioned with subdomain deflation, as it showed the best behaviour in our tests. Smoothed aggregation AMG is used as the local preconditioner. The Trilinos


3D Navier-Stokes problem
Total time
Setup time
Solve time
Iterations
Cores/MPI processes
Cores/MPI processes
1
2
4
8
16
10^1
10^0
175
150
125
100
75
50
25
0
AMGCL (block)
AMGCL (split)
PETSC (split)
Trilinos (block)
AMGCL (block, VexCL)
AMGCL (split, CUDA)


Weak scaling of the Poisson problem on the SuperMUC cluster
Total time
Setup time
Solve time
Iterations
10^0
10^1
10^2
0
10
20
30
40
50
1
7
14
28
112
448
1792
AMGCL, omp=1
AMGCL, omp=14
AMGCL, omp=28
Trilinos


Strong scaling of the Poisson problem on the SuperMUC cluster
Total time
Setup time
Solve time
Iterations
10^2
10^1
10^0
10^{-1}
1
7
14
28
112
448
1792
0
5
10
15
20
25
30
AMGCL, omp=1
AMGCL, omp=14
AMGCL, omp=28
Trilinos
Ideal

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.

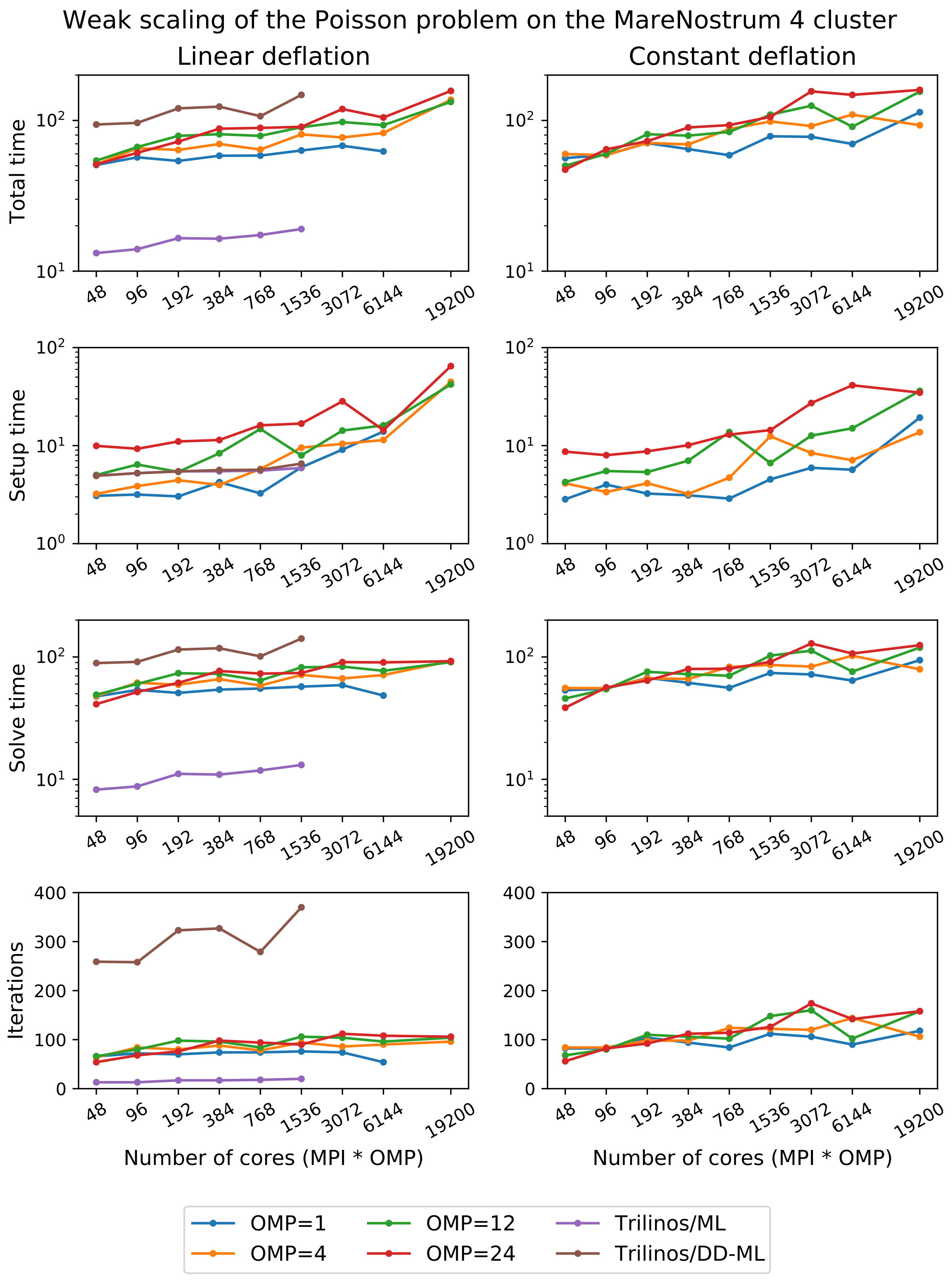
In the case of ideal scaling the timing plots on this figure would be strictly horizontal. This is not the case here: instead, we see that both AMGCL and Trilinos loose about 6-8% efficiency whenever the number of cores doubles. The AMGCL algorithm performs about three times worse that the AMG-based 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


Weak scaling of the Poisson problem on the MareNostrum 4 cluster
Linear deflation
Constant deflation
Total time
Setup time
Solve time
Iterations
Number of cores (MPI * OMP)
48
96
192
384
768
1536
3072
6144
19200
OMP=1
OMP=4
OMP=12
OMP=24
Trilinos/ML
Trilinos/DD-ML

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.

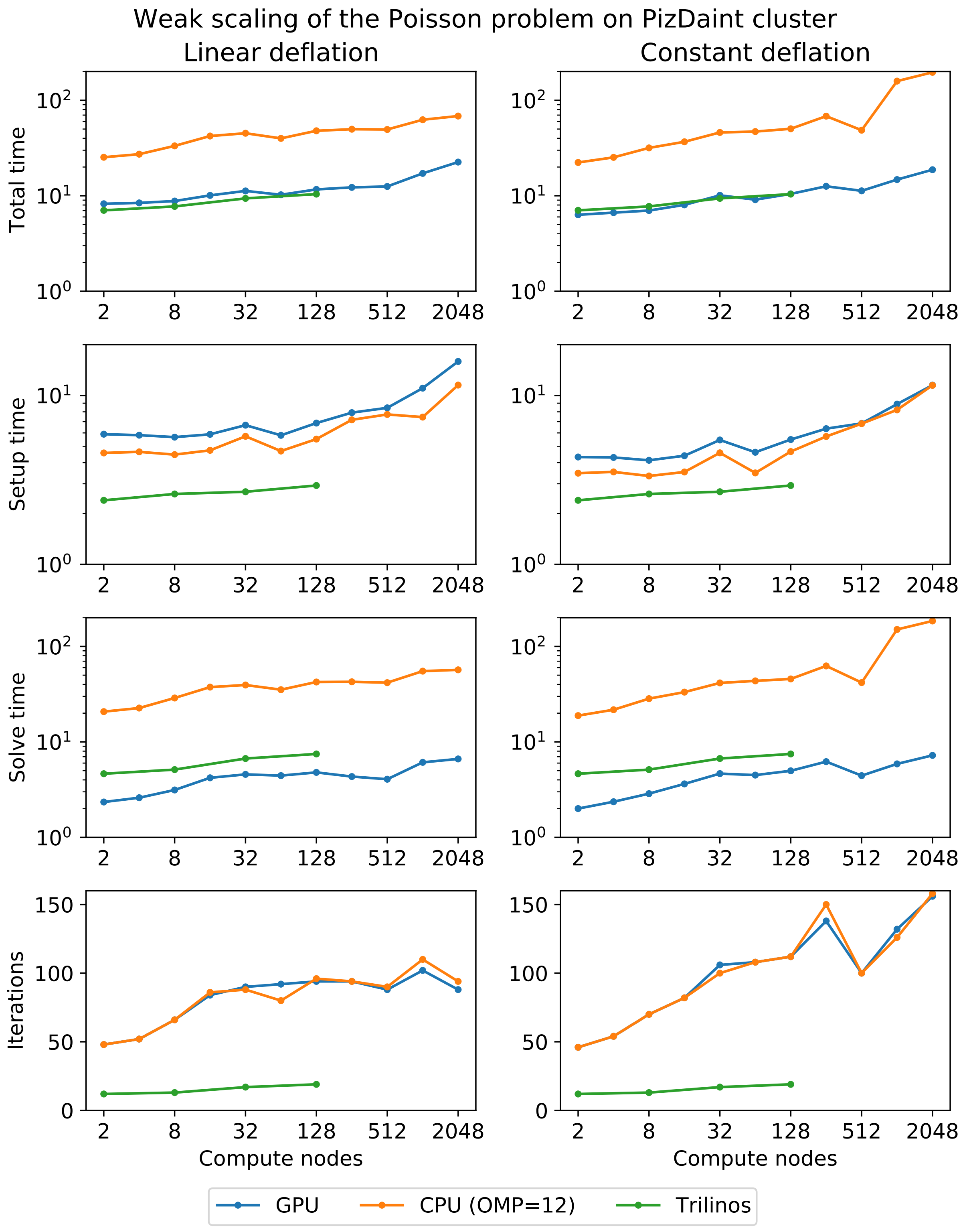
The following figure shows strong scaling results for the MareNostrum 4 cluster. The problem size is fixed to 512^3 unknowns and ideally the compute time should decrease as we increase the number of CPU cores. The case of ideal scaling is depicted for reference on the plots with thin gray dotted lines.

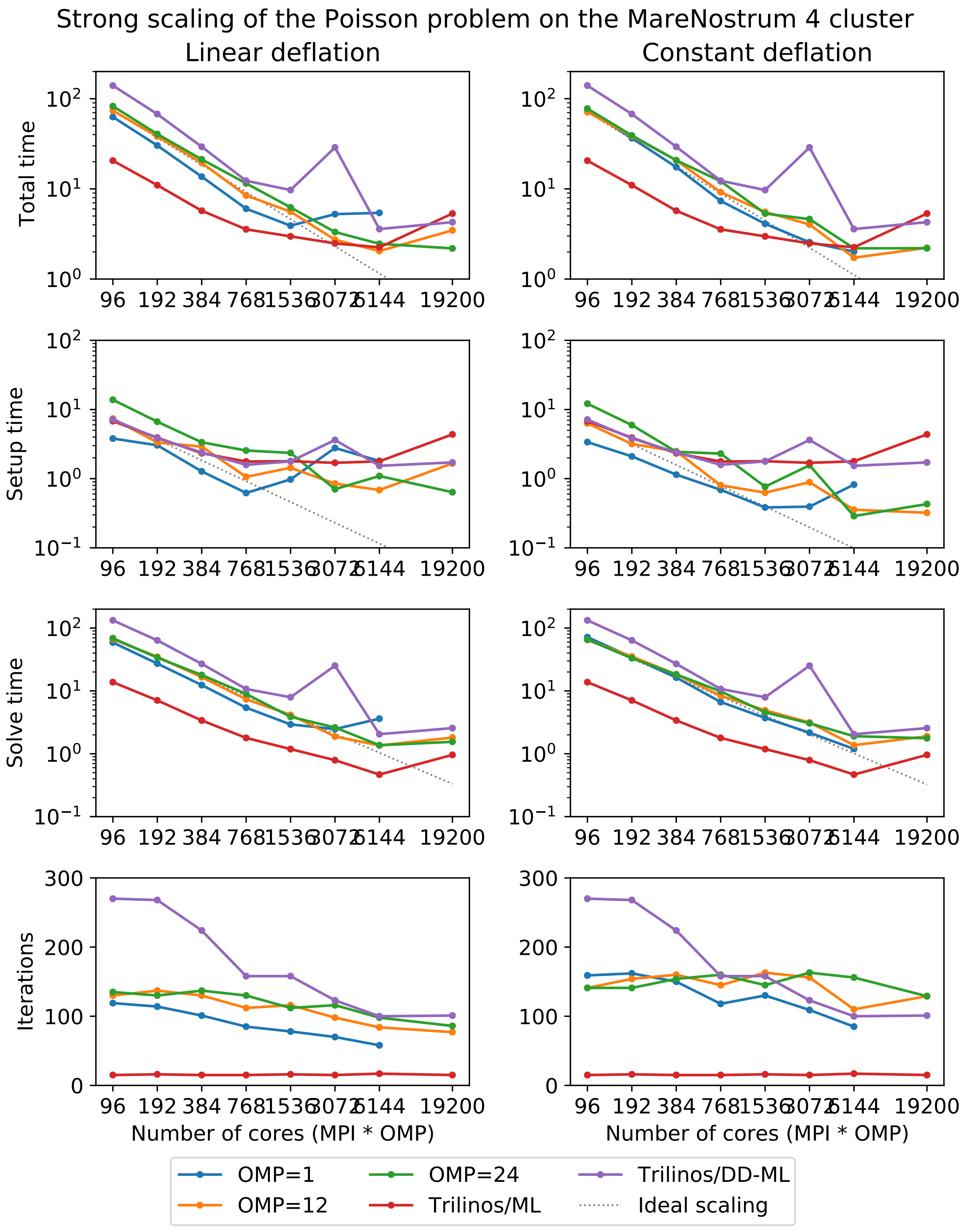
Here, AMGCL demonstrates scalability slightly better than that of the Trilinos ML package. At 384 cores the AMGCL solution for OMP=1 is about 2.5 times slower than Trilinos/AMG, and 2 times faster than Trilinos/DD-ML. As is expected for a strong scalability benchmark, the drop in scalability at higher 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’.

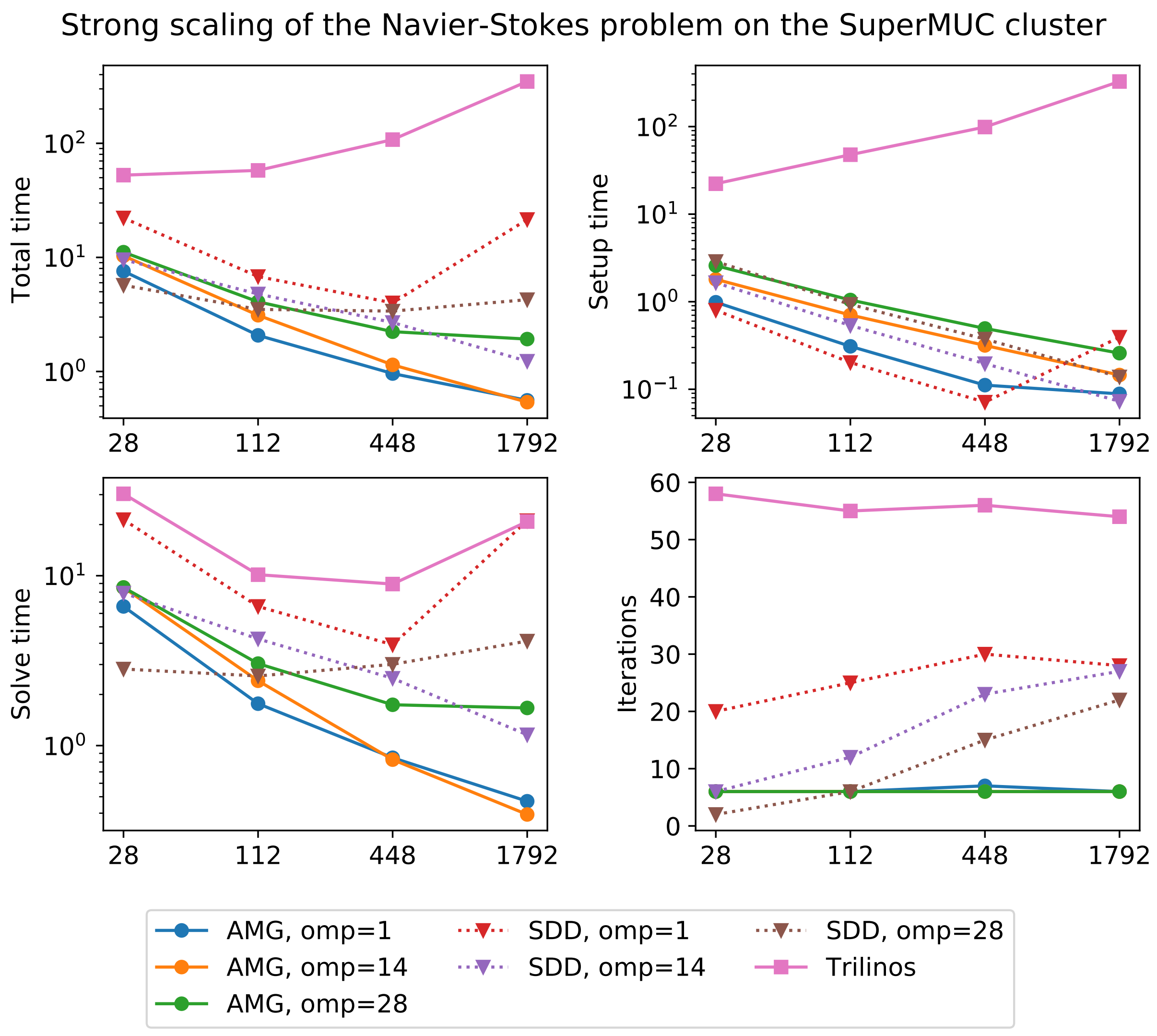

Weak scaling of the Poisson problem on PizDaint cluster
Linear deflation
Constant deflation
Total time
Setup time
Solve time
Iterations
Compute nodes
2
8
32
128
512
2048
10^0
10^1
10^2
0
50
100
150
GPU
CPU (OMP=12)
Trilinos


Strong scaling of the Poisson problem on the MareNostrum 4 cluster
Linear deflation
Constant deflation
Total time
Setup time
Solve time
Iterations
Number of cores (MPI * OMP)
96
192
384
768
1536
3072
6144
19200
OMP=1
OMP=12
OMP=24
Trilinos/ML
Trilinos/DD-ML
Ideal scaling

3D Navier-Stokes problem

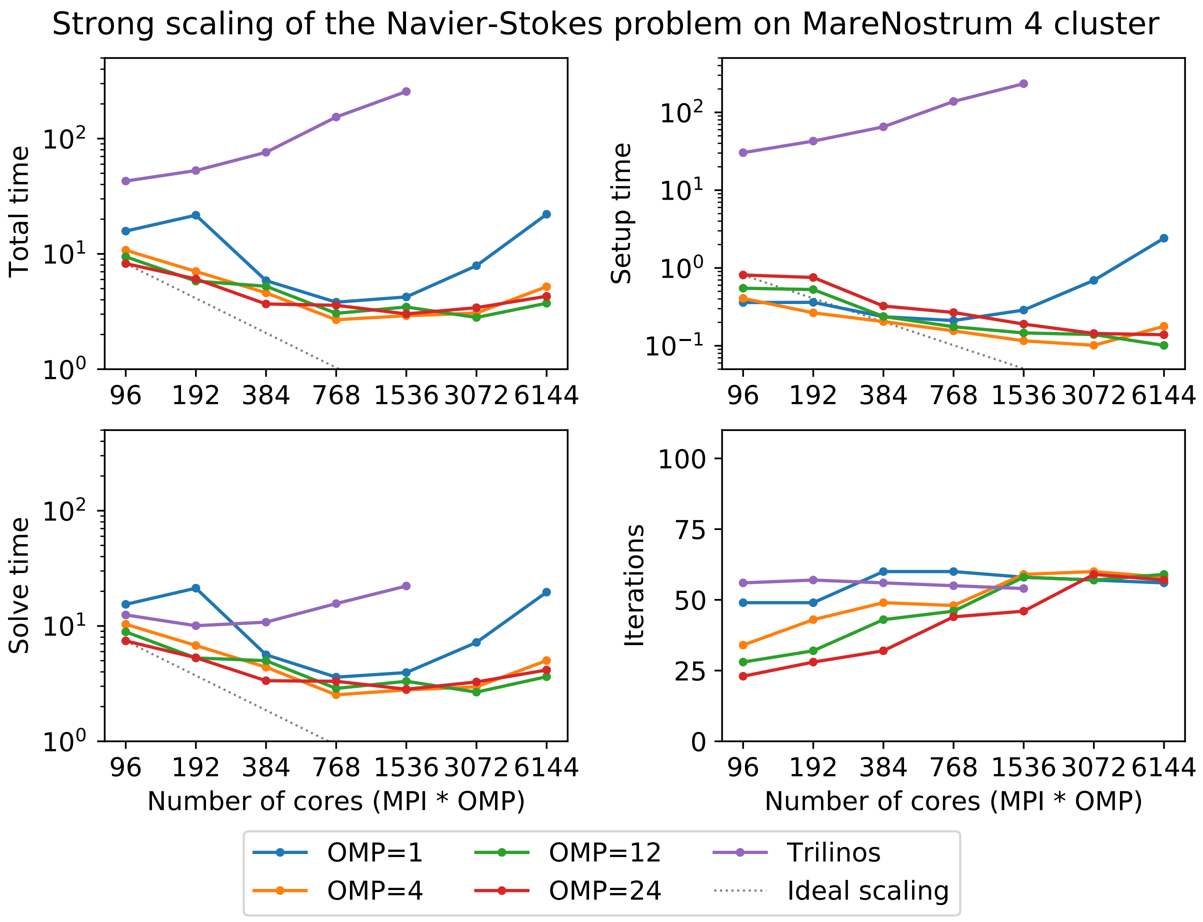
The system matrix in these tests contains 4773588 unknowns and 281089456 nonzeros. The assembled system is available to download at https://doi.org/10.5281/zenodo.1231961. AMGCL library uses field-split approach with the `mpi::schur_pressure_correction` preconditioner. Trilinos ML does not provide field-split type preconditioners, and uses the nonsymmetric smoothed aggregation variant (NSSA) applied to the monolithic problem. Default NSSA parameters were employed in the tests.

The figure below shows scalability results for the Navier-Stokes problem on the SuperMUC cluster. In case of AMGCL, the pressure part of the system is preconditioned with a smoothed aggregation AMG. Since we are solving a fixed-size problem, this is essentially a strong scalability test.



The next figure shows scalability results for the Navier-Stokes problem on the MareNostrum 4 cluster. Since we are solving a fixed-size problem, this is essentially a strong scalability test.

Both AMGCL and ML preconditioners deliver a very flat number of iterations with growing number of MPI processes. As expected, the field-split preconditioner pays off and performs better than the monolithic approach in the solution of the problem. Overall the AMGCL implementation shows a decent, although less than optimal parallel scalability. This is not unexpected since the problem size quickly becomes too little to justify the use of more parallel resources (note that at 192 processes, less than 25000 unknowns are assigned to each MPI subdomain). Unsurprisingly, in this


Strong scaling of the Navier-Stokes problem on MareNostrum 4 cluster
Total time
Setup time
Solve time
Iterations
Number of cores (MPI * OMP)
96
192
384
768
1536
3072
6144
10^0
10^1
10^2
10^{-1}
0
25
50
75
100
OMP=1
OMP=4
OMP=12
OMP=24
Trilinos
Ideal scaling

context the use of OpenMP within each domain pays off and allows delivering a greater level of scalability.

2.7 Bibliography

Bibliography

[FGJT10] M. Ferronato, G. Gambolati, C. Janna, P. Teatini. "Geomechanical issues of anthropogenic CO2 sequestration in exploited gas fields", Energy Conversion and Management, 51, pp. 1918-1928, 2010.

[FePG09] M. Ferronato, G. Pini, and G. Gambolati. The role of preconditioning in the solution to FE coupled consolidation equations by Krylov subspace methods. International Journal for Numerical and Analytical Methods in Geomechanics 33 (2009), pp. 405-423.

[FeJP12] M. Ferronato, C. Janna, and G. Pini. Parallel solution to ill-conditioned FE geomechanical problems. International Journal for Numerical and Analytical Methods in Geomechanics 36 (2012), pp. 422-437.

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

[Alex00] A. Alexandrescu, Modern C++ design: generic programming and design patterns applied, AddisonWesley, 2001.

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

[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.

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

[BeGL05] Benzi, Michele, Gene H. Golub, and Jörg Liesen. Numerical solution of saddle point problems. Acta numerica 14 (2005): 1-137.

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

[BrMH85] Brandt, A., McCormick, S., & Huge, J. (1985). Algebraic multigrid (AMG) for sparse matrix equations. Sparsity and its Applications, 257.

[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.

[ChPa15] Chow, Edmond, and Aftab Patel. Fine-grained parallel incomplete LU factorization. SIAM journal on Scientific Computing 37.2 (2015): C169-C193.

[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.

[Demi19] Demidov, Denis. AMGCL: An efficient, flexible, and extensible algebraic multigrid implementation. Lobachevskii Journal of Mathematics 40.5 (2019): 535-546.

[DeMW20] D. Demidov, L. Mu, and B. Wang. Accelerating linear solvers for Stokes problems with C++ metaprogramming. arXiv preprint arXiv:2006.06052 (2020).

[ElHS08] Elman, Howard, et al. A taxonomy and comparison of parallel block multi-level preconditioners for the incompressible Navier–Stokes equations. Journal of Computational Physics 227.3 (2008): 1790-1808.

[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.

[GhKK12] P. Ghysels, P. Kłosiewicz, and W. Vanroose. Improving the arithmetic intensity of multigrid with the help of polynomial smoothers. Numer. Linear Algebra Appl. 2012;19:253-267.

[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.

[GmHJ15] Gmeiner, Björn, et al. A quantitative performance study for Stokes solvers at the extreme scale. Journal of Computational Science 17 (2016): 509-521.

[Meye05] S. Meyers, Effective C++: 55 specific ways to improve your programs and designs, Pearson Education, 2005.

[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.

[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.

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

[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.

[ViBo92] Vincent, C., and R. Boyer. "A preconditioned conjugate gradient Uzawa-type method for the solution of the Stokes problem by mixed Q1–P0 stabilized finite elements." International journal for numerical methods in fluids 14.3 (1992): 289-298.