university of groningen

University Medical Center Groningen

University of Groningen

Numerical methods for studying transition probabilities in stochastic ocean-climate models

Baars, Sven

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2019

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Baars, S. (2019). *Numerical methods for studying transition probabilities in stochastic ocean-climate models*. Rijksuniversiteit Groningen.

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 02-10-2022

LINEAR SYSTEMS

During the application of Newton's method in both continuation processes as described in Section 2.3 and implicit time stepping as described in Section 2.4.4, linear systems have to be solved. Ultimately, our goal is to solve linear systems with the Jacobian of the model as described in Section 2.5.2, but for now, we will just focus on solving the incompressible Navier–Stokes without the additional tracers. The most elegant way to solve these linear systems is to replace the complete LU factorization by an accurate incomplete one, which can be used as a preconditioner in an iterative method. Moreover, this preconditioner can be used to find approximate eigenvalues and eigenvectors of the Jacobian matrix during the continuation process. By an appropriate ordering technique and dropping procedure, one can construct an incomplete LU (ILU) factorization that yields grid independent convergence behavior and approaches an exact factorization as the amount of allowed fill is increased.

The incompressible Navier–Stokes equations can be written as

$$\begin{aligned} \frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} &= -\nabla p + \frac{1}{\mathrm{Re}} \Delta \boldsymbol{u}, \\ \nabla \cdot \boldsymbol{u} &= 0, \end{aligned} \tag{3.1}$$

where $\mathrm{Re} = \frac{\rho U}{\mu}$ is the Reynolds number, ρ is the density and μ is the dynamic viscosity. These equations are discretized using a second-order symmetry-preserving finite volume method on an Arakawa C-grid; see Figure 3.1. The discretization leads to a system of ordinary differential equations (ODEs)

$$\begin{aligned} M \frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}t} + N(\boldsymbol{u}, \boldsymbol{u}) &= -G\boldsymbol{p} + \frac{1}{\mathrm{Re}} L\boldsymbol{u} + \boldsymbol{f}_{\boldsymbol{u}}, \\ -G^T \boldsymbol{u} &= \boldsymbol{f}_p, \end{aligned}$$

where $\boldsymbol{u}$ and $\boldsymbol{p}$ now represent the velocity and pressure in each grid point, $N(\cdot,\cdot)$ is the bilinear form arising from the convective terms, L is the discretized Laplace operator, G is the discretized gradient operator, M is the mass matrix, which contains the volumes of the grid cells on its diagonal and f contains the known part of the boundary conditions. Note that minus the transpose of the gradient operator gives us the divergence operator.

If we fix one of the variables in the bilinear form N, it becomes linear, hence we may write

$$N(\boldsymbol{u}, \boldsymbol{v}) = N_1(\boldsymbol{u})\boldsymbol{v} = N_2(\boldsymbol{v})\boldsymbol{u},$$

where we assume that $N_1(\boldsymbol{u})$ contains the discretized convection terms, and $N_2(\boldsymbol{u})$ contains the cross terms. Using this notation, the linear system of saddle point type (Benzi et al., 2005) that has to be solved with Newton's method in a continuation is given by

$$\begin{pmatrix} N_1(\boldsymbol{u}) + N_2(\boldsymbol{u}) - \frac{1}{\mathrm{Re}}L & G \\ G^T & 0 \end{pmatrix} \begin{pmatrix} \Delta\boldsymbol{u} \\ \Delta\boldsymbol{p} \end{pmatrix} = - \begin{pmatrix} \boldsymbol{f}_{\boldsymbol{u}} \\ \boldsymbol{f}_p \end{pmatrix}. \tag{3.2}$$

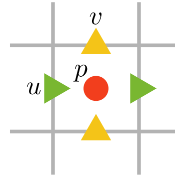


Figure 3.1: Positioning of unknowns in an Arakawa C-grid

It is known (Verstappen and Veldman, 2003), that, on closed domains, dissipation of kinetic energy only occurs by diffusion. Our discretization preserves this property, which has the consequence that for divergence-free $\boldsymbol{u}$, the operator $N_1(\boldsymbol{u})$ is skew-symmetric. This means that if we omit $N_2(\boldsymbol{u})$, the approximate Jacobian will become negative definite on the space of discrete divergence-free velocities. Omitting $N_2(\boldsymbol{u})$ greatly simplifies the solution process since definiteness is a condition under which many standard iterative methods converge. This is a simplification that many authors make, and results in replacing the Newton iteration by a Picard iteration (Elman et al., 2014). This works well for reasonably low Reynolds numbers, and far away from bifurcation points where steady solutions become unstable. The Picard iteration, however, seriously impairs the convergence rate of the nonlinear iteration (Carey and Krishnan, 1987; Baars et al., 2019b).

Similarly some authors use time dependent approaches to study the stability of steady states (Dijkstra et al., 2014). This approach, however, also requires some tricks to obtain the desired information.

Since we want to study precisely the phenomena where the above methods experience difficulty, we would rather use the full Jacobian matrix of the nonlinear equations and apply Newton's method directly.

There are many methods that are based on segregation of physical variables which can solve the linear equations that arise in every Newton iteration. In this approach the system is split into subproblems of an easier form for which standard methods exist. The segregation can already be done at the discretization level, e.g. by doing a time integration and solving a pressure correction equation independently of the momentum equations (Verstappen and Veldman, 2003; Verstappen and Dröge, 2005). Another class of methods performs the segregation during the linear system solve, often in a preconditioning step. Important are physics based preconditioners (De Niet et al., 2007; Klaij and Vuik, 2012; Cyr et al., 2012; He et al., 2018), which try to split the problem in subsystems which capture the bulk of the physics. The subsystems are again solved by iterative procedures, e.g. algebraic multigrid (AMG) for Poisson-like equations. These methods consist of nested loops for: the nonlinear iteration, iterations over the coupled system, and iterations over the subsystems. The stopping criteria of all these different iterations have to be tuned to make the solver efficient. Furthermore, the total number of iterations in the innermost loop is given by the product of the number of iterations performed on all three levels of iteration and thus easily becomes excessive. This is a major problem when trying to achieve extreme parallelism, because each innermost iteration typically requires global communication in inner products. The number of levels of nested iteration may increase even more if additional physical phenomena are added (De Niet et al., 2007; Thies et al., 2009). Our ultimate aim is to get rid of the inner iterations altogether and to solve the nonlinear equations using a subspace accelerated inexact Newton method. In Sleijpen and Wubs (2004) we did this for simple eigenvalue problems using the Jacobi–Davidson method, which is in fact an accelerated Newton method. The method we present here will make this feasible for the 3D Navier–Stokes equations.

For this, fully aggregated approaches are important. In this category, multigrid and multilevel ILU methods for systems of PDEs exist (see Trottenberg et al. (2000); Wathen (2015) and references therein). The former is attractive but for those methods smoothers may fail due to a loss of diagonal dominance for higher Reynolds Numbers, for which a common solution is to use time integration (Feldman and Gelfgat, 2010). The latter comprise AMG algorithms and multilevel methods like MRILU (Botta and Wubs, 1999) and ILUPACK (Bollhöfer and Saad, 2006). ILUPACK has been successful in many fields since it uses a bound to preclude very unstable factorizations. However, this method does not show grid independence for Navier–Stokes problems and is difficult to parallelize (Aliaga et al., 2008). It works well for large problems, but not yet beyond a single shared memory system.

In this chapter we present a novel multilevel preconditioning method

which is specialized for the 3D Navier–Stokes equations. In Section 3.1, we first describe the two-level ILU preconditioner as introduced in Wubs and Thies (2011) and Thies and Wubs (2011). After this, we generalize the two-level method to a multilevel method in Section 3.2. To make this method work for the 3D Navier–Stokes equations, we introduce a skew partitioning method in Section 3.3. In Section 3.4 we then discuss the scalability and general performance of the method, and compare it to existing methods, after which we finalize by providing a summary and discussion in Section 3.5.

3.1 The two-level ILU preconditioner

In Wubs and Thies (2011) a robust parallel two-level method was developed for solving

$$A\boldsymbol{x} = \boldsymbol{b},$$

with $A \in \mathbb{R}^{n\times n}$ for a class of matrices known as $\mathcal{F}$-matrices. An $\mathcal{F}$-matrix is a matrix of the form

$$A = \begin{pmatrix} K & B \\ B^T & 0 \end{pmatrix},$$

with K symmetric positive definite and B such that it has at most two entries per row and the entries in each row sum up to 0, as is the case for our gradient matrix G (Tuma, 2002; De Niet and Wubs, 2008). It was shown that the two-level preconditioner leads to grid-independent convergence for the Stokes equations on an Arakawa C-grid, and that the method is robust even for the Navier–Stokes equations, which strictly speaking do not yield $\mathcal{F}$-matrices because K becomes nonsymmetric and possibly indefinite.

Rather than reviewing the method and theory in detail, we will only briefly present it here. For details, see Wubs and Thies (2011) and Thies and Wubs (2011).

To simplify the discussion, we focus on the special case of the 3D incompressible Navier–Stokes equations in a cube-shaped domain, discretized on an Arakawa C-grid (see Figure 3.1). We refer to the velocity variables, which are located on the cell faces as V-nodes, and to the the pressure, which is located in the cell center, as P-node. The variables are numbered cell-by-cell, i.e. the first three variables are the $u/v/w$-velocity at the north/east/top face of the cell in the bottom south west corner of the domain, and variable four is the P-node in its center. Appropriate boundary conditions (e.g. Dirichlet conditions) are easily implemented in this situation. We remark that the algorithm (and software) can handle more general situations like rectangular domains, interior boundary cells, etc., and could be generalized to arbitrary domain shapes and partitionings.

First we describe the initialization phase of the preconditioner, which is the necessary preprocessing that has to be done only once for a series of linear systems with matrices with the same sparsity pattern. Then we describe the factorization phase, which has to be done separately for every matrix. Finally we describe the solution phase, which has to be performed for every linear system.

Initialization phase 3.1.1

First the system is partitioned into non-overlapping subdomains. These subdomains may be distributed over different processes, which allows for parallelization of the computation. The partitioning may be done based on the graph of the matrix, as implemented for instance in METIS (Karypis and Kumar, 1998), or by a geometric approach, e.g. by dividing the domain into rectangular subdomains. An example of a Cartesian partitioning of a square domain is shown in Figure 3.2.

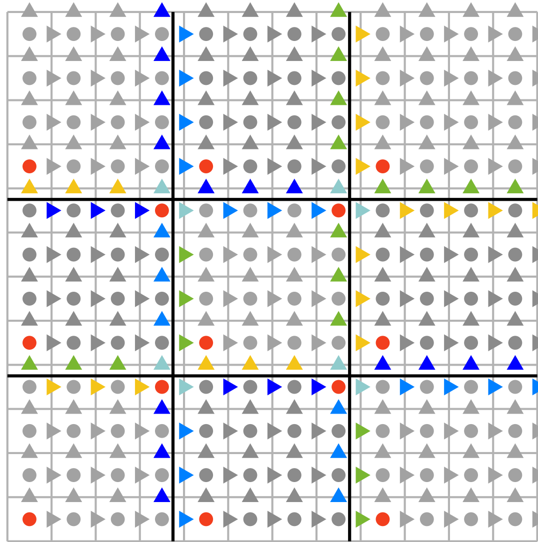


Figure 3.2: Cartesian partitioning of a 12×12 C-grid discretization of the Navier–Stokes equations into 9 subdomains of size $s_x \times s_y$ with $s_x = s_y = 4$. The interiors are shown in gray. Velocities of the same (non-gray) color that are on the same face of neighboring cells form a separator group. The red circles are pressure nodes that are retained.

In the discretization, variables in a subdomain may be coupled to variables in other subdomains. This means that the linear systems associated with the

subdomains may not be solved independently. For this reason we define overlapping subdomains, which have *interior nodes* that are in the non-overlapping subdomain and are coupled only to variables in the overlapping subdomain. The interiors of all subdomains may be solved independently. The nodes in the overlapping subdomains that are not coupled only to variables in the same overlapping subdomain constitute the *separator nodes*. One *separator* is defined as a set of separator nodes that are coupled with the same set of subdomains. One *separator group* is defined as the variables on the same separator that have the same variable type. In Figure 3.2 the interior nodes are shown in gray and the different separator groups are denoted by different colors.

We can now reorder the matrix A such that the interiors (I) and separators (S) are grouped. This gives us the system

$$\begin{pmatrix} A_{II} & A_{IS} \\ A_{SI} & A_{SS} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_I \\ \boldsymbol{x}_S \end{pmatrix} = \begin{pmatrix} \boldsymbol{b}_I \\ \boldsymbol{b}_S \end{pmatrix},$$

where A_{II} consists of independent diagonal blocks. Since A_{II} consists of independent diagonal blocks, applying A_{II}^{-1} is easy and trivial to parallelize. For this reason, we aim to solve

$$\begin{aligned} S\boldsymbol{x}_S &= \boldsymbol{b}_S - A_{SI}A_{II}^{-1}\boldsymbol{b}_I, \\ \boldsymbol{x}_I &= A_{II}^{-1}\boldsymbol{b}_I - A_{II}^{-1}A_{IS}\boldsymbol{x}_S, \end{aligned}$$

where S is the Schur complement given by $S = A_{SS} - A_{SI}A_{II}^{-1}A_{IS}$.

Whether a variable is coupled to a different subdomain can be determined from the graph of the matrix. It may again also be determined geometrically by additionally defining the overlapping subdomains during the partitioning step, and checking what overlapping subdomains a node of the non-overlapping subdomain belongs to. The separators can be determined by, for every node, making a set of subdomains it is coupled to or overlapping subdomains it belongs to, and then grouping the nodes that have the same set.

There are three types of separators: faces (in 3D), edges and corners. For the Navier–Stokes problem on a C-grid, these separators only consist of V-nodes. The P-nodes are only connected to V-nodes that belong to the same overlapping subdomain, so these should never lie on a separator. We arbitrarily choose one P-node in every interior which we also define to be its own separator group to make sure the Schur complement stays nonsingular.

For every separator we now define a Householder transformation which decouples all but one V-node from the P-nodes (Wubs and Thies, 2011). This transformation is of the form

$$H_j = I - 2\frac{\boldsymbol{v}_j\boldsymbol{v}_j^T}{\boldsymbol{v}_j^T\boldsymbol{v}_j}, \tag{3.3}$$

for some separator group g_j with

$$\boldsymbol{v}_j^{(k)} = \begin{cases} \boldsymbol{e}_j^{(k)} + \|\boldsymbol{e}_j\|_2 & \text{if node } k \text{ is the first node of separator group } g_j \\ \boldsymbol{e}_j^{(k)} & \text{otherwise} \end{cases} \tag{3.4}$$

and

$$\boldsymbol{e}_j^{(k)} = \begin{cases} 1 & \text{if node } k \text{ is in separator group } g_j \\ 0 & \text{if node } k \text{ is not in separator group } g_j \end{cases}$$

for all $k = 1, \ldots, n$. We call the one V-node that is still coupled to the P-nodes a V_Σ node.

The physical interpretation of this Householder transformation is that the V_Σ velocities that remain on a separator face provide an approximation of the collective flux through that face. It is therefore important that the variables are correctly scaled before the factorization in a way that they represent fluxes. In the matrix in (3.2) this gives a G-part with entries of constant magnitude, in which case the Householder transformations will exactly decouple the non-V_Σ nodes from the pressures. Since the Householder transformation can be applied independently for every separator, we may collect all these transformations in one single transformation matrix H.

3.1.2 Factorization phase

For every overlapping subdomain Ω_i, $i = 1, \ldots, N$, where N is the total number of overlapping subdomains, we can define a local matrix

$$A^{(i)} = \begin{pmatrix} A_{II}^{(i)} & A_{IS}^{(i)} \\ A_{SI}^{(i)} & A_{SS}^{(i)} \end{pmatrix}.$$

After computing the factorization $A_{II}^{(i)} = L_{II}^{(i)} U_{II}^{(i)}$, the local contribution to the Schur complement is given by

$$S_i = A_{SS}^{(i)} - A_{SI}^{(i)} (L_{II}^{(i)} U_{II}^{(i)})^{-1} A_{IS}^{(i)},$$

and the global Schur complement is given by

$$S = \sum_{i=1}^{N} S_i.$$

We now apply the Householder transformation

$$S_T = H(\sum_{i=1}^{N} S_i) H^T = \sum_{i=1}^{N} H_i S_i H_i^T, \tag{3.5}$$

which can be done separately for every separator or subdomain, or on the entire Schur complement. We choose to do this separately for every subdomain, since H is very sparse, and sparse matrix-matrix products are very expensive and memory inefficient.

We now drop all connections between V_Σ and non-V_Σ nodes and between non-V_Σ nodes and non-V_Σ nodes in different separator groups. The dropping that is applied here is what makes our factorization inexact. Not applying the dropping gives us a factorization that can still be partially parallelized, but is also exact.

Our transformed Schur complement is now reduced to a block-diagonal matrix with blocks for the non-V_Σ nodes for every separator and one block for all V_Σ nodes, which we will call $S_{\Sigma\Sigma}$. Separate factorizations can again be made for all these diagonal blocks, which can again be done in parallel. For the non-V_Σ blocks, sequential LAPACK solves can be used, and for $S_{\Sigma\Sigma}$ we can employ a (distributed) sparse direct solver. We denote the factorization that is computed by these solvers by $L_S U_S$.

The full factorization obtained by the method is given by

$$A = \begin{pmatrix} L_{II} & 0 \\ A_{SI} & H^T L_S \end{pmatrix} \begin{pmatrix} U_{II} & (L_{II} U_{II})^{-1} A_{IS} \\ 0 & U_S H \end{pmatrix}.$$

3.1.3 Solution phase

After the preconditioner has been computed, it can be applied to a vector in each step of a preconditioned Krylov subspace method, for which we use the well-known GMRES method (Saad and Schultz, 1986). Communication has to occur in the application of A_{IS} and A_{SI}, and when solving the distributed V_Σ system. The latter was adressed by using a parallel sparse direct solver in Thies and Wubs (2011), but in the next section we propose a different road that does not rely on the availability of such a solver.

3.2 The multilevel ILU preconditioner

The main issue with the above two-level ILU factorization that prevents scaling to very large problem sizes in three space dimensions is that as the problem size increases and the subdomain size remains constant, the size of $S_{\Sigma\Sigma}$ will increase steadily and any (serial or parallel) sparse direct solver will eventually limit the feasible problem sizes. Increasing the subdomain size, on the other hand, leads to more iterations and therefore more synchronization points.

One of the strong points, on the other hand, is the fact that it preserves the structure of the original problem in the sense that, when applied to an $\mathcal{F}$-matrix, it produces a strongly reduced matrix $S_{\Sigma\Sigma}$ which is again an $\mathcal{F}$-matrix. It is therefore intuitive to try applying the scheme recursively to avoid

the problem of the growing sparse matrix that has to be factorized. From the structure preserving properties of the algorithm, it is immediately clear that such a recursive scheme will again lead to grid-independent convergence if the number of recursions is kept constant as the grid size is increased.

From now on we will refer to the number of recursions, or the number of times a reduced matrix $S_{\Sigma\Sigma}$ is computed, as the number of levels. Note that this means that the method, which was previously referred to the two-level method is in fact the first level of the multilevel method. Applying a direct solver to S_T from (3.5) would be level zero, since in this case the preconditioner itself is in fact just a direct solver.

In order to apply the method to the reduced matrix $S_{\Sigma\Sigma}$, we need a coarser partitioning for which it is most optimal in terms of communication to make sure subdomains are not split up in the new partitioning. In case we have a regular partitioning like a rectangular partitioning this may be done by multiplying the *separator length* by a certain *coarsening factor*. Having a coarsening factor of 2 would mean that in 3D the separator length is increased by a factor 2, and the number of subdomains is reduced by a factor 8.

As stated in the previous section, we require the velocity variables to be correctly scaled to be able to apply the Householder transformation. However, the V_Σ-variables from the previous level that lie on one separator had a different number of variables in their separator groups. In case of a regular partitioning, an edge separator, for instance, consists of V_Σ-nodes from two edges and one corner from the previous level This leads to a different scaling of the V_Σ-nodes and thus to non-constant entries in the G-part of $S_{\Sigma\Sigma}$. Instead of re-scaling the $S_{\Sigma\Sigma}$ matrix on every level, we instead use a test vector $\boldsymbol{t}$. The test vector is defined initially as a constant vector of ones, and is multiplied by the Householder transformation H at each consecutive level. The Householder transformation is as defined in (3.3) and (3.4), but now with

$$\boldsymbol{e}_j^{(k)} = \begin{cases} \boldsymbol{t}^{(k)} & \text{if node } k \text{ is in separator group } g_j \\ 0 & \text{if node } k \text{ is not in separator group } g_j \end{cases}$$

for all $k = 1, \ldots, n$. After applying the Householder transformation, we can again apply dropping to remove connections between V_Σ and non-V_Σ nodes and between non-V_Σ nodes and non-V_Σ nodes in different separator groups. When the matrix $S_{\Sigma\Sigma}$ is sufficiently small, a direct solver is applied to factorize it.

Instead of just having one separator group per variable per separator, we may also choose to have multiple separator groups, meaning that instead of retaining only one V_Σ node per variable per separator we retain multiple V_Σ nodes. This in turn means that less dropping is applied, and therefore the factorization is more exact. Retaining all nodes in this way, possibly only after reaching a certain level, gives us an exact factorization, which, in terms of

iterations for the outer iterative solver, should give the same results as using any other direct solver at that level.

3.3 Skew partitioning in 2D and 3D

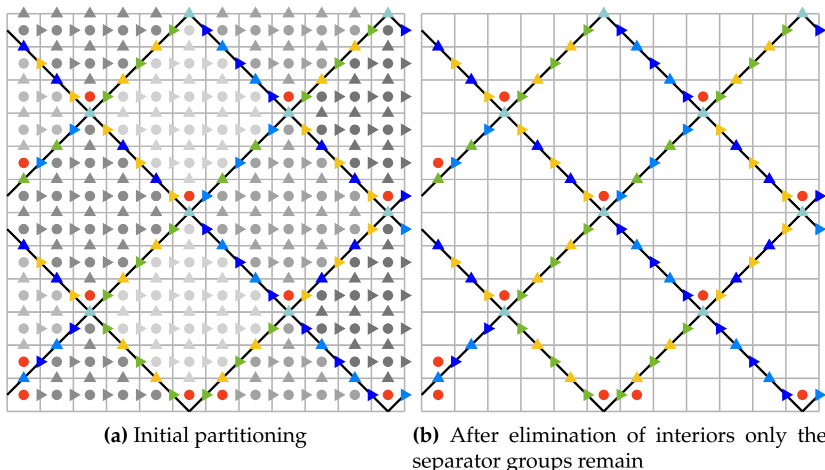
Looking at Figure 3.2, we see that there are pressures that are located at the corners of the subdomains that are surrounded by velocity separators. This means that if we add these pressures to the interior, as was suggested above, we get a singular interior block. We call these pressure nodes that are surrounded by velocity separators *isolated pressure nodes*. For the two-level preconditioner in 2D, it was possible to solve this problem by turning these pressures into their own separator groups. This unfortunately does not work for the multilevel method, since in this case velocity nodes around the isolated pressure nodes get eliminated. It also does not work in 3D because there the isolated pressure nodes also exist inside of the edge separators, where they form tubes of isolated nodes.

A possible way to solve this for the multilevel case and in 3D is to also turn all velocity nodes around the isolated pressure nodes into separate separator groups. This means that they will all be treated as V_Σ nodes and will never be eliminated until they are in the interior of the domain at a later level. This, however, has a downside that a lot more nodes have to be retained at every level, resulting in much larger $S_{\Sigma\Sigma}$ matrices at every level. Another downside is that a lot of bookkeeping is required to properly keep track of all the extra separator groups.

In 2D, we can resolve these problems by rotating the Cartesian partitioning by 45 degrees. The result is shown in Figure 3.3a. It is easy to see that in this case no pressure nodes are surrounded by only velocity separators. We call this partitioning method *skew partitioning*. In Figure 3.3, we also show the workings of the multilevel method, with all the steps being displayed in the different subfigures.

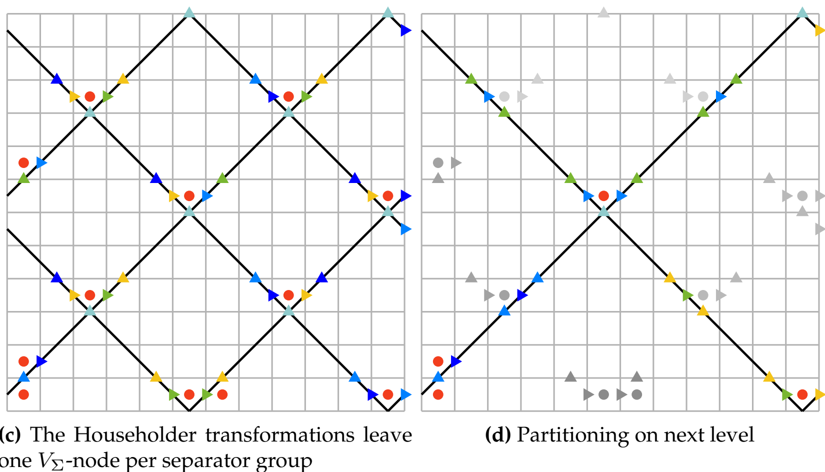
The most basic idea for generalizing the rotated square shape to a 3D setting was to use octahedral subdomains. Partitioning with this design turned out to be unsuccessful, but it is still briefly discussed here since it led to some new insights. Since regular octahedrons (the dual to cubes, having their vertices at the centers of the cube faces) in itself are not space tiling, the octahedrons can be slightly distorted to make them fit within a single cubic repeat unit. The resulting subdomains are space tiling, but only by using three mutually orthogonal subdomain types. The fact that it requires the use of three types of templates increases the programming efforts significantly since it introduces a lot of exceptional cases that should be considered, e.g. for subdomains located at the boundary of the domain.

A problem with the octahedral subdomains is that they are not scalable, meaning that we can not construct a larger octahedral subdomain from mul-



(a) Initial partitioning

(b) After elimination of interiors only the separator groups remain



(c) The Householder transformations leave one V_Σ-node per separator group

(d) Partitioning on next level

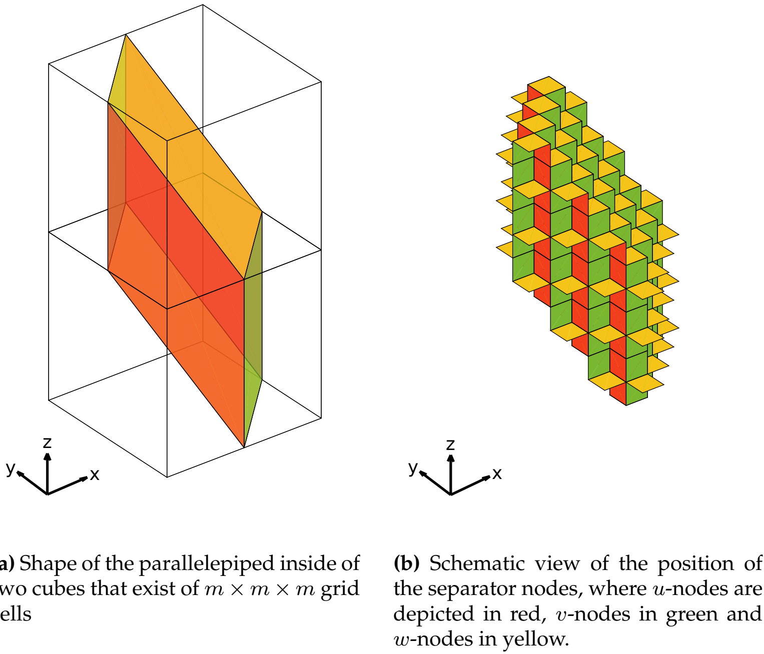
Figure 3.3: Skew partitioning of a 12×12 C-grid discretization of the Navier–Stokes equations into 12 subdomains. The interiors are shown in gray. Velocities of the same (non-gray) color that are on the same face of neighboring cells form a separator group. The red circles are pressure nodes that are retained. This example uses a coarsening factor of 2, i.e. the separators on the next level have twice the length of those on the previous level.

tiple smaller octahedral subdomains. This is of course required for the multilevel method. However, scalability can be achieved by splitting the octahedrons into four smaller tetrahedrons, of which six different types are required to fill 3D space. This would introduce additional separator planes that are similar to the 2D skew case and hence it increases the risk of isolating a pressure node when such planes intersect. Especially planar intersections which are parallel to either of the Cartesian axes have a high risk of producing isolating pressure nodes.

We did indeed not manage to find any scalable tiling using tetrahedrons that would not give rise to any isolated pressure nodes. Moreover, we would like to have a singular subdomain shape that we can use instead of six, since this would greatly simplify the implementation. A lesson we learned is that isolated pressure nodes always seem to arise when having faces that are aligned with the grid. Therefore, we looked into a rotated parallelepiped shape that does not have any faces that are aligned with the grid (Van der Klok, 2017). This shape is shown in Figure 3.4a, where the cubes represent a set of $s_x \times s_x \times s_x$ grid cells. A welcome property of this domain is that its central cross section resembles the proposed skew 2D partitioning method.

A schematic view of the position of the separator nodes is shown in Figure 3.4b. Here we see that on the side that is facing towards us, only half of the w-nodes are displayed. This is because the other half of the separator nodes is behind the u-nodes and v-nodes that are shown. This alternating property is required to make sure that we do not obtain isolated pressure nodes. If, for instance, all w-separator nodes that stick out in the figure were instead flipped to the inside, we would have an isolated pressure node in the bottom row. A consequence of this alternating property is that the w-planes have to be divided into two separate separator groups; one for each of the two neighboring subdomains in which these nodes are actually located.

Another advantage of using a skew domain partitioning is that the amount of communication that is required is reduced compared to a square partitioning. In Bisseling (2004), it is estimated that for the Laplace problem, the communication is asymptotically reduced by a factor of $\sqrt{2}$ for the 2D diamond shape. If we instead compare the diamond shape to a rectangular domain with the same number of nodes (having the same number of nodes with a square domain is impossible), we find that communication is reduced by a factor of $\frac{3}{2}$. In the same manner, we can compare a 3D domain of size $s_x \times s_x \times s_x/2$ to the rotated parallelepiped, and find a factor of $\frac{4}{3}$. We remark that the truncated octahedron that is used in Bisseling (2004) for the 3D domain and has a factor of 1.68 can not be used for our multilevel method, since truncated octahedra are not scalable.



(a) Shape of the parallelepiped inside of two cubes that exist of $m \times m \times m$ grid cells

(b) Schematic view of the position of the separator nodes, where u-nodes are depicted in red, v-nodes in green and w-nodes in yellow.

Figure 3.4: Parallelepiped shape for partitioning the domain.

Numerical results 3.4

A common benchmark problem in fluid dynamics is the lid-driven cavity problem. We consider an incompressible Newtonian fluid in a square three-dimensional domain of unit length, with a lid at the top which moves at a constant speed U. The equations are given by (3.1). No-slip boundary conditions are applied at the walls, meaning that they are Dirichlet boundary conditions, and the equations are discretized on an Arakawa C-grid as described before. We take $n_x = n_y = n_z$ grid points in every direction.

At first, however, we will only look at the Stokes equations of the form

$$\mu \Delta \boldsymbol{u} - \nabla p = 0,$$
$$\nabla \cdot \boldsymbol{u} = 0.$$

This is because our preconditioner is constructed in such a way that memory usage and time cost for both computation and application of the preconditioner should not be influenced by inclusion of the convective term. After

this, we will further investigate the behavior on the lid-driven cavity problem for increasing Reynolds numbers, which constitute harder problems. Therefore we expect an increase in iterations of the iterative solver, but otherwise the same behavior.

For obtaining the exact memory usage, we developed a custom library which overrides all memory allocation routines when linking against it. The library contains a hash table in which the amount of memory that is allocated is stored by its memory address. We keep track of the total amount of memory that is allocated, which is increased on memory allocation, and reduced by the amount that is stored in the hash table when memory is freed. The reason we did this is that existing methods rely on rough estimates of the memory usage of the used data structures, use the data that is available from `/proc/meminfo` which is inaccurate, or actually count memory usage in a similar way as we do (i.e. valgrind), but have a large performance impact.

We perform every experiment twice: once to determine the memory usage, and once to determine the run time, without linking to the memory usage library. This means that when reporting timing results, we are not impacted by the performance impact of tools to determine memory usage. The reason that we are still concerned about their performance impact, even though we developed our own library, is that it adds roughly a constant amount of time per process, which impacts scalability results. The memory usage that we report is the exact difference in memory usage before and after a certain action is performed, e.g. before and after the construction of the preconditioner.

For the timing results, we do not include the time it takes to compute the partitioning. The partitioning is computed by first constructing a template sequentially, which contains all possible nodes of exactly one overlapping subdomain, which may even partially be outside of the domain. This template is then mapped to the correct position for every other overlapping subdomain to determine the interiors and separator groups. However, since the template contains all possible nodes, every time we increase the number of levels by 1, the amount of time to compute this template increases by a factor 8, which is the worst possible scenario. The reason we do not include this in the timing results is that this may be resolved by only computing the partitioning for nodes that are still present in the Schur complement at that level. This would, however, require a full rewrite of the existing partitioning code, while it would not have any impact on the timing results of the rest of the code, since this is completely decoupled. This means that even though the partitioning method does not scale at all, the preconditioning method itself can be studied reliably without including the timing of the partitioning.

For the implementation of the preconditioner and solver, we use libraries from the Trilinos project (Heroux et al., 2005). The libraries we use are Epetra (vector and matrix data structures), IFPACK (direct solver and preconditioning interfaces) (Sala and Heroux, 2005), Amesos (direct solvers) (Sala et al., 2008) and Belos (iterative solvers) (Bavier et al., 2012). As iterative solver we

use GMRES(250) (Saad and Schultz, 1986), as parallel sparse direct solver on the coarsest level we use SuperLU_DIST 6.1 (Liu et al., 2018), and as direct solver for the interior blocks we use KLU (Davis and Natarajan, 2010) with the fill-reducing ordering from De Niet and Wubs (2008).

The benchmark is performed on Cartesius, which is the Dutch national supercomputer. We used the Haswell nodes, which consist of 2 × 12-core 2.6 GHz Intel Xeon E5-2690 v3 (Haswell) CPUs per node with 64 GB per node. For all experiments, we disabled multithreading inside the MPI processes, since this is poorly implemented in Epetra, and causes results to become worse instead of better.

Due to issues with the interconnection between nodes, which are most likely caused by hardware issues, we were not able to run our code on more than 2048 cores. Note that this holds for a large number of applications and not just our application, and has been confirmed to not be an issue with our code. For this reason we do not look at the scalability past $8^3 = 512$ cores, even though we also wanted to add experiments for $8^4 = 4096$ cores. The connection issues that we observed are also likely to have impacted performance of the experiments that we report in this section, especially when larger numbers of cores are used.

3.4.1 Weak scalability

First, we look at results obtained when increasing the grid size n_x at the same rate as the number of used cores n_p, i.e. the problem size on each core is kept constant. The exact configurations that we use are 1 core for $n_x = 16$, 1 core for $n_x = 32$, 8 cores for $n_x = 64$, 64 cores for $n_x = 128$ and 512 cores for $n_x = 256$. The size of the subdomains (the size of the cubes in Figure 3.4a) at the first level is $s_x = 8$, while we choose the coarsening factor to be 2, meaning we increase s_x by a factor of 2 at each level. We perform experiments when keeping the number of levels constant at $L = 2$, and when increasing the number of levels by 1 when doubling the grid size. For the latter, we look at three cases where we retain a different number of separator nodes starting at level 2: 1, 4, and all.

For the fixed number of levels, we expect the number of iterations of the iterative solver to converge to a constant number as the domain size increases. For the case where we increase the number of levels as the domain size increases, we expect the number of iterations to only increase mildly, and we expect that retaining more separator nodes starting at level 2 decreases the number of iterations until it again becomes constant as we retain more and more nodes per separator.

The results are shown in Figure 3.5. We see that indeed the number of iterations becomes constant when fixing the number of levels or when retaining all separator nodes. When increasing the number of levels with the grid size, we see that the number of iterations appears to increase linearly. Interesting

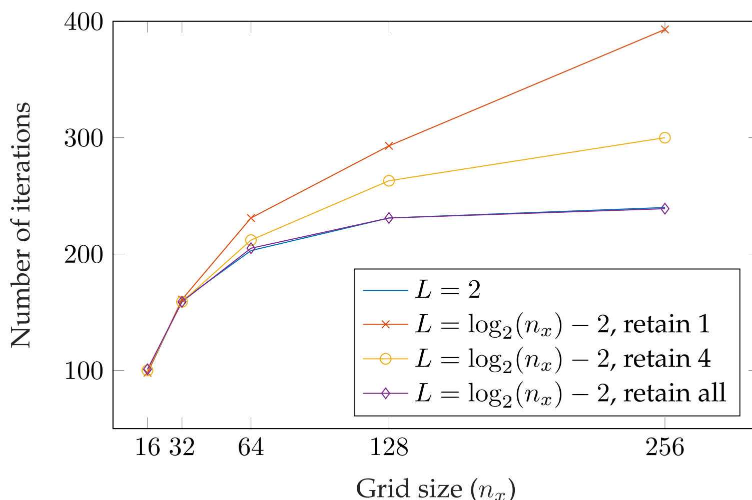


Figure 3.5: Number of iterations of GMRES for the Stokes problem on a grid of size $n_x \times n_x \times n_x$, while keeping the number of levels at $L = 2$, and when increasing the number of levels by 1 when n_x is doubled. A relative residual of 10^{-8} was used as convergence tolerance.

is that when retaining 4 instead of 1 separator node per separator group after level 2, the number of iterations that is required decreases drastically, even though this does not have a significant impact on the memory usage as we will see later.

The computational time of both computing the preconditioner, as well as the solution of the linear system would ideally become constant when keeping the problem size at each core constant while increasing the problem size. However, in practice this is not possible since increasing the number of cores also increases the required amount of communication. The results are shown in Figure 3.6 and Figure 3.7.

When computing the preconditioner, we see that especially at the largest grid sizes, the amount of computational time tends to increase, while for smaller problems it appeared to become constant. This is because for the smaller problems, at most 3 computing nodes were used, meaning very little communication between computing nodes was required. When increasing the number of computing nodes, the amount of required communication increases, which happens mainly at the point where contributions of neighboring subdomains have to be added up in the Schur complement. Since retaining more nodes per level means an increase in the amount of communication, we see that retaining 4 or all nodes takes more time than retaining only 1 node per separator at every level.

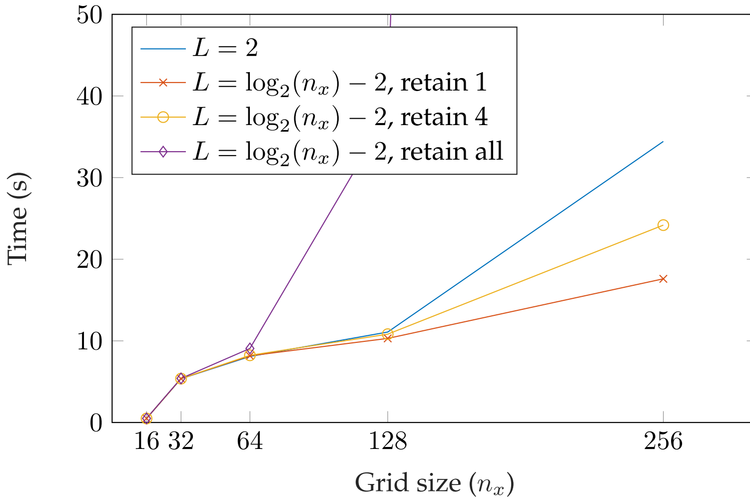


Figure 3.6: Time to compute the preconditioner for the Stokes problem on a grid of size $n_x \times n_x \times n_x$, while keeping the number of levels at $L = 2$, and when increasing the number of levels by 1 when n_x is doubled.

We also notice that retaining all nodes after level 2 is much more inefficient than just using SuperLU_DIST at level 2, meaning that our preconditioner performs very poorly as a direct solver. This is mainly due to the fact that the Schur complement at the last level is quite large and full. The last level Schur complement for a grid of size 256^3 has size 20961×20961 and its factorization is filled with 72% nonzeros. Computing the factorization of this matrix and applying it is therefore really expensive. Using a parallel dense solver instead of SuperLU_DIST should help to make it more efficient. Moreover, since the cost of computing the factorization in a sparse direct solver goes at best with $\mathcal{O}(n^2/n_p) = \mathcal{O}(n_x^3)$ (Liu et al., 2018), we expect that the cost of computing the preconditioner when keeping the number of levels constant, or when retaining all separator nodes, increases with n_x^3.

In Figure 3.7, we show both the time it takes to solve the linear system after computation of the preconditioner, and the time of 1000 applications of the preconditioner, which is not influenced by the number of iterations of the iterative solver and does not include time spent on for instance matrix-vector products and orthogonalization. First of all, we again observe that retaining all separator nodes is a bad idea, since the computation time goes off the chart. For the case where we only use 2 levels, we also see the unwanted behavior of a time that keeps increasing linearly for the total solution time, and superlinearly for the application of the preconditioner, where we would actu-

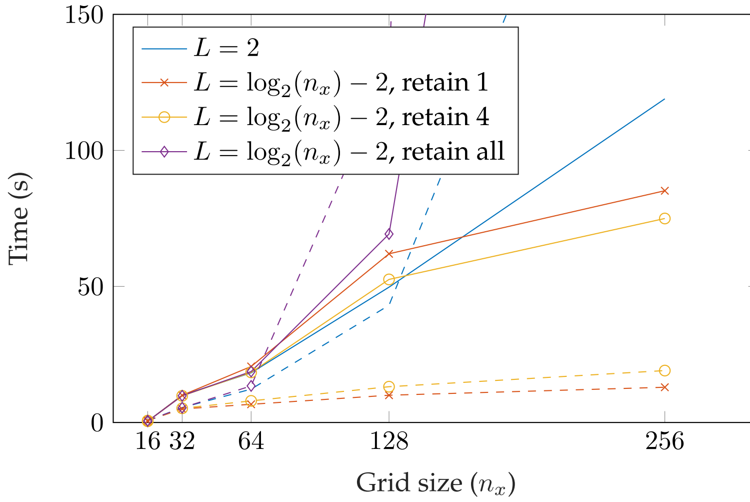


Figure 3.7: Time to solve the linear system with GMRES (lines), and time of 1000 applications of the preconditioner (dashed lines). This is for the Stokes problem on a grid of size $n_x \times n_x \times n_x$, while keeping the number of levels at $L = 2$, and when increasing the number of levels by 1 when n_x is doubled. A relative residual of 10^{-8} was used as convergence tolerance.

ally expect $\mathcal{O}(n^{4/3}/n_p) = \mathcal{O}(n_x)$ behavior from the triangular solve (Liu et al., 2018). This may be caused by disabling multithreading support, which results in a lot of extra communication inside of SuperLU_DIST.

For both cases where we retain 1 and 4 separator nodes after 2 levels, we see that the computational time appears to become constant for larger grid sizes. We also note that even though the case where we retain 4 nodes is slower per application, it is faster in total solution time due to the lower number of iterations that is required, as can also be seen in Figure 3.5, and the fact that applying the preconditioner is so cheap.

In Figure 3.8, we see the average memory usage of the preconditioner per core. Here we again observe that the 2 level case, and the case where we retain all separator nodes after level 3 performs poorly. The case where we retain 1 or 4 separator nodes per separator group after level 2 show similar behavior in terms of memory usage. We see, however, that the memory usage of the latter two cases does not become constant. We will discuss this further in the next section.

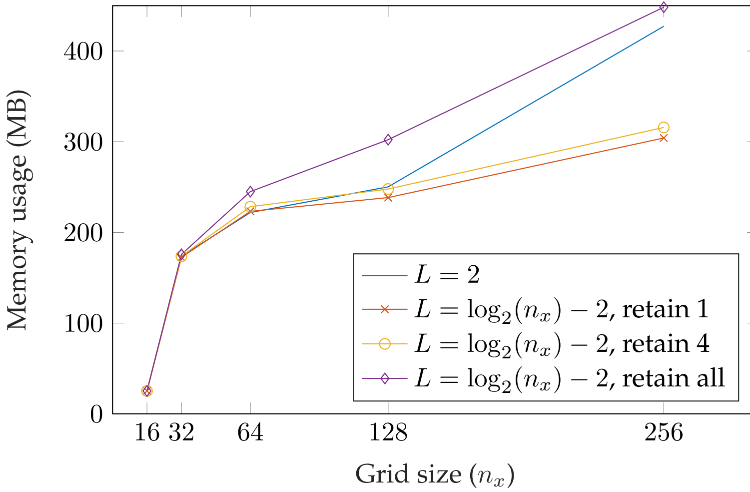


Figure 3.8: Memory usage of the preconditioner per core for the Stokes problem on a grid of size $n_x \times n_x \times n_x$, while keeping the number of levels at $L = 2$, and when increasing the number of levels by 1 when n_x is doubled.

3.4.2 Strong scalability

In this section, we will look at a problem with size $n_x = 128$, with 6 levels and retaining only one node per separator group. We use 1 to 128 cores for the memory usage and 2 to 128 cores for the computational time, with steps of a factor of 2. Reason we do not look at 1 core for the timing is that this configuration caused a memory allocation error in the iterative solver (Belos).

We first look at the total memory usage in Figure 3.9. We observe that there is a large difference between the memory usage on one core, and the memory usage on two cores. This is due to the fact that mainly Epetra uses different implementations of its data structures for serial and parallel computations. We also observe that the total memory usage increases when using more cores. We found that this increase happens when the so called column maps are constructed, which are used in every sparse matrix data structure in Epetra. The matrices in Epetra are distributed by rows, meaning that a row is always stored on a single core. The global indices of local rows of a certain process are stored in the row map. This map is of roughly constant size independent of the number of processes due to its unique nature. The column indices of the nonzeros that are present in one row may, however, not belong to rows that are also stored in the same process. Therefore the corresponding column map, which stores all the column indices, has overlap between different processes, and increases in size when more processes are used. The map is needed in

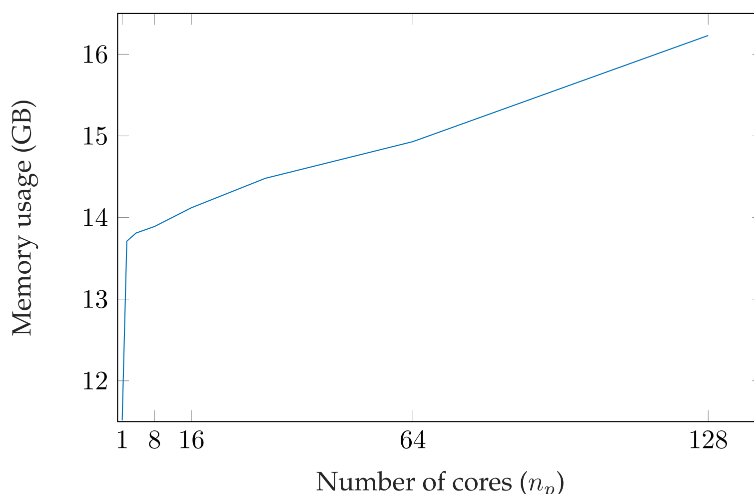


Figure 3.9: Memory usage of the preconditioner for the Stokes problem on a grid of size 128^3, with 1 to 128 cores.

for instance a matrix-vector product. That is, in case of a square matrix, the left- and right-hand side vectors have the same map as the row map of the matrix, so to multiply one row of the matrix with the left hand side vector requires communication with all indices that are stored in the column map. This means that not only the memory usage increases, but also the time cost, since a larger column map means more communication is required. Note that we also see this behavior not only in our preconditioner but also in the storage of the original matrix.

Since the difference in communication between a cubical domain and a parallelepiped shaped domain is a constant factor of $\frac{3}{4}$, we may instead look at a cubical domain to explain this behavior. Say we have a subdomain of size $s_x \times s_y \times s_z$, then communication is required for $2s_xs_y + 2s_xs_z + 2s_ys_z$ grid cells. If we now split this subdomain in the x-direction, then we require communication for $2s_xs_y + 2s_xs_z + 4s_ys_z$ grid cells. If we now assume that $s_x = s_y = s_z$, we see that communication increases with a factor $\frac{4}{3}$, meaning that we expect the size of the column map increases with a factor $\frac{4}{3}$ when doubling the number of cores. This is, however, not what we observe. When increasing the number of cores even further, we even see a factor of 10 instead of $\frac{4}{3}$. This appears to be because of caching that happens inside of both Epetra and MPI itself when the column map is constructed. We checked that the actual column map shows the $\frac{4}{3}$ behavior that we expect.

In Figure 3.10 we look at the speedup when using more cores. Ideally, we would see that using twice the number of cores means half of the computa-

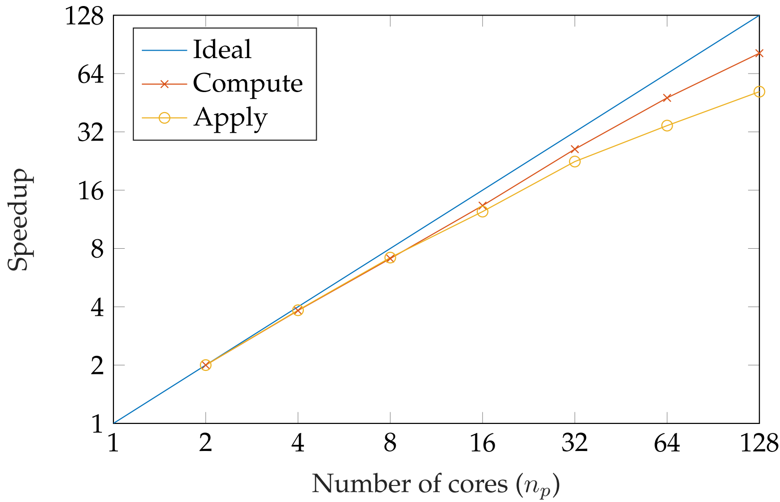


Figure 3.10: Speedup of computation and application of the preconditioner for the Stokes problem on a grid of size 128^3, with 2 to 128 cores.

tional time. We plotted this ideal line for reference. We see that the speedup of the computation of the preconditioner is very close to this ideal line. The application of the preconditioner is a bit further from the ideal line. This is again mostly due to the communication that is required for the non-local column indices, but may also be affected by the issues with the interconnection between nodes that we discussed earlier.

3.4.3 Lid-driven cavity

In the previous sections we determined the strong and weak scalability properties of the preconditioner, which means that we can now continue with the robustness of the solver on the lid-driven cavity problem with increasing Reynolds numbers. We perform a continuation with steps of Re $= 100$ starting from the solution of the Stokes problem. We show the results of the first iteration of Newton at Re $= 500$ and Re $= 2000$. Reason we go up to Re $= 2000$ is that a Hopf bifurcation is located between Re $= 1900$ and Re $= 2000$, which is of interest (Baars et al., 2019b).

In Figure 3.11 and Figure 3.12, we show the results at Reynolds number 500, which show good correspondence with the results on the Stokes problem. Again the number of iterations appears to become constant when the number of levels is kept the same, and retaining more nodes gives better convergence. The main difference is that more iterations are needed, since higher

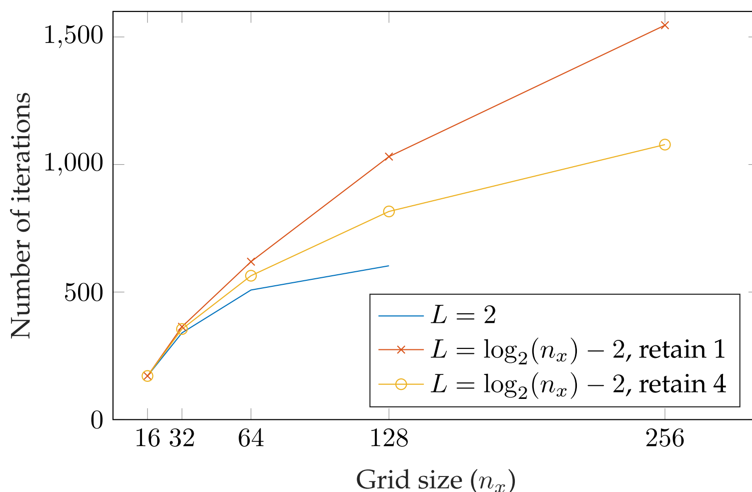


Figure 3.11: Number of iterations of GMRES for the lid-driven cavity problem at Re = 500 on a grid of size $n_x \times n_x \times n_x$, while keeping the number of levels at $L = 2$, and when increasing the number of levels by 1 when n_x is doubled. A relative residual of 10^{-8} was used as convergence tolerance.

Reynolds numbers constitute harder problems. Also interesting is that for the 256^3 grid with 2 levels, SuperLU_DIST aborted without any error message, also when using different amounts of cores. In Baars et al. (2019b), we observed convergence, however, so we suspect this is due to a change that was made in version 6 of SuperLU_DIST.

The results at Reynolds number 2000 are shown in Figure 3.13 and Figure 3.14. We again observe that the number of iterations is much larger. What is odd, however, is that retaining more nodes now actually gives worse convergence. This may be because we use GMRES(250) instead of GMRES due to memory limitations, and therefore do not preserve the convergence properties of GMRES. This effect is more prevalent for this problem because of the large number of iterations that is required. We do observe that for the first 250 iterations, retaining 4 nodes after level 2 gives rise to better convergence, as we expected.

In Table 3.1 we show results for Reynolds number 500 using the block preconditioner LSC (Least-Squares Commutator) implemented in Teko (Cyr et al., 2012). We chose Teko for comparison because it is available in Trilinos and can therefore easily be coupled to our code. We also tried other preconditioners, but those did unfortunately not yield convergence. The stopping criterion of the linear solver is 10^{-8}, as before.

Compared to our method, we see that Teko has much more difficulty with

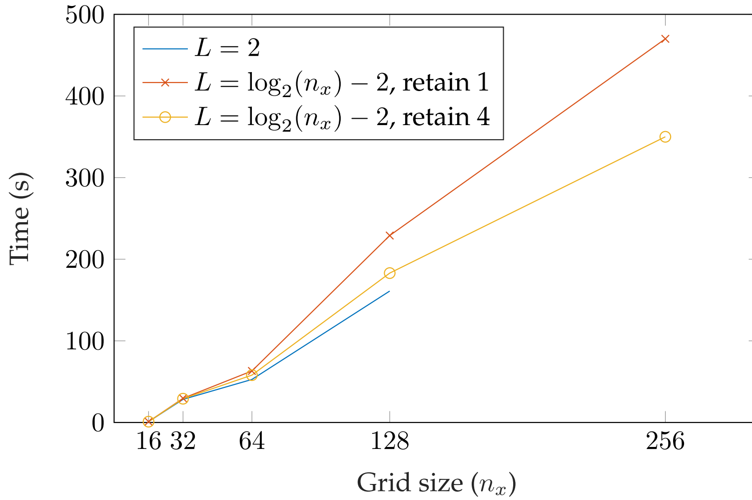


Figure 3.12: Time for GMRES to converge for the lid-driven cavity problem at Re $= 500$ on a grid of size $n_x \times n_x \times n_x$, while keeping the number of levels at $L = 2$, and when increasing the number of levels by 1 when n_x is doubled. A relative residual of 10^{-8} was used as convergence tolerance.

n_p	n_x	L	its	t_c (s)	t_s (s)
1	16	2	142	0.12	0.77
1	32	2	187	9.88	18.80
8	64	3	245	1511.12	313.00

Table 3.1: Performance of Teko with LSC preconditioner for the lid-driven cavity problem at Re $= 500$ on a grid of size $n_x \times n_x \times n_x$. Here n_p is the number of cores, L is the number of levels, its is the number of iterations, t_c is the time to compute the preconditioner and t_s is the time for solving the linear system.

the grid refinement, leading to a huge computational cost already at a grid size of 64^3. A crude computation shows that per grid point the method becomes about 65 times more expensive per iteration. This must be attributed to slow convergence of algebraic multigrid on the subblocks. One of these blocks is the $L + N$ block from (3.2), with N indefinite, and this is something that is difficult for a standard AMG method. We chose the number of levels to be 2 for grid sizes 16^3 and 32^3, and 3 levels for 64^3 since these seemed to give the optimal results. For Reynolds number 2000, we did not observe convergence past the 16^3 grid.

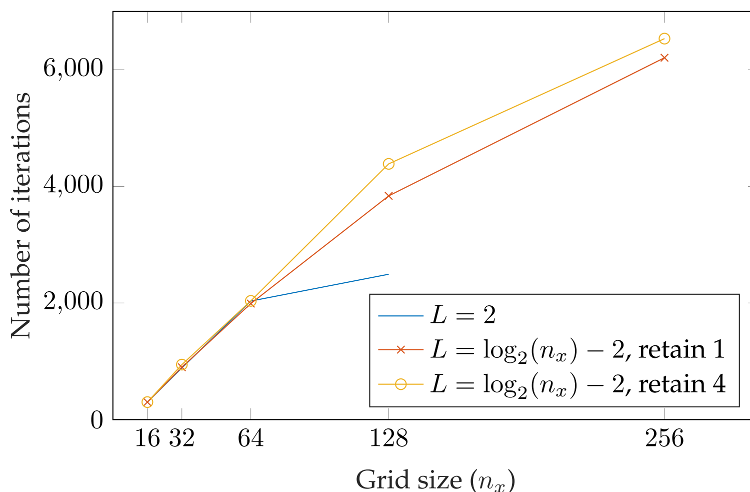


Figure 3.13: Number of iterations of GMRES for the lid-driven cavity problem at Re $=$ 2000 on a grid of size $n_x \times n_x \times n_x$, while keeping the number of levels at $L = 2$, and when increasing the number of levels by 1 when n_x is doubled. A relative residual of 10^{-8} was used as convergence tolerance.

3.5 Summary and Discussion

We presented a robust method for solving the steady, incompressible Navier–Stokes equations, which makes use of parallelepiped shaped overlapping subdomains. The interiors of these overlapping subdomains can be eliminated in parallel. On the interfaces of the subdomains, Householder transformations are applied to decouple all but one velocity node from the pressure nodes, after which all decoupled nodes can also be eliminated in parallel. The key to the multilevel approach is the resulting reduced Schur complement matrix, which has the same structure as the original matrix. Therefore we can recursively apply the preconditioner to this matrix. The shape of the subdomains makes it such that pressure nodes are not isolated in the factorization process, which would result in a singular Schur complement matrix.

Our weak scalability experiments show that if the number of levels of the multilevel approach is kept constant while increasing the problem size, grid independent convergence is obtained. We also show that by increasing the number of levels and processors as the problem size increases, we only require a small amount of additional time and memory for both the factorization and solution process. Moreover, by increasing the number of nodes that is retained per separator after level 2, we can further decrease the time that is required to solve the system.

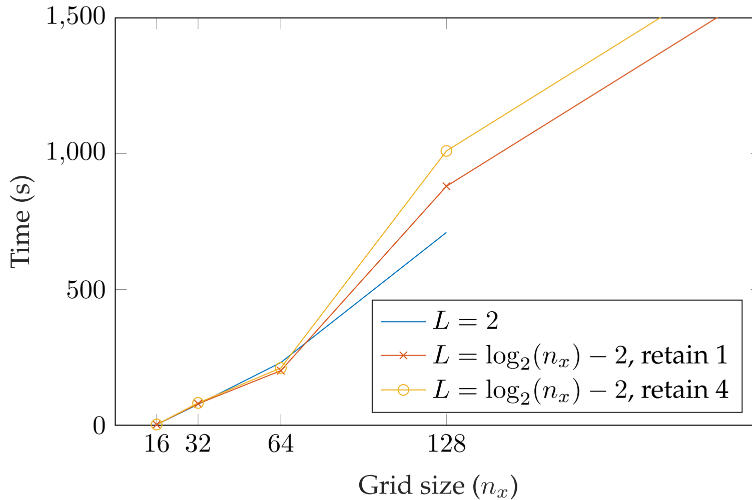


Figure 3.14: Time for GMRES to converge for the lid-driven cavity problem at Re $=$ 2000 on a grid of size $n_x \times n_x \times n_x$, while keeping the number of levels at $L = 2$, and when increasing the number of levels by 1 when n_x is doubled. A relative residual of 10^{-8} was used as convergence tolerance.

Our strong scalability experiments show that the time that is required to compute the preconditioner scales very well. The memory that is used for the preconditioner scales slightly less optimal, but this can be explained by the caching of communication related data structures. The slightly less optimal time it takes to apply the preconditioner can also be explained by means of the increased communication.

We also showed that the preconditioner gives rise to convergence of GMRES for the lid-driven cavity problem at high Reynolds numbers, where other methods, such as the LSC preconditioner that in implemented in Teko, fail to do so.

This leads us to conclude that we implemented a robust solution method for the Navier–Stokes equations on staggered grids which shows good parallel scalability. In the future we want to apply our preconditioner in ocean-climate models. The fact that it has proven to be robust for the lid-driven cavity problem with higher Reynolds numbers leads us to believe that it will also perform well for the ocean model described in Section 2.5.2.