EXPLORING UNSTRUCTURED POISSON SOLVERS FOR FDS

Dr. Susanne Kilian

hhpberlin, Ingenieure für Brandschutz GmbH, 10245 Berlin
s.kilian@hhpberlin.de

Abstract. Many presently employed Poisson solvers which offer a sufficient amount of computational efficiency and robustness are limited to the use of regular geometries and corresponding grid structures. This restriction can affect the precise representation of complex fire scenarios involving complex bodies and flow obstructions. Usually, their parallel application in the context of multi-core architectures contributes to a further impairment. The current solver for the Poisson equation in FDS is based on the use of local FFT methods on the single meshes of the underlying domain decomposition. This approach has proven to be computationally efficient and accurate in a multitude of cases. But due to its restriction to rectilinear meshes and its purely local character, there are two possible drawbacks, namely the presence of velocity errors based on penetrations into immersed obstacles as well as possibly large velocity errors along mesh interfaces. In order to face this challenge several alternative Poisson solvers of direct and iterative type are examined which basically apply global solution strategies spanning over the whole domain decomposition. Furthermore, their ability to deal with unstructured grids along with the exact setting of boundary conditions on internal obstacle surfaces shall be analyzed. The paper and its associated talk are intended to give some insights into the current state of development and to compare the pros and cons of the different Poisson approaches with respect to their efficiency and accuracy.

1. MOTIVATION

The efficient solution of the Poisson equation for the pressure is an essential part in many CFD codes for the simulation of realistic flow phenomena. There are two main classes of numerical solvers for the Poisson equation, namely *direct* and *iterative* ones, with various representatives each. Both classes differ comprehensively in view of the underlying algorithmic approach which strongly influences their applicability and efficiency for different types of problems.

The numerical simulation of fire scenarios with respect to their high geometric complexity and extreme computational requirements represents a particular challenge when selecting a suitable pressure solver. This also holds true for FDS where the different choices are associated with several advantages and disadvantages which have to be weighed carefully against each other.

A basic design feature of FDS for the representation of the underlying geometry is the use of one or more cubic blocks (meshes) based on rectilinear grid decompositions each. Objects like walls or other obstacles internal to these meshes are mapped as rectangular shapes conforming with the underlying mesh grid.

Currently, the solution of the FDS Poisson equation is achieved by a representative of the direct Poisson solvers, namely a highly optimized blockwise *Fast Fourier Transformation* (FFT) solver which is part of the solver package CRAYFISHPAK [1]: On every single mesh a local FFT is performed which produces a correct solution (up to machine precision) to the related local Poisson problem. Then, the local solutions are clustered together to a global one by means of an interconnecting process along the mesh interfaces. This strategy has proven to be extremely fast and robust for many different constellations.

But the use of the FFT-solver is basically restricted to structured meshes. Therefore, all cells internal to immersed obstacles must be included into the system of equations which may lead to non-zero normal components of velocity at internal obstacles even in case of solid boundaries with no mass transfer.

In order to compensate this 'penetration' an additional iterative *Direct Forcing Immersed Boundary Method* [2] is used. In every time step it requires the repeated blockwise FFT-solution of the Poisson equation until a specified tolerance for the velocity error along internal objects has been reached.

A similar strategy comes into effect along mesh interfaces, because the mentioned interconnection process namely produces a continuous global Poisson solution but isn't able to guarantee that the normal components of velocity match there, too. In the worst case this 'velocity-correction' method may converge slowly along with a comprehensible increase of computational costs.

Finally, the purely blockwise execution of FFT methods doesn't take sufficient account of the intrinsically global character of the physical pressure which spreads local information immediately over the whole domain. In case of big geometries with many meshes (i.e. tunnels) and/or transient boundary conditions this strategy may experience difficulties to reproduce this strong overall data coupling fast enough.

In order to get rid of these drawbacks, some other Poisson solvers are taken into account for the solution of the FDS pressure equation:

- a direct solver, namely the shared-memory multiprocessing *Parallel Direct Sparse Solver* (Pardiso) of the Intel Math Kernel Library (MKL), and its version for *Cluster Interfaces* (Cluster_Sparse_Solver),
- some iterative solvers based on *Scalable Recursive Clustering* (ScaRC) which combine techniques of *Conjugated Gradient Methods* (CG) and *Geometric Multigrid Methods* (GMG) with optimized preconditioning or smoothing.

Both variants are able to scope with unstructured meshes such that cells internal to obstacles can now be omitted from the system of equations and the correct internal boundary conditions can be set. Furthermore, they both manage to compute the correct normal components of velocity along the mesh interfaces. In total, there is no need to apply additional velocity-correction procedures any more.

However, the computational costs for all considered variants may be far higher compared to the current FFT-solver depending on the underlying constellation. Therefore, the proper choice of the Poisson solver for a given application requires a sensible consideration of the specific advantages and disadvantages. To provide a basic overview, the different solution techniques will be explained in more detail and analyzed with respect to their accuracy and scalability properties as well as their computational costs below.

2. DIFFERENT POISSON SOLVERS

2.1. The Poisson equation for the Pressure

Based on its non-conservative formulation, the momentum equation can be simplified by a series of substitutions to

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{F} + \nabla \mathcal{H} = 0 \quad ; \quad \mathbf{F} = \mathbf{F}_\mathrm{A} + \mathbf{F}_\mathrm{B}\,. \tag{1}$$

Taking the divergence of equation (1) the pressure equation of FDS reads as

$$\nabla^2 \mathcal{H} = -\frac{\partial(\nabla \cdot \mathbf{u})}{\partial t} - \nabla \cdot \mathbf{F}\,. \tag{2}$$

The pressure term $\mathcal{H} \equiv |\mathbf{u}|^2/2 + \tilde{p}/\rho$ includes the velocity field $\mathbf{u}$, the density ρ as well as the perturbation pressure $\tilde{p}$ by which the fluid motion is driven. The force term $\mathbf{F}$ is defined as sum of an advective term $\mathbf{F}_\mathrm{A}$ and a baroclinic term $\mathbf{F}_\mathrm{B}$. Note that $\mathbf{F}_\mathrm{B}$ also contains the perturbation pressure $\tilde{p}$ which is taken from the last computed $\mathcal{H}$ during time iteration.

The main advantage of the mentioned simplification process is that the system of equations arising from equation (2) has constant coefficients (i.e. is separable) which allows the use of fast, optimized solvers for uniform grids as the already mentioned FFT solver above. For a detailed derivation of the pressure equation see the FDS Technical Reference Guide [3]. From a mathematical point of view this so-called *Poisson equation* is an elliptic partial differential equation of second order for which different types of boundary conditions i.e. for open and solid boundaries must be specified.

The numerical time-stepping scheme in FDS, an *explicit second-order predictor/corrector scheme*, requires the solution of equation (2) at least twice per time iteration. Because of its strong interaction with the calculation of all other thermodynamic quantities, the solution of the pressure equation is an essential step in the whole time marching scheme and must be treated very efficiently.

Due to the mathematical theory, a purely local approach isn't sufficient to reproduce the global data dependencies for elliptic problems. Increasing the number of subdomains will worsen the convergence rate, possibly up to divergence at a (problem-dependent) critical number of subdomains.

To find a remedy, a domain-spanning correction process should be used which is able to couple the local solutions and spread global information all over the domain.

2.2. Discretization of the Poisson equation

Based on a subdivision of the underlying geometric domain Ω into cubic subdomains Ω_m, $m = 1,\dots,M$, with rectilinear grid decompositions each, the spatial derivative of $\mathcal{H}$ in equation (2) is now discretized by a second-order accurate finite difference method. Note that scalar quantities in FDS such as the pressure are assigned in the center of the single grid cells.

Now, the discretized form of the Poisson equation reads as

$$\frac{\mathcal{H}_{i+1,jk} - 2\mathcal{H}_{ijk} + \mathcal{H}_{i-1,jk}}{\delta x^2} + \frac{\mathcal{H}_{i,j+1,k} - 2\mathcal{H}_{ijk} + \mathcal{H}_{i,j-1,k}}{\delta y^2} + \frac{\mathcal{H}_{ij,k+1} - 2\mathcal{H}_{ijk} + \mathcal{H}_{ij,k-1}}{\delta z^2}$$
$$= -\frac{F_{x,ijk} - F_{x,i-1,jk}}{\delta x} - \frac{F_{y,ijk} - F_{y,i,j-1,k}}{\delta y} - \frac{F_{z,ijk} - F_{z,ij,k-1}}{\delta z} - \frac{\delta}{\delta t}(\nabla\cdot\mathbf{u})_{ijk} \tag{3}$$

where different discretizations for the time derivative of the divergence are used in the predictor and corrector step of the time marching scheme, see the FDS Technical Reference Guide [3].

In the single-mesh case which is based on a subdivision of Ω into n grid cells, this discretization process leads to the following system of equations

$$Ax = b\,, \tag{4}$$

where x, b are vectors in $\mathbb{R}^n$ and A is the well-known *7-point matrix* in $\mathbb{R}^{n\times n}$ with its *5-point* counterpart in 2D as indicated in figure (1). Note that the vector x corresponds to the single components of $\mathcal{H}_{ijk}$ and b to the right hand side entries of equation (3) for short.

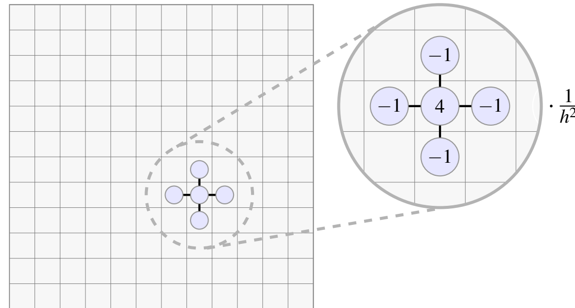


Figure 1: Five-point matrix stencil for equidistant grid size h in 2D

In the multi-mesh case, there are M local systems of equations

$$A_m x_m = b_m \qquad m = 1, \ldots, M, \tag{5}$$

where each subdomain Ω_m has n_m local grid cells. Here, $A_m \in \mathbb{R}^{n_m \times n_m}$ is the local system matrix on subgrid Ω_m with corresponding local solution vector x_m and right hand side vector b_m in $\mathbb{R}^{n_m}$ each. Informally spoken, A_m is the restriction of the global matrix A to the subdomain Ω_m, i.e. $A_m \sim$ "$A\big|_{\Omega_m}$".

2.3. Direct versus Iterative Poisson solvers

Typical approaches for the solution of the Poisson equation belong either to the class of direct or iterative solvers whose basic properties will be shortly summarized below.

2.3.1. Direct solvers

Direct solvers compute the solution of the system of equations (4) in only one single computational cycle which may be very complex. Two prominent representatives, which will be of importance subsequently, are:

- ***LU*-factorization**

  Within the framework of the *Gaussian elimination algorithm*, many direct algorithms are based on the computation of the *LU-factorization* for a permutation of the system matrix, $PAQ = LU$, with suitable permutation matrices P and Q and the lower and upper triangular matrices L and U.

  The solution of (4) can then be obtained using a forward substitution step, $Ly = P^T b$, followed by a backward substitution step, $U(Q^T x) = y$. If A is symmetric a *Cholesky factorization*, $PAP^T = LL^T$, can be applied.

  The whole process can be subdivided into three phases: (i) a *reordering phase* where the matrix is analyzed to produce an ordering of the matrix which allows a more efficient factorization, (ii) a *factorization phase* where the LU-factorization is actually computed and stored, (iii) a *solution phase* where the forward and backward substitution is performed. Typically, the factorization phase (ii) requires the most computing time while the solution phase (iii) is an order of magnitude faster.

  Usually, these solvers are performed with enormous speed and used for the demonstration of potential computer power, see the LINPACK-tests by Dongarra et al. [4].

  They have proven to be very robust even in the non-symmetric and ill-conditioned case. Besides, they are nearly independent of the degree of grid distortion (except of rounding errors) and well suited for the application on unstructured grids.

A main disadvantage is that less benefit can be drawn from a very convenient property of the discretization matrix A, namely its intrinsic *sparsity*: Even though A has only very few non-zero entries compared to the total number of possible entries n^2, the LU-factorization process leads to *fill-in*, i.e. produces non-zero entries in L and U where A was zero before. The computing and storing of both triangular matrices can become prohibitively expensive, especially in case of huge systems of equations with hundreds of millions of unknowns. In view of the computational efficiency of LU-methods for a given application the resulting fill-in is a decisive criterion.

- **Fast Fourier Transformation**

  Spectral solvers like the Fast Fourier Transformation exploit a very special property of the underlying Poisson problem, namely that sine and cosine functions are eigenvectors of the Laplace operator. They expand the solution as a Fourier series which can be quickly performed at rather low complexity. In practice, this approach has proven to be highly efficient and is used in many different fields of application. But in contrast to the LU-approach, they are restricted to structured grids which may impede the use for complex geometries.

Both, the LU- and the FFT-solvers, are based on highly recursive algorithms which strongly couple the overall data of the whole domain. For single-mesh applications this design principle very good reflects the mentioned global character of the pressure. However, for multi-mesh applications the subdivision into single meshes causes a fragmentation of its physical connectivity which inevitably leads to dependencies on the number of meshes and possibly large deteriorations of the resulting efficiency and/or accuracy.

To diminish this effect, the basic methods can be combined with additional strategies. For instance, in FDS the local FFT-solutions are embedded into an iterative update for the velocity components coupling the local solutions by averaging coincident values of the normal velocity at adjacent mesh interfaces. Nevertheless, for problems of moderate size, especially if only less meshes are used, direct solvers may be incomparably fast and should generally be preferred to iterative ones.

2.3.2. Iterative solvers

In contrast to direct solvers, iterative solvers perform multiple computational cycles producing a sequence of iterates which gradually improve an initial estimate of the solution until a specified tolerance has been reached.

Usually, they are easier to implement than direct ones, because they can be reduced to a series of core components such as matrix-vector multiplications, linear-combinations of vectors, scalar-products, etc. for which highly optimized program packages may be used, e.g. BLAS [5].

The computational complexity associated with each single cycle is comprehensively less compared to direct methods. Thus, the decisive question is how many cycles are needed for convergence.

Because iterative methods don't produce any fill-in, they preserve the sparsity structure of the system matrix and are much less demanding with respect to storage than direct methods.

However, iterative methods may depend on special properties of the underlying problem such as symmetry or positive-definiteness. Convergence can be very slow for ill-conditioned problems such that many iterations must be performed to reach the specified tolerance. Furthermore, they often require the optimal choice of different methodical parameters which are highly problem-dependent and difficult to predict.

The rate of convergence usually depends on the grid resolution, but can be considerably improved by a suitable *preconditioning matrix* $B \in \mathbb{R}^{n\times n}$ which transforms the original system $Ax = b$ into an equivalent system $B^{-1}Ax = B^{-1}b$ which may be solved much faster. The more special properties of the given application can be incorporated into B, the better the convergence is, but the higher the computational costs are, too.

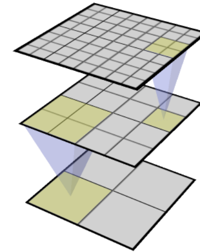
Based on B, the core component of an iterative method is a simple defect correction scheme, the so-called *basic iteration*, $x^k = x^{k-1} - \omega B^{-1}(Ax^{k-1} - b)$, used to minimize the *defect* $d^{k-1} := Ax^{k-1} - b$, which indicates how good $Ax = b$ is fulfilled by the current iterate x^k (measured in a suitable norm). Here, x^k, x^{k-1} are vectors in $\mathbb{R}^n$ with an initial guess x^0, and ω is a *relaxation parameter* which must be chosen very carefully. Simple candidates for B are $B_{diag} \sim$ "diagonal of A'', which corresponds to the *Jacobi scheme*, or $B_{ssor} \sim$ "lower triangular part of A'', which corresponds to the *Gauss-Seidel scheme*.

On its own, the basic iteration is inefficient and takes many iterations to produce a reasonable solution. But the single representatives can be embedded into much stronger surrounding schemes:

- **Conjugate gradient methods**

  CG-methods belong to the class of *Krylov subspace* methods and rest upon the minimization of the equivalent problem $F(x) = 1/2x^TAx - x^Tb \to 0$ by using mutually conjugate directions such that the search directions towards the minimum are continually improved and the minimum is obtained in at most n steps. CG-methods are restricted to symmetric positive-definite problems and only need less storage space for several auxiliary vectors. They are largely based on matrix-vector multiplications which only require computationally cheap data-exchanges between neighboring meshes. Besides, they also make use of global scalar products using data-exchanges between all meshes which are computationally expensive but also contribute to a stronger global coupling.

- **Geometric multigrid methods**
  GMG-methods use a complete hierarchy of grids with different resolutions and are able to achieve fast convergence rates independent of the grid size with moderate computational complexity. The basic idea is to improve the convergence speed of the basic iteration by correcting the defects on successively coarser grids whose data are interpolated between each other. This process explicitly takes advantage of the so-called *smoothing property* of the single representatives of the basic iteration, namely that they smooth out the high frequent error components very fast.



Both, CG- and GMG-methods, are able to reasonably improve the convergence speed of the basic iteration returning the exact solution in a moderate or even small number of iterations depending on the underlying problem. For a detailed overview of iterative methods see Saad [6].

With regard to an efficient parallelization, iterative methods seem to be much easier and more universally applicable than direct ones. In the context of domain decomposition a natural choice for preconditioning is the blockwise solution of the local mesh problems known as *Schwarz preconditioning*, see Xu [7]. But again, this approach necessarily leads to dependencies of the number of meshes which can impair the speedup especially for massively parallel applications.

2.4. Structured versus unstructured discretization

The appropriate treatment of the boundary conditions is of major importance for the accuracy of the whole numerical solution of the pressure equation. There are two different classes of boundaries conditions which must be specified at mesh faces matching with the borders of the computational domain:

- **Dirichlet boundary conditions**
  This type of boundary condition is applied to open boundaries, where the fluid motion into or out of the domain is driven by the pressure gradient. In this case a value for $\mathcal{H}$ itself must be specified at corresponding grid cells.

- **Neumann boundary conditions**
  This type of boundary condition is applied to internal solid obstructions as well as external faces which are entirely determined by a forced flow or a solid. In this case a value for the normal gradient $\partial\mathcal{H}/\partial n$ must be specified at corresponding grid cells.

If there is a mix of open and solid surfaces along an external face, the specification of the Dirichlet condition is given precedence to the Neumann condition (i.e. it is more important to specify $\mathcal{H}$ itself than its gradient).

In case of a multi-mesh application the blockwise FFT method additionally requires the specification of Dirichlet boundary conditions for cells at the interface between adjacent meshes. In contrast to that, the LU-, CG- and GMG-methods solve the corresponding global problem without the need of extra settings at mesh interfaces.

As a member of the Neumann conditions the *no-flux* or *forced-flow* condition

$$\frac{\partial \mathcal{H}}{\partial n} = -F_n - \frac{\partial u_n}{\partial t} \tag{6}$$

is of special interest. F_n denotes the normal component of $\mathbf{F}$ at vents or solid walls and $\partial u_n/\partial t$ the rate of change in the normal component of velocity at a forced vent. In case of an internal obstruction based on a stationary non reaction solid material, a *homogeneous* Neumann condition is present, i.e. the right hand side of (6) is zero.

The external boundary conditions for the forced in- and outflow are prescribed by corresponding forced-flow Neumann conditions. For both, the structured and unstructured discretization, these external boundary settings are similarly included to the discretization matrix and therefore not considered any further.

If the mesh blocks are discretized including all solid obstructions within the computational domain, cells internal to the obstructions are masked as blocked cells. But without any exception, all of these cells are incorporated into the resulting discretization matrix by applying the same matrix stencil all over the domain. Thus, the matrix takes a highly regular shape such that optimized solvers can be applied.

While the no-flux boundary condition is exact at external boundaries, it is not possible to directly prescribe the homogeneous no-flux condition at internal boundaries. To reduce flow penetration into the obstructions, currently a direct forcing method is applied in FDS. In each time step, this method requires the repeated application of the blockwise FFT method until the normal components of velocity are driven to within a specified tolerance. This approach guarantees that the total flux into a given obstruction is always identically zero, even if the normal components of velocity may contain small errors. For more details see the FDS Technical Reference Guide [3].

The only way to get rid of these penetration errors is to discretize on cells that belong to the gas phase only. Now, all cells internal to solid obstructions are omitted from the discretization matrix. On gas cells which are directly adjacent to the surface of an obstruction the proper boundary conditions are explicitly specified and included to the matrix.

In contrast to structured case this strategy requires the usage of individual matrix stencils for the different grid cells depending on their location with respect to obstructions. While the resulting discretization matrix has less entries than before, it hasn't a regular shape anymore and the choice of efficient solvers gets much more difficult.

The different settings for the matrix entries in case of the structured and unstructured approach are illustrated below. Figure (2) outlines a simple 2D geometry with an angled obstruction and the resulting FDS velocity field.

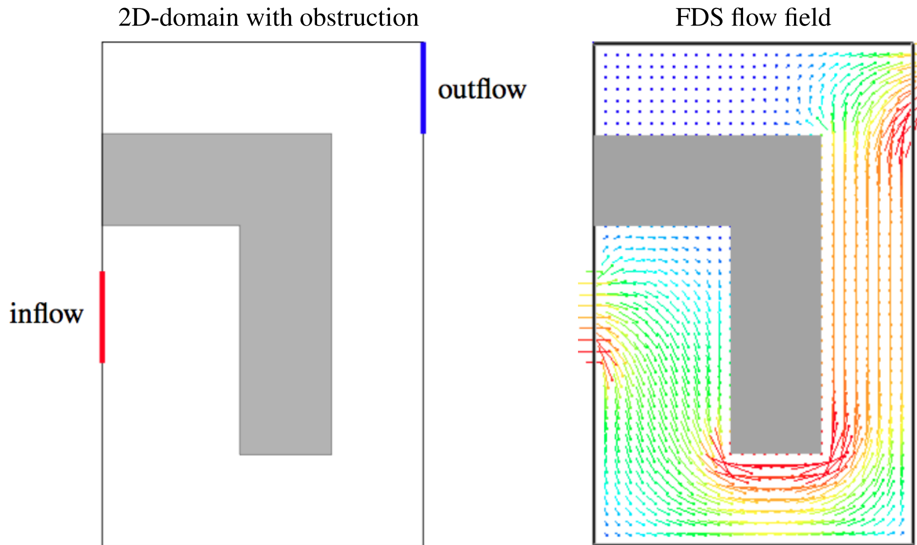


Figure 2: Simple 2D test geometry

The differences get obvious for the treatment of the internal obstruction in figure (3):

- In the structured case the single grid cells are numbered in a lexicographic order including all cells within the obstruction and starting from the bottom left cell up to the top right cell. Here, the cell indices are only determined by the number of grid cells in the single coordinate directions. Based on this regular numbering the connectivity graph of the matrix is clearly defined and the same matrix stencil is applied for all grid cells which allows the usage of the optimized FFT-solver.
- In the unstructured case, there is no seamless numbering any more because the obstruction is excluded from the numbering. Instead, an additional connectivity list must be stored which specifies the neighborhood relations between the single grid cells. For every single cell an individual matrix stencil must be applied which prevents the usage of the optimized FFT-solver. But the unstructured discretization offers more flexibility and accuracy because the correct boundary information along the internal obstruction is now included into the system.

Both discretization strategies will be analyzed in the course of the following numerical tests, namely the structured variant in combination with the FFT-method and the unstructured variant in combination with the LU-, CG- and GMG-methods.

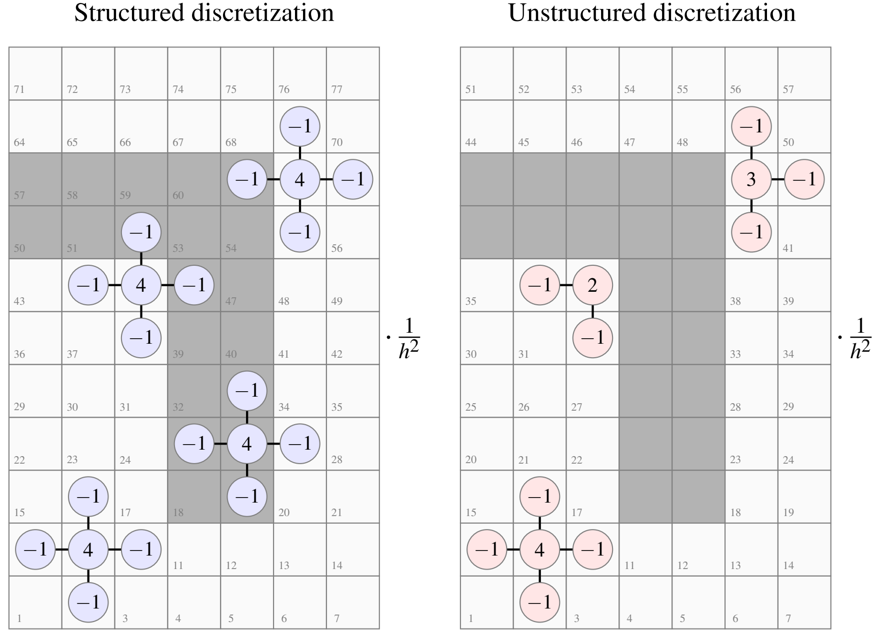


Figure 3: Matrix generation and grid numbering for different grid types

3. NUMERICAL TESTS

The presented classes of Poisson solvers will now be compared within a series of numerical tests. The objective behind is to analyze how the different direct and iterative solvers compete against the currently used direct FFT-solver with respect to their accuracy and scalability properties as well as their computational costs.

3.1. Basic Poisson Solvers

For the solution of the pressure equation in the predictor and corrector step of each single FDS time step the following Poisson solvers will be considered:

- **FFT**(**tol**): Block-FFT with iterative velocity correction up to tolerance **tol**

  (i) First, each mesh performs its own FFT-method to compute a local solution $x_m = A_m^{-1} b_m$, $m = 1, \ldots, M$. The global solution is then built as composition of the local solutions including an averaging process at mesh interfaces, $x = \tilde{\sum}_{m=1}^{M} x_m$.

  (ii) Based on the resulting x it must be checked, if both, the normal velocity components along internal obstructions and the differences of the normal velocity components between neighboring meshes are below the specified tolerance **tol**. If this is not the case, steps (i) and (ii) must be repeated.

Depending on the special test case different values for the tolerance **tol** will be considered. As a mixture of local direct FFT-methods and an iterative correction process this variant is a hybrid Poisson solver.

- **MKL**: Cluster interface of Intel MKL Pardiso solver

  (i) During the FDS-initialization phase a global Cholesky factorization is generated for the global discretization matrix $A = \sum_{m=1}^{M} A_m$. This step includes the preceding computation of a suitable permutation of A which allows a more efficient factorization than the original matrix itself.

  (ii) In every FDS time iteration the solution of the Poisson equation is simply based on a forward and backward substitution step on every single mesh with respect to the stored factorization as described in Section 2.3.1.

  This variant is a purely direct Poisson solver. Note, that the factorization must be recomputed if the geometric situations change during the simulation (e.g. in case of disappearing obstructions). Otherwise the expensive factorization phase is only needed once.

- **ScaRC**: Different representatives of Scalable Recursive Clustering

  Either a global CG-method with blockwise preconditioning, **ScaRC-CG**, or a global GMG-method with blockwise smoothing is applied, **ScaRC-GMG**. Both, the preconditioning and smoothing are based on the application of

  (i) local SSOR-methods with optimal relaxation parameter on the single meshes (with 1 grid cell of overlap), or

  (ii) forward and backward substitutions based on local $L_m U_m$-factorizations (generated by local calls of Intel MKL Pardiso during initialization).

  Depending on the special choices the resulting solvers are of purely iterative or of hybrid type. If computational times for different **ScaRC** cases will be listed below, they are always related to the fasted measured variant for the current test case each.

  In case of **ScaRC-GMG** different coarse grid solvers were tested, namely an iterative global CG-method (up to rounding error) or a direct global **MKL**-solver, on the coarsest grid level each.

Table (1) summarizes the basic properties of the upper solution strategies.

Method	Grid	Type
FFT(**tol**)	structured	hybrid
MKL	unstructured	direct
ScaRC-CG	unstructured	iterative/hybrid
ScaRC-GMG	unstructured	iterative/hybrid

Table 1: Properties of different Poisson solvers

The hybrid combination of local **SSOR**-smoothers with global **MKL** as coarse grid solver within **ScaRC-GMG** only uses two grid levels, the original fine grid level on which the iterative blockwise **SSOR** is used and the next coarser grid level (with double grid size) on which the direct global **MKL** solver is applied. Although the factorization and storing of the global LU-decomposition for level 2 causes additional overhead again, it is only based on an eighth of the grid cells compared the global **MKL** solver on the finest grid level. This hybrid version is associated with the idea that an exact solution on level 2 within the multigrid hierarchy still may contribute to good global coupling but at comprehensively lower costs. Note that this strategy can also be applied for 3 and more grid levels L where the blockwise smoothing is applied on levels 1 up to $L-1$ and the global **MKL** solver on the coarsest level L. These combinations will be subject to future tests.

3.2. Basic Test Geometries

As illustrated in figure (4) the subsequent test computations are based on two very simple test geometries:

- **Cube**$^{-}$: a square-shaped domain without obstruction,
- **Cube**$^{+}$: a square-shaped domain with obstruction.

Both cases use a forced constant inflow of $1m/s$ from the left (with a slight ramping up at the beginning) and an open outflow on the right. Depending on the test case, the cubes are subdivided into different numbers of cells, namely $24^3, 48^3, 96^3, 192^3, 240^3$ and 288^3.

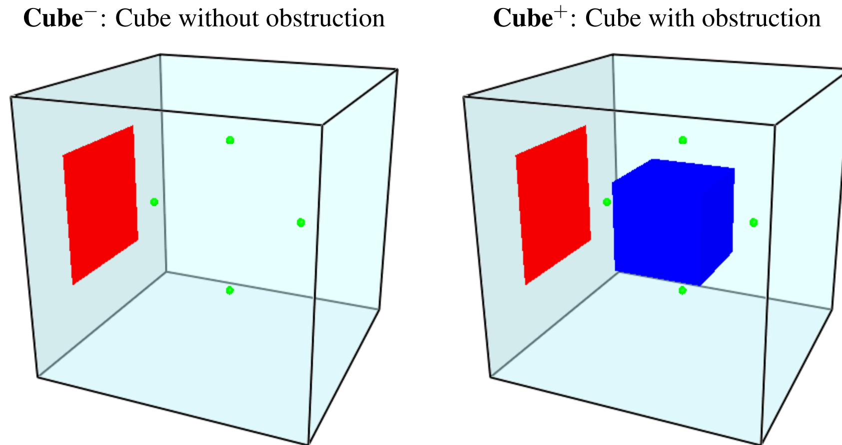


Figure 4: Simple square-shaped test geometries **Cube**$^{-}$ and **Cube**$^{+}$

For all shown test cases, the differences of the volume flow between in- and outflow is measured over the whole simulation time as well as progress of several quantities (e.g. velocity and pressure) in a couple of device positions, indicated as green dots in figure (4). These measurements are used to assess the resulting accuracy for all solver variants.

3.3. Basic Domain Decompositions

In order to analyze the basic scalability and accuracy properties of the different solvers, different subdivisions for $\mathbf{Cube}^-$ and $\mathbf{Cube}^+$ into 1, 8 and 64 meshes are considered, see figure (5).

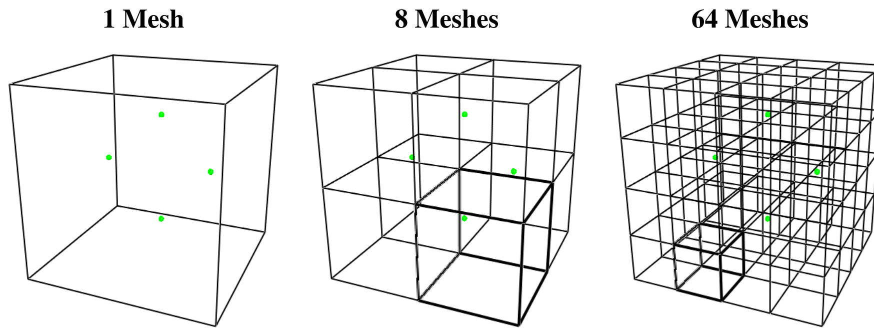


Figure 5: Subdivisions into 1, 8 and 64 subdomains

The number of subdomains for the different geometries is indicated within parenthesis, e.g. $\mathbf{Cube}^+(\mathbf{8})$ denotes the subdivision of $\mathbf{Cube}^+$ into 8 meshes.

3.4. Test 1: $\mathbf{FFT}(\mathbf{tol})$ for different tolerances

The first test is related to the $\mathbf{FFT}(\mathbf{tol})$-method for the $\mathbf{Cube}^+$-geometry. It analyzes how fast the penetration error along the internal obstruction decreases if the tolerance is driven towards zero using **tol**=$10^{-2}, 10^{-6}$ to 10^{-16}, for a grid resolution of 24^3. The same tests have also been performed for finer grid resolutions.

Figure (6) displays the velocity error along the internal obstruction, plotted in a display range with a minimum of 10^{-16} each, with explicit specification of the average number of pressure iterations needed to reach the required tolerances. Obviously, the correction iteration successfully reduces the velocity error along the internal obstruction with decreasing tolerance up to rounding error size.

But whereas only 1 pressure iteration is needed for the coarse tolerance of **tol**=10^{-2}, the number of pressure iterations increases comprehensively from about 3 to 4 for **tol**=10^{-6} up to 30 for **tol**=10^{-16}. Note again that this effort is necessary twice a time step and that **MKL** and **ScaRC** (as unstructured-grid based and globally working methods) automatically reach a tolerance of 10^{-16}.

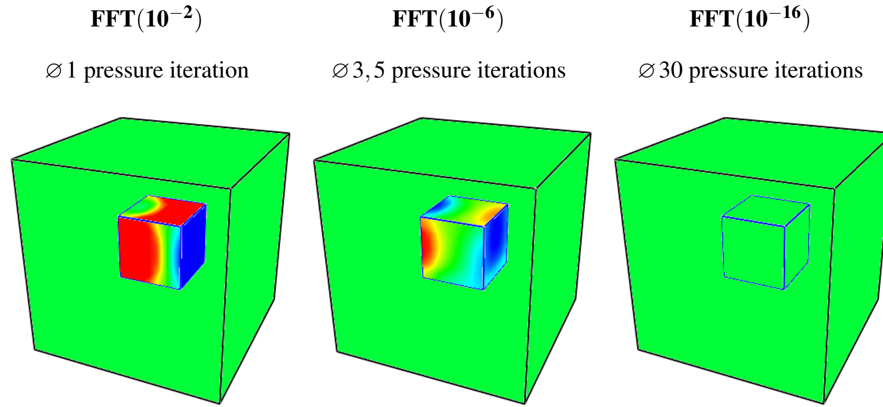


Figure 6: Velocity error and number of pressure iterations for **FFT**(**tol**) on **Cube**$^{+}$(**1**) with 24^3 cells

The next interesting question is how the number of iterations is influenced if the grid additionally is decomposed into single meshes. To this end Table (2) compares the required number of pressure iterations for different **Cube**$^{+}$(**M**)-cases, $M = 1, 8$ and 64, at a grid resolution of 48^3 and a tolerance of **tol**=10^{-6}.

In order to distinguish the costs only related to the reduction of the velocity error along the internal obstruction from that for the reduction of the mesh interface error, the corresponding results 'without obstruction' for **Cube**$^{-}$(**M**) are opposed.

Geometry	Number of meshes M		
	1	8	64
Cube^{-}(M)	1	106	222
Cube^{+}(M)	8	123	254

Table 2: Number of pressure iterations for **FFT**($\mathbf{10^{-6}}$) in case of 48^3 cells and different mesh decompositions

Apparently, the mesh decomposition causes a much higher rise of pressure iterations than the internal obstruction does. Increasing the number of meshes from 1 to 8 for **Cube**$^{-}$, the number of pressure iterations rises up to 106, while the additional internal obstruction in **Cube**$^{+}$(**8**) leads to another rise of only 17 iterations compared to **Cube**$^{-}$(**8**).

A similar relation can be observed for **Cube**$^{-}$(**64**) and **Cube**$^{+}$(**64**): Using 64 meshes instead of 1 mesh leads to an increase of 222 iterations for **Cube**$^{-}$ due to the additional velocity correction along the mesh interfaces. In contrast to that, the addition of the obstruction leads to only 32 more iterations, i.e. the difference between 222 and 254.

3.5. Test 2: Average Poisson solution times for constant problem size

The following subsection is related to a comparison of all three Poisson solvers, **FFT**(**tol**), **MKL** and **ScaRC**, with respect to the time typically needed for the solution of one Poisson equation. All presented simulations have been performed up to a final time of 0.5 seconds where the steady state has long been reached due to the constant inflow. The times indicated subsequently are the average times for all measured Poisson times during the whole simulation.

Note, that a fair time comparison is only possible if the accuracy of the obtained solutions is at the same level for all considered variants. Thus, the identification of a suitable tolerance for **FFT**(**tol**) which guarantees a comparable accuracy with **MKL** and **ScaRC** is of great importance.

For the **Cube**$^-$(**M**) and **Cube**$^+$(**M**) cases it turned out that there is no need to perform the velocity correction in **FFT**(**tol**) up to rounding error size 10^{-16} to be comparable with the other solvers. In case of the constant inflow, a tolerance of 10^{-6} is typically enough to provide a sufficient accuracy related to the mentioned device values and volume flow differences.

However, alternative tests have shown that considerably smaller tolerances may be needed to guarantee a sufficient accuracy in more complex situations e.g. for strongly transient inflow conditions or big numbers of meshes.

For the situation with constant inflow, figure (7) displays the average times for **Cube**$^-$(**8**) and **Cube**$^+$(**8**) at a grid resolution of 48^3 cells each.

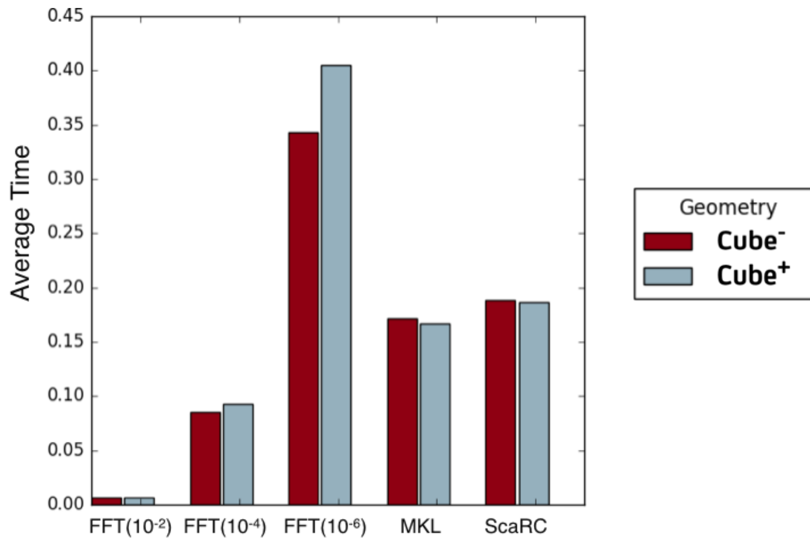


Figure 7: Average Poisson solution times, **Cube**$^-$(**8**) versus **Cube**$^+$(**8**)

In order to point out the resulting large differences in computing time, three variants of **FFT**(**tol**) are listed, namely for the toleranced $10^{-2}, 10^{-4}$ and 10^{-6}, and faced with the **MKL** and **ScaRC** times. It is very evident that **FFT**($\mathbf{10^{-2}}$) is by far the fastest method. In order to drive the velocity error below a tolerance of 10^{-2} only 1 pressure iteration has to be done in average.

Thus, the left both little bars in figure (7) reveal the computational time needed for the pure FFT-method (without additional velocity iteration), however accompanied by the poor accuracy of 10^{-2} along the obstruction and the mesh interfaces.

With regard to the accuracy considerations above, a tolerance of at least 10^{-6} should be used for a proper comparison with the other solvers. For $\mathbf{FFT(10^{-6})}$ however, the situation changes in favor of **MKL** having to the best relation of computational time and accuracy, closely followed by **ScaRC**.

3.6. Test 3: Average Poisson solution times for growing problem size

Up to now both cube domains are associated with a fixed grid resolution of 48^3 and then subdivided into different numbers of meshes. To the same extend as the number of meshes is increased for a given geometry, the communicational overhead for the coordination of these meshes increases too, whereas the computational load per mesh decreases correspondingly. Thus, each additional mesh causes a slight impairment of the relation 'computation versus communication'.

However, it doesn't seem appropriate to use a multi-processor system for the solution of a problem which already can be solved on a single-processor system. Parallel computers should be used to solve huge problems which exceed single-processor capacities by summing up the computational powers of many processors best possibly. Therefore, the next interesting question is, what happens if the problem size is increased by the same relation as the number of meshes is increased, keeping the load per processor at a constant level.

To this end, figure (8) compares the average times for $\mathbf{Cube^+(8)}$ for a grid resolution of 48^3 cells with that of $\mathbf{Cube^+(64)}$ for a grid resolution of 96^3 cells. Again, the three different tolerances $10^{-2}, 10^{-4}$ and 10^{-6} are distinguished for $\mathbf{FFT(tol)}$. Note that the values on the left correspond to the light blue $\mathbf{Cube^+(8)}$-values in figure (7).

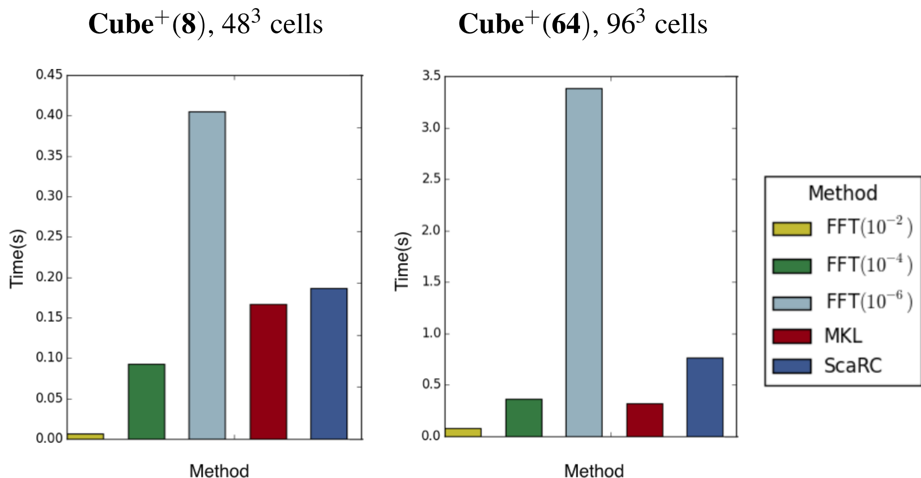


Figure 8: Average times for 1 pressure solution, $\mathbf{Cube^+(8)}$ versus $\mathbf{Cube^+(64)}$

Cube^{+}(**64**) is now based on 8 times as much grid cells as well as 8 times as much meshes compared to **Cube**$^{+}$(**8**) such that each single mesh has the same load of 24^3 cells in both cases. Thus, one would expect nearly the same computational times for both, at most slightly increased times for the 64-mesh case due to the additional overhead for synchronization and communication.

However, figure (8) reveals that the measured times for **Cube**$^{+}$(**64**) are basically much higher than that for **Cube**$^{+}$(**8**). For example, the times for **MKL** amount to about 0.17 seconds for 8 meshes, but to 0.34 seconds for 64 meshes which is just twice as much in spite of the same load per mesh. This trend is still more pronounced for **FFT**($\mathbf{10^{-4}}$) and **ScaRC** which both need about 4 times as long in the 64-mesh case compared to their 8-mesh times. Even worse, the comparably accurate **FFT**($\mathbf{10^{-6}}$) needs more than 8 times as long for 64 meshes.

A next observation is that the relation between **FFT**(**tol**) and **MKL** changes with growing number of meshes to the disadvantage of **FFT**(**tol**). Whereas **MKL** still needs the double time of **FFT**($\mathbf{10^{-4}}$) in the 8-mesh case, it is already a bit faster in the 64 mesh case. The relation between **MKL** and **FFT**($\mathbf{10^{-6}}$) is even worse: While **FFT**($\mathbf{10^{-6}}$) needs about 2.4 times as long as **MKL** for 8 meshes, it needs about the tenfold time for 64 meshes.

All these relations clarify that the increased communication overhead related to the bigger number of meshes massively dominates the whole simulation, even for the optimized **MKL** solver. The losses of global coupling, which are algorithmically unavoidable in **FFT**(**tol**) and **ScaRC**, contribute to another deterioration. Obviously, there is always a price which must be payed for the parallelization, either with respect to losses of accuracy and/or losses of speedup.

It is therefore all the more important to perform simulations at the highest possible usage of local processor storage and computing power in order to relativize these negative impacts and maximally exploit the local processor potential.

3.7. Test 4: Costs of the MKL-method

So far, the preceding section suggests that the **MKL** method provides a good compromise between accuracy and scalability on the one hand and computational efficiency on the other hand. Clearly, the upper test cases only provide a very simple snapshot. Further tests based on more complex geometries and higher numbers of meshes are necessary to get a more comprehensive overview.

However, there is another important property of **MKL** which can affect the decision whether it is a suitable Poisson solver for a given constellation or not. As already mentioned, the **MKL** method rests upon the computation and storage of a *LU*-decomposition which is substantially bigger than the system matrix *A* itself.

Figure (9) illustrates the growing storage needs of **MKL** for **Cube**$^-$ in case of progressive grid refinements with $24^3, 48^3, 96^3$ and 192^3 cells. The blue bars indicate the number of non-zeros in the global system matrix A and the red bars the number of non-zeros in the corresponding LU-decomposition, respectively. Note, that a logarithmic scale is used for the presentation and that the matrix A has only about 7 times as much entries as the number of grid cells due to the 7-point discretization.

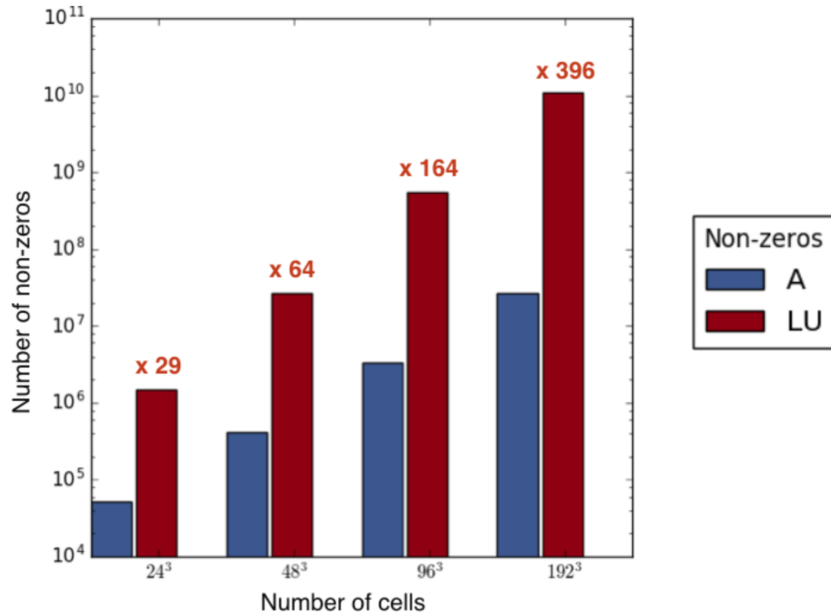


Figure 9: Number of non-zeros in A and LU for **Cube**$^-$ and different grid resolutions

For each grid size, the relation between A and LU is listed above each single LU-bar. If the geometry is decomposed into single meshes, each mesh holds its part of the global A and LU.

It gets very clear, that the additional memory requirements for **MKL** are extremely high. Furthermore, there is no fixed relation between A and LU for all grid resolutions but the LU requires successively more storage as the grid is refined. For a resolution of 96^3 the LU-decomposition is more than 160 times bigger than A, for 192^3 even nearly 400 times.

In contrast to that **FFT**(**tol**) and **ScaRC** only need less additional storage: As **FFT**(**tol**) is based on the regular matrix stencils it doesn't even require the storage of A. In case of **ScaRC** the matrix A and only a handful of additional vectors must be stored.

Furthermore, the initialization phase in **MKL** takes quite a long time. As a small example the **Cube**$^-$(**8**) case with 96^3 grid cells is noted: Here, the initialization phase requires about 5000 seconds, i.e. more than 83 minutes, whereas each single Poisson solution only needs 17 seconds.

In view of the already very long simulation times for a typical fire scenario a longer initialization phase doesn't seem to be that crucial. But it must be taken into account that the LU-factorization probably has to be computed more than once if the geometric situation changes during the simulation, e.g. when obstructions (dis)appear triggered by some actuation mechanisms.

Usually, a given problem must be best possibly adapted to the available hardware resources. In order to achieve a high level of accuracy, finer grid resolutions should be preferred. However, due to its immense additional storage needs **MKL** doesn't allow the same degree of grid resolution on a given platform as **FFT**(**tol**) and **ScaRC**. On the Linux cluster which was used for all test computations (8 multicore processors with 8 cores of type Intel Xeon E7340 each), a grid resolution of 288^3 cells could still be handled by **FFT**(**tol**) and **ScaRC** whereas **MKL** already failed for a resolution of 240^3 cells.

All in all, a careful trade-off between the different advantages and disadvantages of the different solvers must be made to identify the best Poisson solver for a given problem.

3.8. Duct_flow-case: Flow through a pipe

As a final example the duct_flow case of the FDS Verification Guide is presented. It was designed to test the velocity correction of **FFT**(**tol**) for a flow through a duct across different meshes.

The sidewalls of the duct are defined by thin obstructions and the flow is only restricted to the interior of the duct. Figure (10) displays the geometric situation for a mesh decomposition into 8 meshes. Obviously, the flow field is characterized by many direction changes. Due to the variety of internal obstructions and the additional mesh decomposition this case is a big challenge for the **FFT**(**tol**) solver.

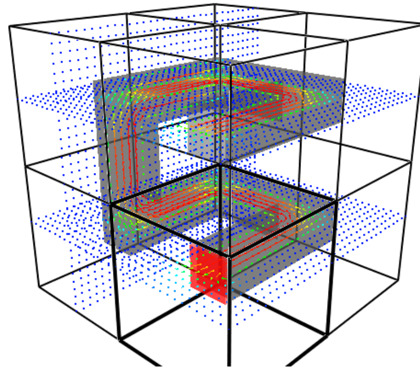


Figure 10: Duct_flow for 8 meshes

For **FFT**($\mathbf{10^{-4}}$), **MKL** and **ScaRC** the following figure (10) compares the average Poisson times and resulting velocity errors along the duct walls for a grid resolution of 64^3 in case of the 8-mesh decomposition. Obviously, the full accuracy 10^{-16} is achieved by **MKL** and **ScaRC** as can be expected by construction. **MKL** provides the shortest average time per Poisson solution again, followed by **ScaRC** which needs 1.7 times longer. But due to the handling of the more complex internal structures, the velocity correction in **FFT**($\mathbf{10^{-4}}$) converges rather slowly such that about 1000 iterations are needed to even reach the relatively coarse tolerance of 10^{-4} along with nearly the tenfold average Poisson time.

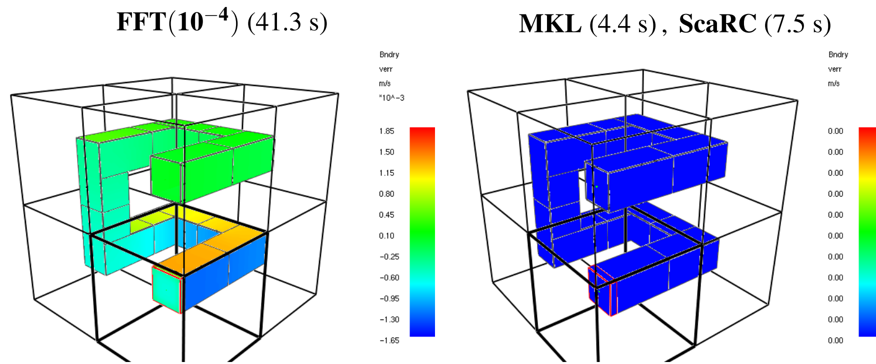


Figure 11: Velocity errors and average Poisson times for the duct-flow case

Corresponding tests were also performed for the 64-mesh decomposition confirming similar relations. Besides several tests were done regarding different settings for the usage of OpenMP: For grid resolutions of 64^3 and 128^3 cells, either 1, 2 or 4 OpenMP-threads were used. But the results were somewhat sobering. At least on the used Linux cluster platform the usage of 4 OpenMP-threads only led to an improvement of 5% in the maximum compared to the use of 1 OpenMP-thread which doesn't seem to be worth the fourfold increase of cores. More related tests are planned on other platforms.

3.9. Summary

The upper tests suggest that the choice of a suitable Poisson solver for a given constellation depends on a variety of different criteria and and must be oriented towards a clever balance between:

- a sufficient accuracy (How big is the resulting error along internal obstructions and mesh interfaces?)
- a good performance (How much computational time is needed for a single Poisson solution?)
- the algorithmical costs (How many additional storage is needed? Is it necessary to install extra libraries?)

If the present Poisson solvers are viewed in this light, the following conclusions can currently be drawn: Although **FFT**(**tol**) is only working for structured discretizations, it has proven to be incomparably fast and efficient for a big variety of single-mesh constellations. For multi-mesh problems it depends: If a sufficient accuracy can be achieved by using only a coarse tolerance (e.g. in steady state situations and/or for single- or 'less-mesh'-decompositions) the same holds true in many cases. But the resulting accuracy along (complex) internal obstructions or mesh interfaces may drop down comprehensively for more transient situations or massively parallel subdivisions and must be kept in view carefully.

MKL and **ScaRC** are based on unstructured discretizations which allow the correct mapping of internal boundary conditions along immersed bodies. Thus, they are basically able to provide a higher accuracy in complex geometric situations. Additionally, they both manage without the setting of artificial boundary conditions along mesh interfaces, but rather act in a more domain-spanning way which better fits the requirements of the global pressure.

For the cases discussed above, the **MKL** method has shown the most efficient and robust behavior with respect to accuracy and computational time if the number of meshes is increased. The reordering and factorization of the LU-decomposition requires a substantial effort but has to be done only once at the best. The forward/backward substitution is much faster, such that the effort per FDS time step is acceptable. However, the LU-decomposition is much more dense than the system matrix itself and must be stored additionally which is extremely memory intensive for small grid resolutions and subdivisions with many meshes.

ScaRC lies somewhere in-between **MKL** and **FFT**(**tol**) providing a relatively high level of accuracy combined with a moderate computational speed. It should be noted that there is still additional potential of improvement, as will be explained in a little more detail in the following outlook section.

For every considered solver it may be difficult to evaluate its behavior for a new constellation in advance. The scalability of all variants is far from being optimal: The speedup decreases considerably with increasing number of processors even if the load per processor is kept constant. Thus, their behavior for massively parallel applications with hundreds or even thousands of meshes cannot be predicted yet. The best scalability behavior can currently be observed for **MKL**, but its competitiveness or growing subdivisions will decrease comprehensively due to the need of storing the global LU-factorization for an ever-growing number of meshes and grid cells.

In contrast to that, **ScaRC** shows better scalability properties than **FFT**(**tol**) and much less memory needs than **MKL** (while attaining the same level of accuracy). Thus, it could be a good compromise on the way to a higher degree of parallelism, but this has to be confirmed by further tests for more complex situations.

3.10. Outlook

All in all, the upper test cases only provide some basic assessments of the relationship between the considered Poisson solvers but are still insufficient to give a consistent overall picture yet. In order to widen the view further test series are planned based on geometries with a higher degree of complexity using bigger numbers of meshes and more complicated internal obstructions.

With respect to **ScaRC** additional optimizations regarding its applicability on unstructured meshes and the related preconditioning and smoothing techniques are to be implemented. Furthermore, the use of optimized libraries for vector- and matrix-operations as well as the use of OpenMP capabilities will be included into the code in the near future.

It is furthermore intended to use **MKL** and **ScaRC** in two additional contexts:

- Currently, the development of *Immersed Boundary Methods* (IBM) which allow the inclusion of complex, non-Cartesian bodies into the FDS grid, is the object of continual further improvement. **MKL** and **ScaRC** are going to be tested as solvers for the solution of the implicit scalar advection-diffusion in a band of cells around the surface of an immersed body, the so-called *cut-cell* region. Compared to the whole Poisson problem this is a much smaller problem such that better performances can be expected.
- In order to preserve the advantages of the **FFT**-method with respect to its low memory needs and its high local performance it is intended to decompose the Poisson solution on the Cartesian unstructured mesh into a Poisson solution based on **FFT** on the structured mesh plus a Laplace solution based on **MKL** or **ScaRC** on the Cartesian unstructured mesh, where the Laplace solution serves as velocity correction process.

References

[1] A. Sweet, Roland. Crayfishpak: A vectorized fortran package to solve helmholtz equations.

[2] P. Orlandi E.A. Fadlun, R. Verzicco and J. Mohd-Yusof. Combined immersed-boundary finite- difference methods for three-dimensional complex flow simulations. *Journal of Computational Physics*, 161, 2000.

[3] Kevin B. McGrattan, Simo Hostikka, Randall McDermott, Jason Floyd, Craig Weinschenk, and Kristopher Overholt. *Fire Dynamics Simulator (Version 6) Technical Reference Guide, Volume 1: Mathematical Model*. National Institute of Standards and Technology, 6.5.2 edition, August 2016.

[4] Jack J. Dongarra, Lain S. Duff, Danny C. Sorensen, and Henk A. Vander Vorst. *Numerical Linear Algebra for High Performance Computers*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[5] Jack Dongarra. Basic linear algebra subprograms technical forum standard, international journal of high performance. *Applications and Supercomputing*, (16(1)):1–111, 2002.

[6] Y. Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, second edition, 2003.

[7] Jinchao Xu. Iterative methods by space decomposition and subspace correction. *SIAM Review. A Publication of the Society for Industrial and Applied Mathematics*, 34(4):581–613, 1992.