AMGCL Documentation

Release 1.3.99

Denis Demidov

Jul 14, 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. doi pdf

```
@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"
}
```

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

AMGCL uses the compile-time policy-based design approach, which allows users of the library to compose their own version of the AMG method from the provided components. This also allows for easily extending the library when required.

2.2.1 Components

AMGCL provides the following components:

- **Backends** – classes that define matrix and vector types and operations necessary during the solution phase of the algorithm. When an AMG hierarchy is constructed, it is moved to the specified backend. The approach enables transparent acceleration of the solution phase with OpenMP, OpenCL, or CUDA technologies, and also makes tight integration with user-defined data structures possible.
- **Value types** – enable transparent solution of complex or non-scalar systems. Most often, a value type is simply a `double`, but it is possible to use small statically-sized matrices as value type, which may increase cache-locality, or convergence ratio, or both, when the system matrix has a block structure.
- **Matrix adapters** – allow AMGCL to construct a solver from some common matrix formats. Internally, the CRS format is used, but it is easy to adapt any matrix format that allows row-wise iteration over its non-zero elements.
- **Coarsening strategies** – various options for creating coarse systems in the AMG hierarchy. A coarsening strategy takes the system matrix A at the current level, and returns prolongation operator P and the corresponding restriction operator R.
- **Relaxation methods** – or smoothers, that are used on each level of the AMG hierarchy during solution phase.
- **Preconditioners** – aside from the AMG, AMGCL implements preconditioners for some common problem types. For example, there is a Schur complement pressure correction preconditioner for Navie-Stokes type problems, or CPR preconditioner for reservoir simulations. Also, it is possible to use single level relaxation method as a preconditioner.
- **Iterative solvers** – Krylov subspace methods that may be combined with the AMG (or other) preconditioners in order to solve the linear system.

To illustrate this, here is an example of defining a solver type that combines a BiCGStab iterative method [Barr94] preconditioned with smoothed aggregation AMG that uses SPAI(0) (sparse approximate inverse smoother) [BrGr02] as relaxation. The solver uses the `amgcl::backend::builtin` backend (accelerated with OpenMP), and double precision scalars as value type.

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

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

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

2.2.2 Parameters

Each 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, you can easily lookup parameter definitions and set parameters for individual components. For example, we can set the desired tolerance and maximum number of iterations for the iterative solver in the above example like this:

```
Solver::params prm;
prm.solver.tol = 1e-3;
prm.solver.maxiter = 10;
Solver solve( std::tie(n, ptr, col, val), prm );
```

2.2.3 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>
```

(continues on next page)

(continued from previous page)

```
#include <amgcl/relaxation/runtime.hpp>
#include <amgcl/solver/runtime.hpp>

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

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

boost::property_tree::ptree prm;

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

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

2.3 Components

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:

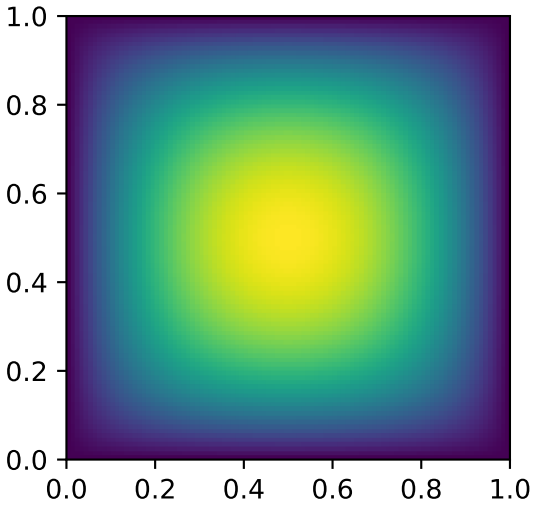
```
boost::property_tree::ptree prm;
prm.put("solver.tol", 1e-3);
Solver solve( std::tie(n, ptr, col, val), prm );
```

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:

```
#include <vector>

// Assembles matrix for Poisson's equation with homogeneous
// boundary conditions on a n x n grid.
```

(continues on next page)

(continued from previous page)

```
// 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:

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

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

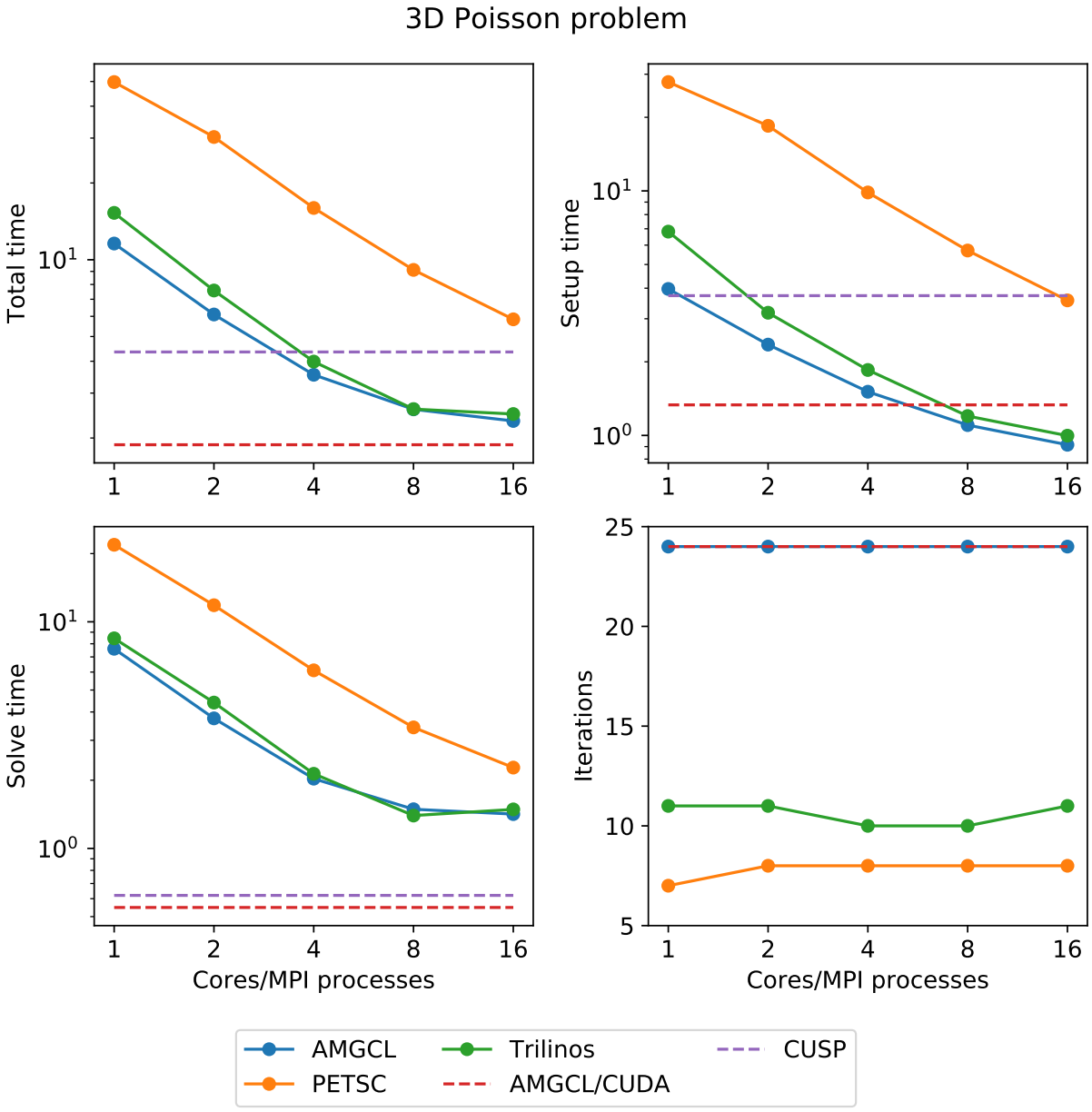
In this section we test performance of the library on a shared memory system. We also compare the results with PETSC and Trilinos ML distributed memory libraries and CUSP GPGPU library. The tests were performed on a dual socket system with two Intel Xeon E5-2640 v3 CPUs. The system also had an NVIDIA 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
Cores/MPI processes
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

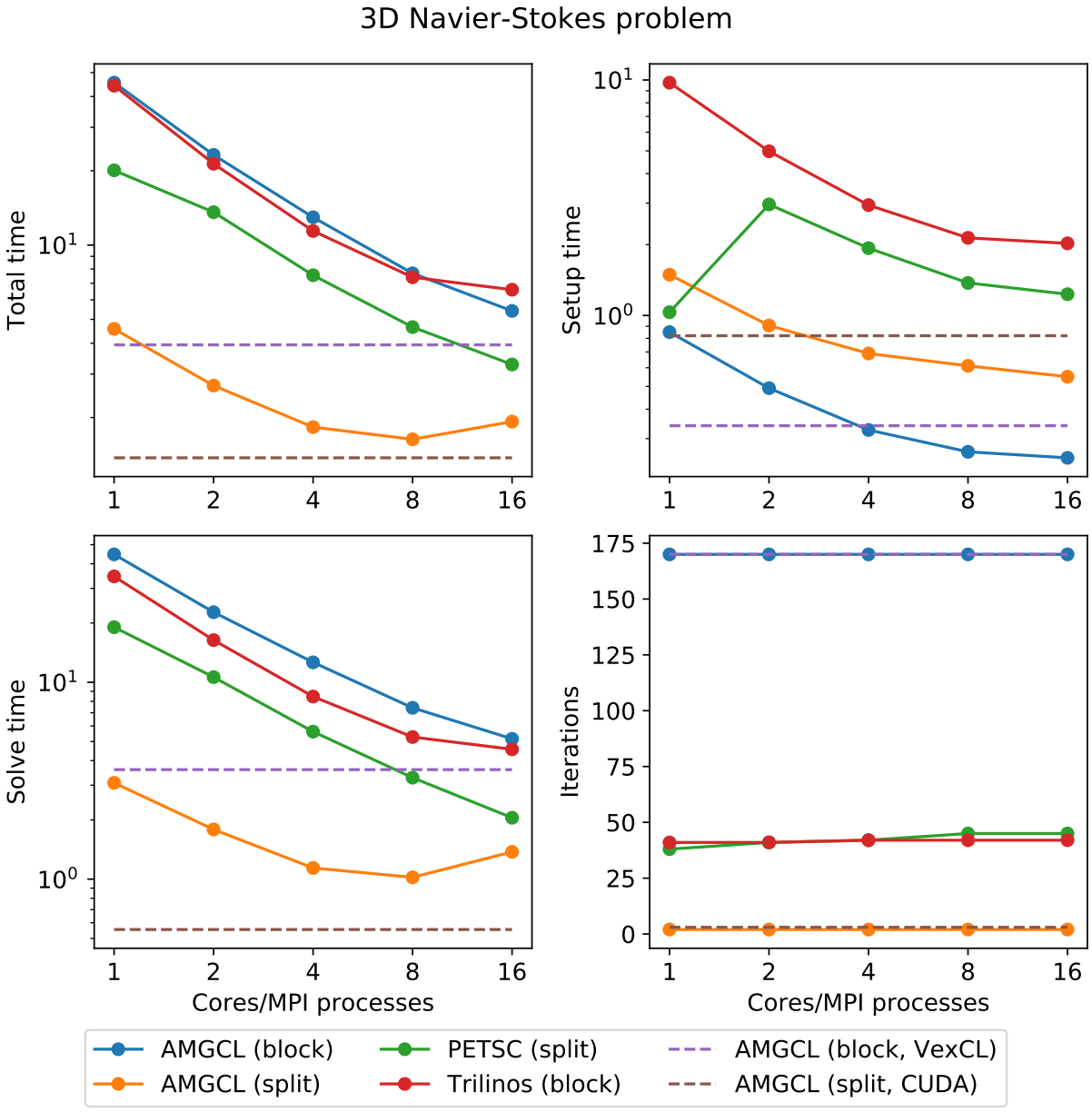
Here we demonstrate performance and scalability of the distributed memory algorithms provided by AMGCL on the example of a Poisson problem and a Navier-Stokes problem in a three dimensional space. To provide a reference, we compare performance of the AMGCL library with that of the well-established Trilinos 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.

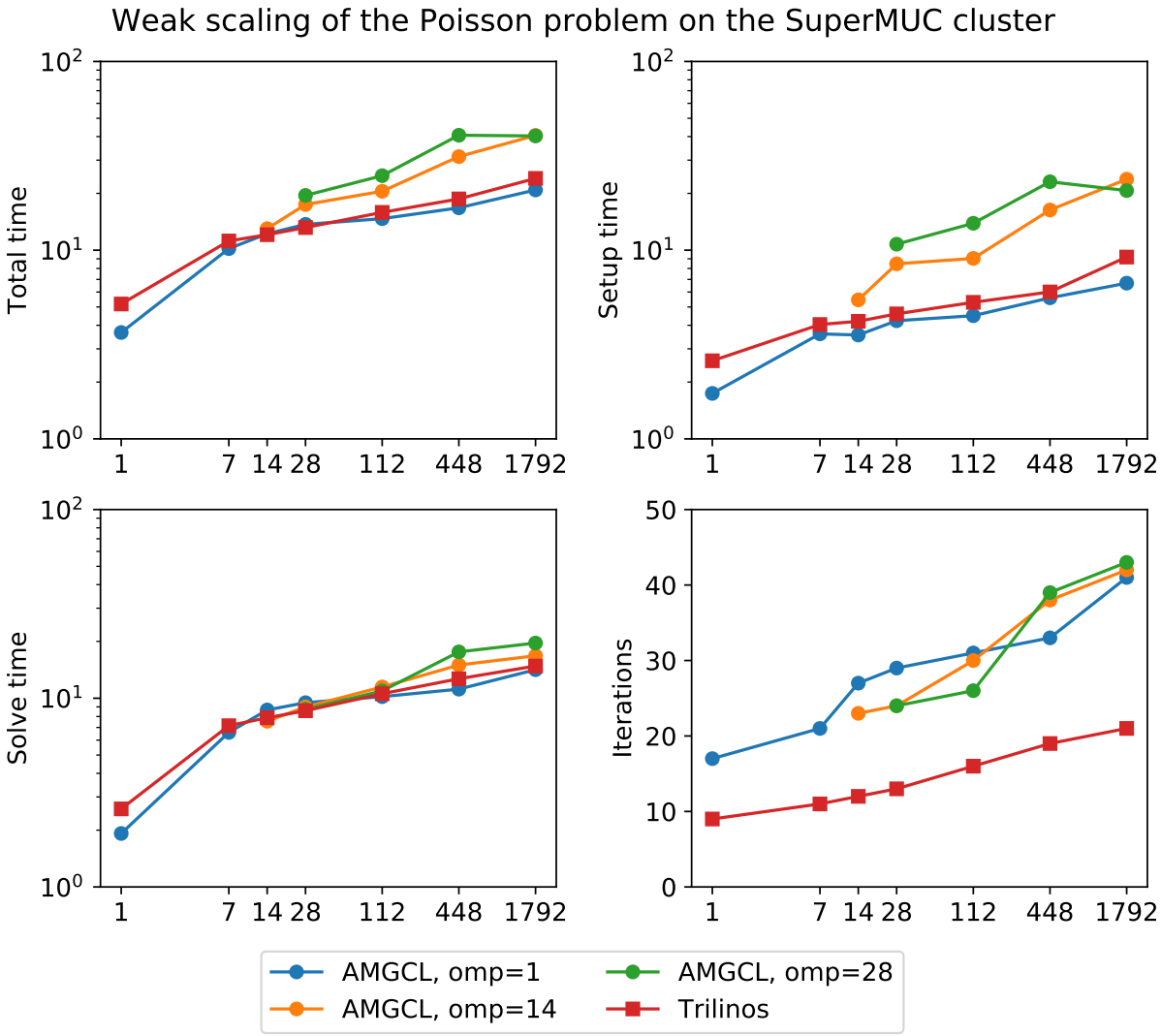
3D Poisson problem

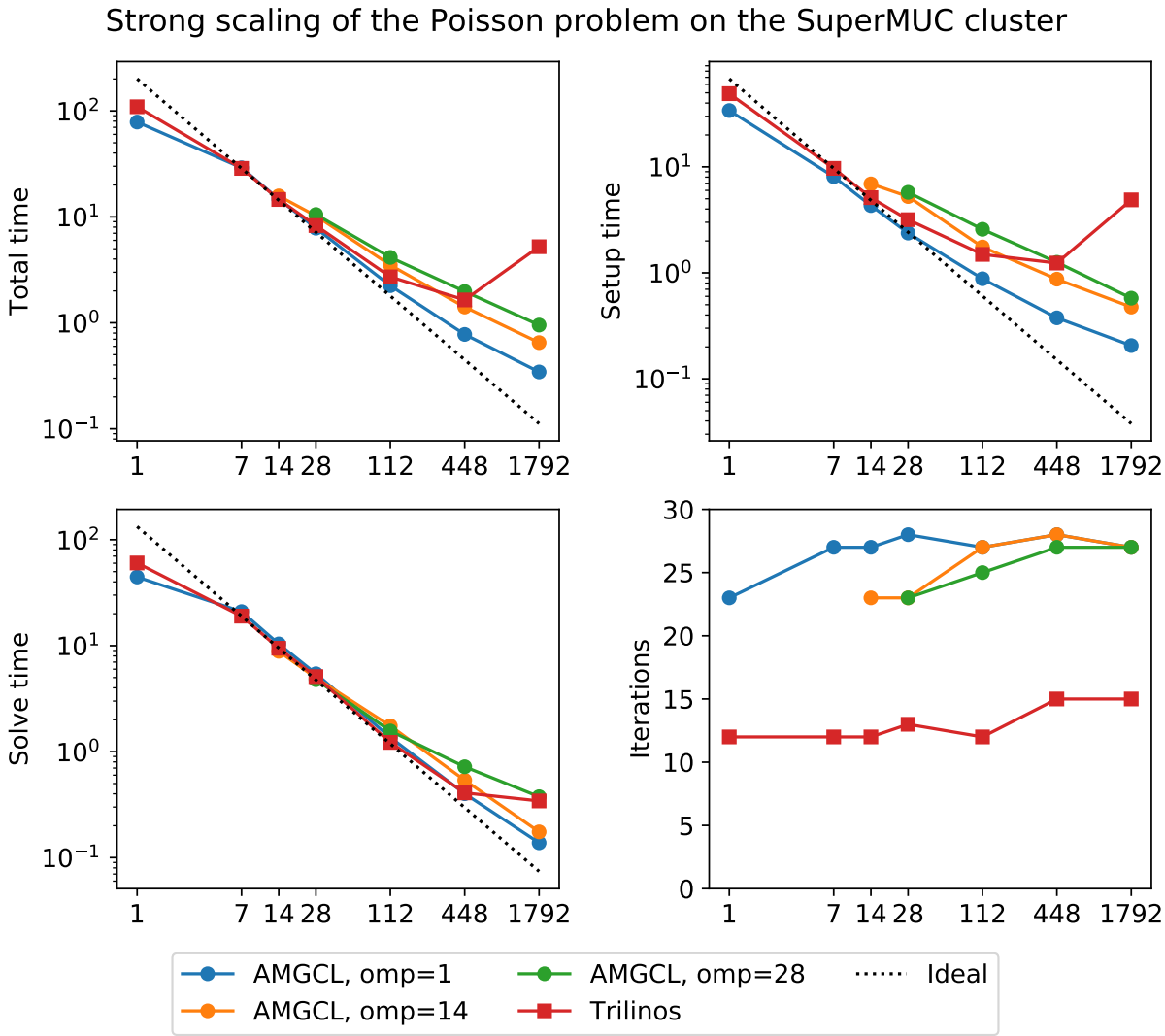
The figure below shows weak scaling of the solution on the SuperMUC cluster. Here the problem size is chosen to be proportional to the number of CPU cores with about 100^3 unknowns per core. Both AMGCL and Trilinos implementations use a CG iterative solver preconditioned with smoothed aggregation AMG. 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
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
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
1
7
14
28
112
448
1792
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.

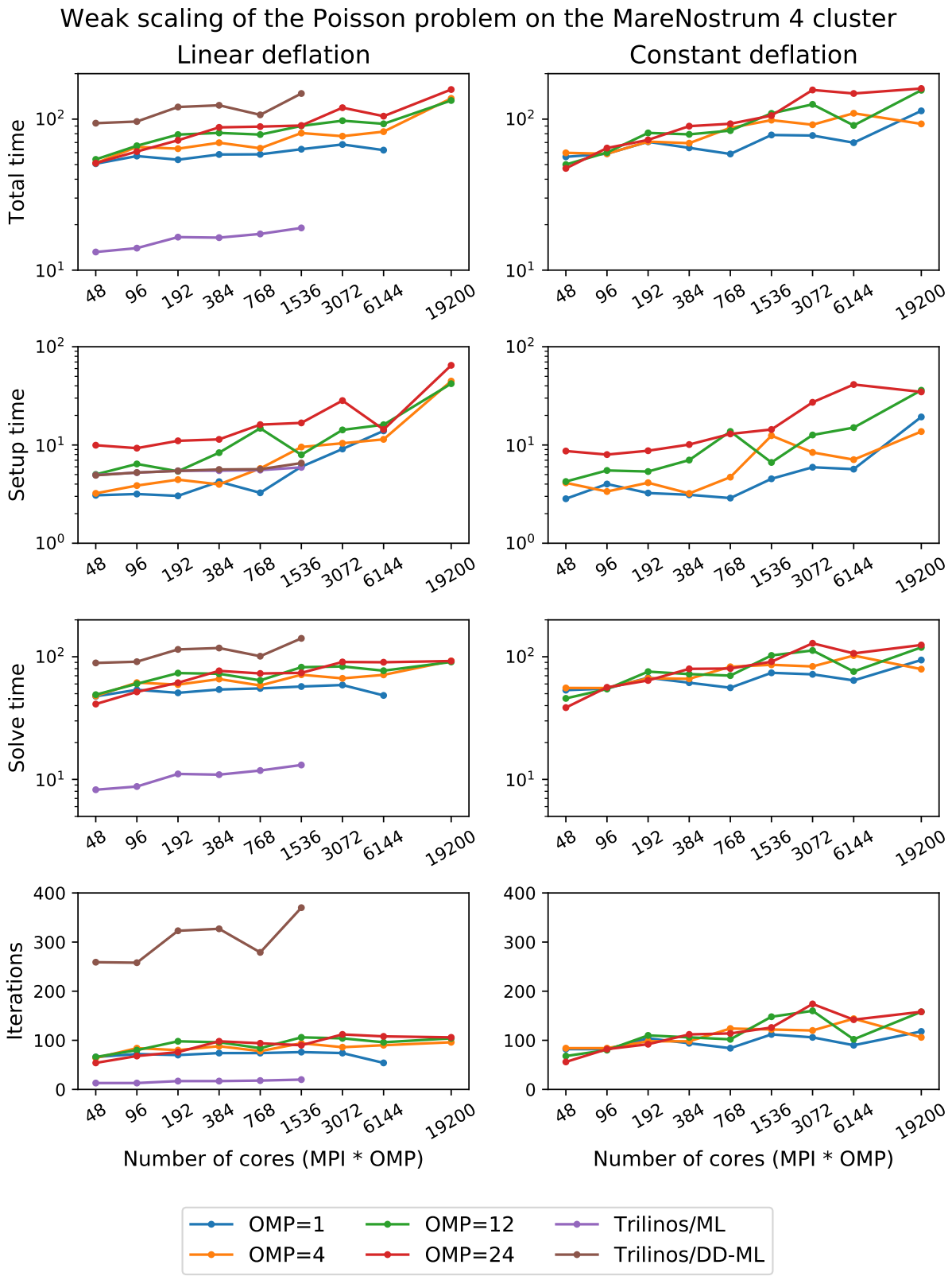
In the case of ideal scaling the timing plots on this figure would be strictly horizontal. This is not the case here: instead, we see that both AMGCL and Trilinos loose about 6-8% efficiency whenever the number of cores doubles. The AMGCL algorithm performs about three times worse that the AMG-based 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
48
96
192
384
768
1536
3072
6144
19200
Number of cores (MPI * OMP)
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.

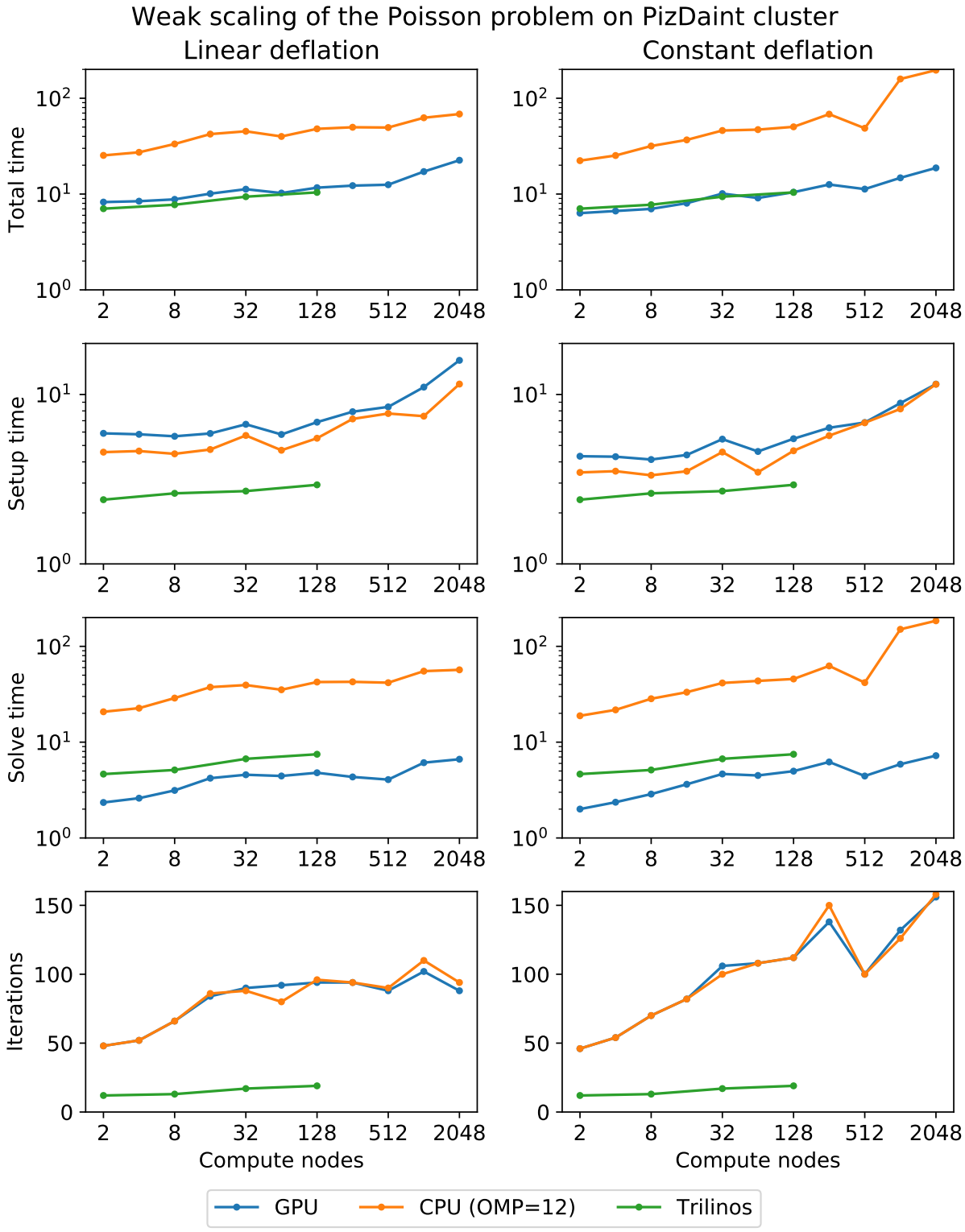
The following figure shows strong scaling results for the MareNostrum 4 cluster. The problem size is fixed to 512^3 unknowns and ideally the compute time should decrease as we increase the number of CPU cores. The case of ideal scaling is depicted for reference on the plots with thin gray dotted lines.

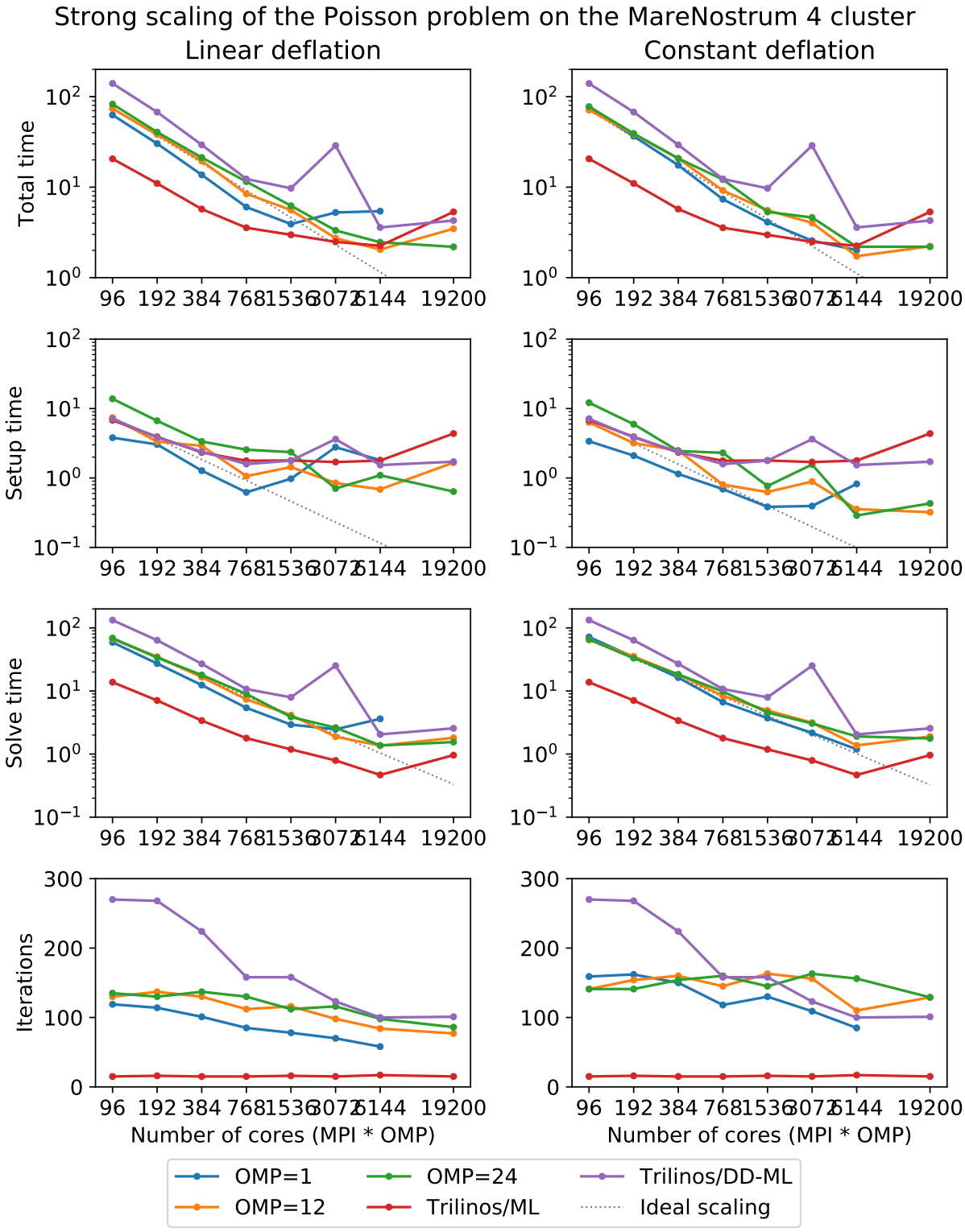
Here, AMGCL demonstrates scalability slightly better than that of the Trilinos ML package. At 384 cores the AMGCL solution for OMP=1 is about 2.5 times slower than Trilinos/AMG, and 2 times faster than Trilinos/DD-ML. As is expected for a strong scalability benchmark, the drop in scalability at higher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

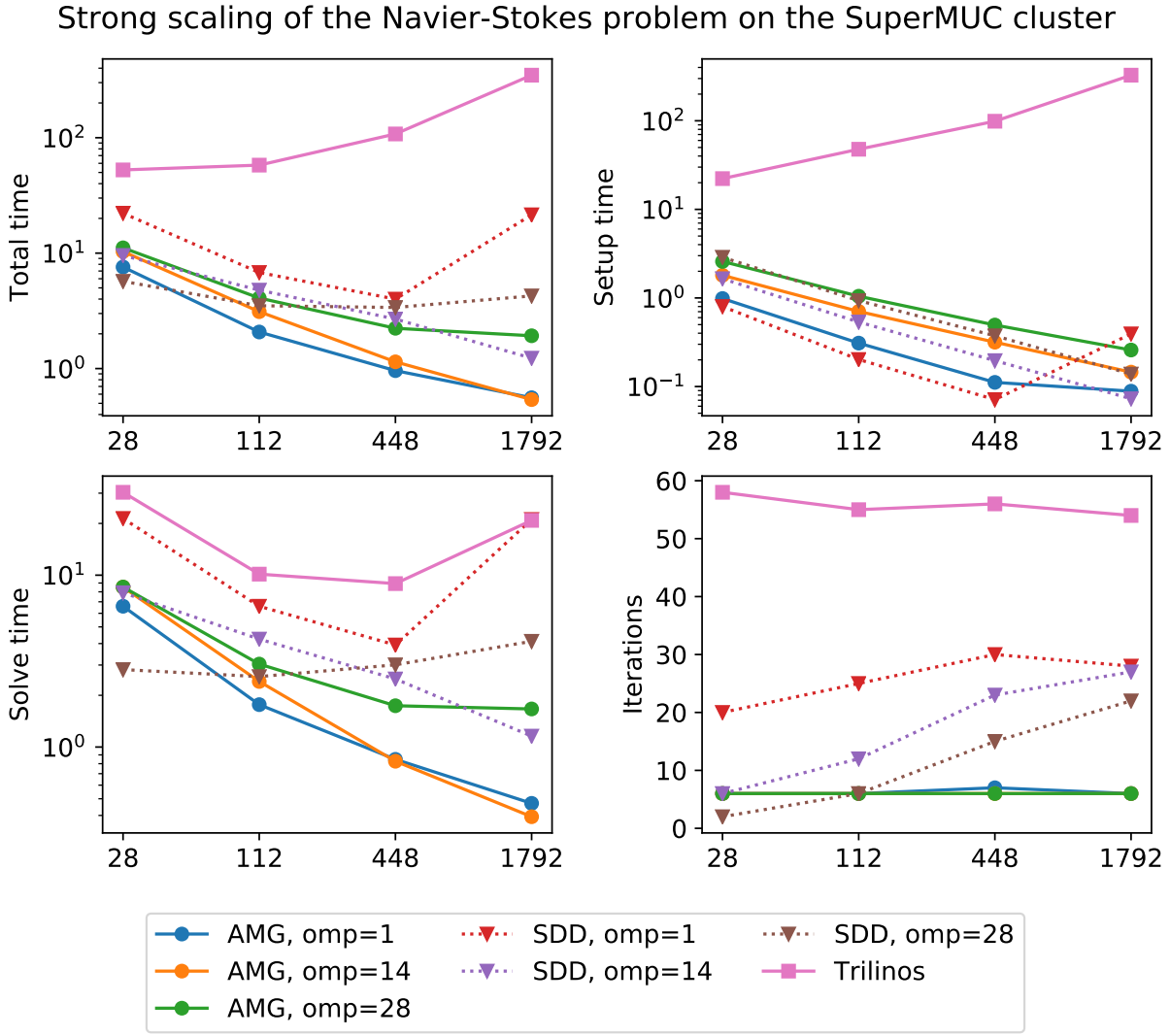

Weak scaling of the Poisson problem on PizDaint cluster
Linear deflation
Constant deflation
Total time
Setup time
Solve time
Iterations
Compute nodes
Compute nodes
GPU
CPU (OMP=12)
Trilinos


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

3D Navier-Stokes problem

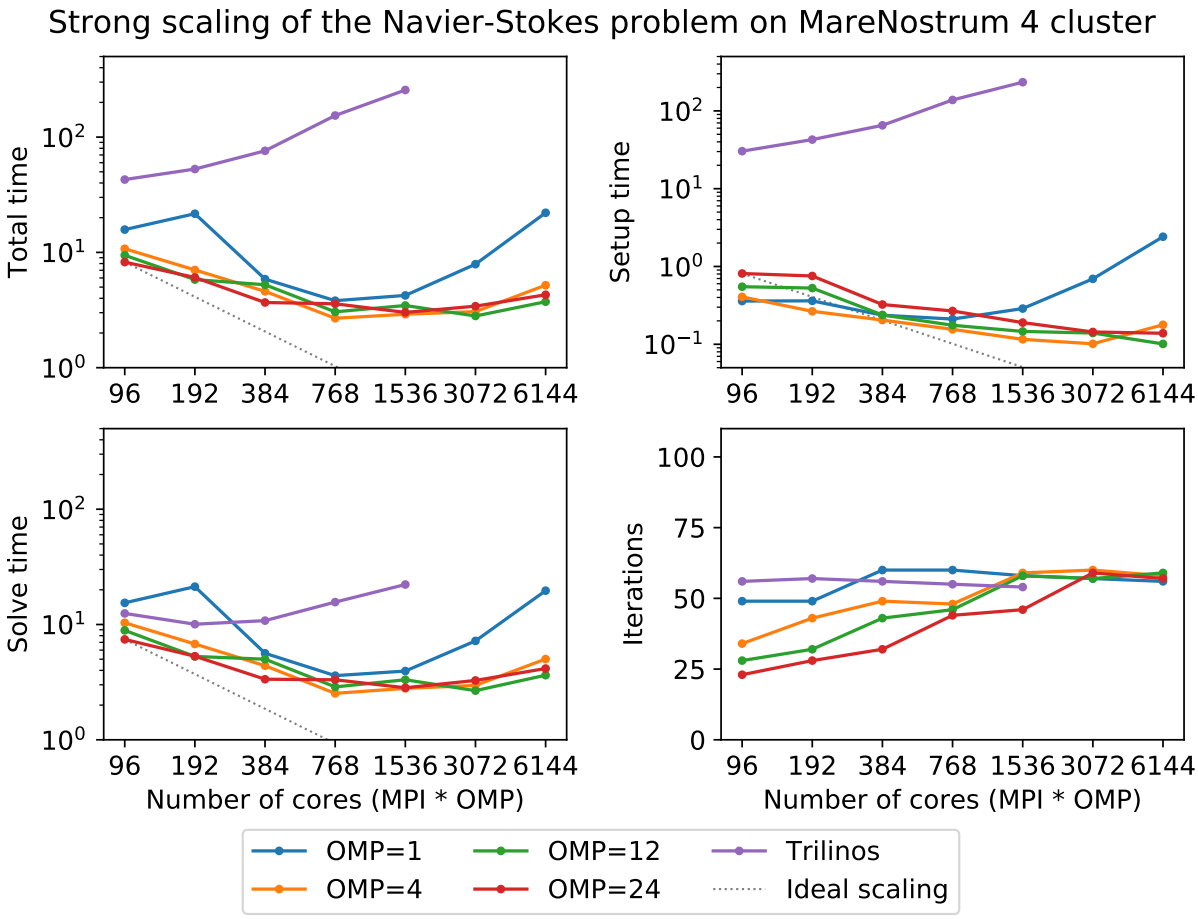
The system matrix in these tests contains 4773588 unknowns and 281089456 nonzeros. The assembled system is available to download at https://doi.org/10.5281/zenodo.1231961. AMGCL library uses field-split approach with the `mpi::schur_pressure_correction` preconditioner. Trilinos ML does not provide field-split 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
96 192 384 768 1536 3072 6144
Number of cores (MPI * OMP)
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

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

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

[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.

[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.

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

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

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

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

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

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

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

[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.