Parallel Preconditioners for Stokes Flow

A. Wildeman

August 27, 2009

NACM
Applied Mathematics
University of Twente

Master's Thesis

Parallel Preconditioners for Stokes Flow

Author:
Albert Wildeman

Supervisors:
Prof.dr.ir. J.J.W. van der Vegt
Dr. M.A. Bochev

August 27, 2009

Contents

Summary

This report compares parallel preconditioners for 3D Stokes flow. Its motivation lies in Amphi3D, a diffuse interface algorithm simulating three dimensional interfacial dynamics of complex fluids, the parallelization of which has to date been ineffective due to the lack of an efficient parallel preconditioner. The approach taken here is to evaluate the Block Jacobi, Parallel ILU and Sparse approximate inverse parallel preconditioners on the reduced problem of Stokes flow for a small range of Reynolds numbers. Although surprisingly good results are achieved without preconditioning, it is the sparse approximate inverse preconditioner that delivers the most promising results.

Chapter 1

Introduction

Amphi (Adaptive Meshing for ϕ, the phase-field parameter) is a numerical algorithm for the solution of two-phase complex fluid flows. The use of a phase field parameter to distinguish between the two phases is its key feature, but the extremely fine mesh this requires for adequate resolution of the thin interfacial layer results in very large meshes, despite the use of adaptive meshing. As this requirement is even more pronounced in 3D, only very small 3D problems can be modeled with sufficient accuracy at reasonable computation times. In order to enable its application in the exploration of new and physically relevant flow problems, a parallel version of the algorithm was recently implemented. The approach was to leave the sequential code largely intact, and handle only the solution of the linear systems, which accounts for the bulk of the overall computational load, in parallel. The set-up of the linear systems and preconditioners, as well as the application of Krylov iterative solvers, is done through PETSc, a Portable, Extensible Toolkit for Scientific Computation [4]. This is essentially a library of functions, such as parallel linear and nonlinear solvers, built on top of MPI, the Message Passing Interface.

In this parallel code, the preconditioners available through PETSc were problematic and ineffective, but their parameter spaces are vast and were not fully explored. The goal of this project is to test and evaluate parallel preconditioners for Stokes flow on unstructured grids. Because Stokes flow is an important building block of the system of equations arising in Amphi3D, evaluation of parallel preconditioners on this simpler problem can subsequently serve as a basis for further selection or development of an effective preconditioner. Once it is clear how different preconditioners perform for Stokes flow, an effective preconditioner for the full equations can be determined in a structured manner by gradually increasing the complexity of the equations while re-selecting or modifying the preconditioner and its parameters.

Several surveys [13, 57, 106] of parallel preconditioners have been published, but publications featuring direct numerical comparison of multiple parallel preconditioners are less common. Two papers by Ma [84, 85] approach the nature of this report, but the lack of the Stokes flow case and, much more imporantly, the dominant role of the reordering schemes associated with regular grids prevent a meaningful comparison to these papers.

After an introductory description of the Amphi3D algorithm and a general discussion of parallel algebraic preconditioning, the Stokes flow test case is detailed as well as the particular preconditioners tested. The concluding chapters contain the results obtained, a review of their performance and an evaluation of their potential.

Chapter 2

Amphi3D

In a system of two immiscible fluids, the interface plays a central role in the fluid dynamics and rheology. The mathematics of such a system is generally complicated by the movement of the interface, even more so when break-up or coalescence have to be accounted for. Two general strategies have emerged to encompass the evolution of the interface; interface tracking and phase-field methods. The former is a Lagrangian approach which directly tracks the interface with a fixed set of gridpoints, a process which becomes problematic when break-up or coalescence occur. In contrast, these phenomena are incorporated naturally in a phase-field framework such as Amphi, where the morphology is described by a continuous phase-field parameter. Furthermore, a diffuse interface is more physical in the sense that real interfaces have a finite thickness, or mixing region. The interfacial thickness used in simulations is, however, unphysically large for numerical reasons, but it leads to an energy-based variational formalism which allows a natural inclusion of complex fluid rheologies, on the condition that they can be described by a free energy.

Amphi3D can essentially be split into three different parts: the diffuse interface method, its finite element formulation and the adaptive meshing scheme.

2.1 Diffuse Interface Method

The diffuse interface model was previously described by [3, 82, 69] and extended to incorporate non-Newtonian fluids by [131, 133]. The mixture of a Newtonian and an Oldroyd-B fluid will be used to outline their results. The two fluids are immiscible except for a very thin interfacial region, where some mixing does occur and mixing energy is contained. An Oldroyd-B fluid is a dilute suspension of polymer chains modeled as Hookean dumbbells in a Newtonian solvent. The free energy of the mixture comprises both the elastic energy of these dumbbells and the

mixing energy. The phase-field variable ϕ is defined to indicate the concentrations of the Newtonian and the Oldroyd-B fluid with $(1+\phi)/2$ and $(1-\phi)/2$, respectively. The mixing energy takes the Ginzburg-Landau form [24]:

$$f_{\text{mix}}(\phi, \nabla\phi) = \frac{1}{2}\lambda|\nabla\phi|^2 + \frac{\lambda}{4\epsilon^2}(\phi^2-1)^2 \tag{2.1}$$

where λ represents the mixing energy density and ϵ is the capillary width, proportional to the width of the diffuse interface. The surface tension can be extracted from the ratio λ/ϵ as $\epsilon \to 0$ [69, 131]:

$$\sigma = \frac{2\sqrt{2}}{3}\frac{\lambda}{\epsilon} \tag{2.2}$$

Through Young's equation, $\sigma\cos\theta_S = \sigma_{w2} - \sigma_{w1}$, the fluid-solid interfacial tensions σ_{w1} and σ_{w2} for the two fluids determine the static contact angle θ_S. The diffuse-interface wall energy thus becomes [35, 70, 132]

$$f_w(\phi) = -\sigma\cos\theta_S\frac{\phi(3-\phi^2)}{4} + \frac{\sigma_{w1}+\sigma_{w2}}{2}. \tag{2.3}$$

The evolution of ϕ is governed by the Cahn-Hilliard equation:

$$\frac{\partial\phi}{\partial t} + \boldsymbol{v}\cdot\nabla\phi = \gamma\nabla^2 G \tag{2.4}$$

where the chemical potential reads

$$G = \frac{\delta\int f_{\text{mix}}\, d\Omega}{\delta\phi} = \lambda\left[-\nabla^2\phi + \frac{\phi(\phi^2-1)}{\epsilon^2}\right] \tag{2.5}$$

and γ is the mobility [131].

The elastic energy of the Oldroyd-B fluid is described in [24]:

$$f_d = \frac{1+\phi}{2}\int_{R^3}\left(kT\ln\Psi + \frac{1}{2}H\boldsymbol{Q}\cdot\boldsymbol{Q}\right)\Psi\, d\boldsymbol{Q} \tag{2.6}$$

with n denoting the number density of the dumbbells, k the Boltzmann constant, T the temperature, H the elastic spring constant, $\Psi(\boldsymbol{Q})$ the configuration distribution and $\boldsymbol{Q}$ is the vector connecting both ends of the spring.

The stress tensor τ_d is obtained by applying a variational procedure to the total free energy, and satisfies the Maxwell equation [131, 133]:

$$\tau_d + \lambda_H\tau_{d(1)} = \mu_p[\nabla\boldsymbol{v} + (\nabla\boldsymbol{v})^T] \tag{2.7}$$

where $\lambda_H = \zeta/(4H)$ is the relaxation time, the subscript$_{(1)}$ denotes the upper convected derivative, ζ is the friction coefficient between the dumbbell beads and the suspending solvent and $\mu_p = nkT\lambda_H$ is the polymer viscosity. Upon addition of the viscous stresses, the total stress tensor without the contribution of the mixing energy becomes

$$\tau = \left(\frac{1-\phi}{2}\mu_n + \frac{1+\phi}{2}\mu_s\right)[\nabla \boldsymbol{v} + (\nabla \boldsymbol{v})^T] + \frac{1+\phi}{2}\tau_d \tag{2.8}$$

with μ_n and μ_s denoting the viscosities of the Newtonian component and the Newtonian solvent, respectively. With this stress tensor, the equations of motion can be written as

$$\rho\left(\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v}\cdot\nabla\boldsymbol{v}\right) = \nabla\cdot(-p\boldsymbol{I} + \tau) + G\nabla\phi + \rho\boldsymbol{g} \tag{2.9}$$

$$\nabla\cdot\boldsymbol{v} = 0 \tag{2.10}$$

where $\rho = \frac{1+\phi}{2}\rho_n + \frac{1-\phi}{2}\rho_s$ with ρ_n and ρ_s the densities of the Newtonian and Oldroyd-B fluids, respectively, and similarly $\mu = \frac{1+\phi}{2}\mu_s + \frac{1-\phi}{2}\mu_n$.

Furthermore, $\boldsymbol{g}$ is the gravitational acceleration and $G\nabla\phi$ is the interfacial stress, the diffuse-interface representation of the inertial forces on both fluids [131].

The boundary conditions come to

$$\boldsymbol{v} - \boldsymbol{v}_w = 0 \tag{2.11}$$

on the solid wall boundary $(\partial\Omega)_u$, with $\boldsymbol{v}_w$ denoting the velocity of the wall, and

$$\boldsymbol{n}\cdot\nabla G = 0 \tag{2.12}$$

$$\lambda\boldsymbol{n}\cdot\nabla\phi + f_w^{'}(\phi) = 0 \tag{2.13}$$

on the entire boundary $\partial\Omega$, with $\boldsymbol{n}$ denoting the unit normal to the boundary. The no-slip condition, (2.11), leaves Cahn-Hilliard diffusion as the sole source of contact line motion. Mass conservation of both fluids follows from (2.12), which ensures zero flux through the solid wall and mass conservation of each fluid. Finally, (2.13) is the natural boundary condition from the variation of the wall energy f_w, and the left-hand side represents the surface chemical potential. Thus, this condition entails that the fluid layer is always at equilibrium with the solid substrate.

2.2 Finite Element Formulation

The discretization of the governing equations follows the standard Galerkin formalism [66]. However, the use of C^0 elements, which are smooth within in each

element and continuous across element boundaries, do not accomodate the representation of spatial derivates of order higher than two. To circumvent this issue, the Cahn-Hilliard equation (2.4) is decomposed into two second-order equations:

$$\frac{\partial \phi}{\partial t} + \boldsymbol{v} \cdot \nabla \phi = \frac{\gamma \lambda}{\epsilon^2} \Delta(\psi + s\phi) \tag{2.14}$$

$$\psi = -\epsilon^2 \Delta \phi + (\phi^2 - 1 - s)\phi \tag{2.15}$$

The parameter s was introduced to enhance stability of the numerical method and in all applications to date [133, 135], $s = 0.5$ has been adequate. A convenient side effect of this value is that it reduces the chemical potential to $G = \lambda(\psi + s\phi)$.

The weak solution sought is $(\boldsymbol{v}, p, \tau_d, \phi, \psi) \in \mathcal{U} \times \mathcal{P} \times \mathcal{T} \times \mathcal{S} \times \mathcal{S}$, where for 3D flows, the solution spaces satisfy $\mathcal{U} \in H^1(\Omega)^3$, $\mathcal{P} \in L^2(\Omega)$, $\mathcal{T} \in L^2(\Omega)^6$, and $\mathcal{S} \in H^1(\Omega)$. The weak form of the governing equations with basis functions $(\tilde{\boldsymbol{v}}, \tilde{p}, \tilde{\tau}, \tilde{\phi}, \tilde{\psi})$, is derived in appendix A.1:

$$\int_\Omega \left\{ \left[\rho \left(\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \nabla \boldsymbol{v} - \boldsymbol{g} \right) - G \nabla \phi \right] \cdot \tilde{\boldsymbol{v}} + (-p\boldsymbol{I} + \tau) : \nabla \tilde{\boldsymbol{v}} \right\} d\Omega = 0 \tag{2.16}$$

$$\int_\Omega (\nabla \cdot \boldsymbol{v}) \tilde{p} \, d\Omega = 0 \tag{2.17}$$

$$\int_\Omega \left\{ \tau_d + \lambda_H \tau_{d(1)} - \mu_p \left[\nabla \boldsymbol{v} + (\nabla \boldsymbol{v})^T \right] \right\} : \tilde{\tau} \, d\Omega = 0 \tag{2.18}$$

$$\int_\Omega \left[\left(\frac{\partial \phi}{\partial t} + \boldsymbol{v} \cdot \nabla \phi \right) \tilde{\phi} + \frac{\gamma \lambda}{\epsilon^2} \nabla(\psi + s\phi) \cdot \nabla \tilde{\phi} \right] d\Omega = 0 \tag{2.19}$$

$$\int_\Omega \left\{ \left[\psi - (\phi^2 - 1 - s)\phi \right] \tilde{\psi} - \epsilon^2 \nabla \phi \cdot \nabla \tilde{\psi} \right\} d\Omega - \int_{\partial\Omega} \frac{\epsilon^2}{\lambda} f_w'(\phi) \tilde{\psi} \, dS = 0 \tag{2.20}$$

The associated boundary conditions take the following form:

$$\boldsymbol{v} - \boldsymbol{v}_w = 0 \qquad \text{on } (\partial\Omega)_u \tag{2.21}$$

$$(-p\boldsymbol{I} + \tau) \cdot \boldsymbol{n} = 0 \qquad \text{on } (\partial\Omega)_\tau \tag{2.22}$$

$$\tau_d - \tau_{in} = 0 \qquad \text{on } (\partial\Omega)_{in} \tag{2.23}$$

$$\nabla \phi \cdot \boldsymbol{n} + f_w'(\phi)/\lambda = 0 \qquad \text{on } \partial\Omega \tag{2.24}$$

$$\nabla(\psi + s\phi) \cdot \boldsymbol{n} = 0 \qquad \text{on } \partial\Omega \tag{2.25}$$

where $\partial\Omega = (\partial\Omega)_u \bigcup (\partial\Omega)_\tau$, $(\partial\Omega)_u \bigcap (\partial\Omega)_\tau = \emptyset$ and $(\partial\Omega)_{in}$ is the inflow boundary. Note that the natural boundary condition (2.24) was used in the derivation of (2.20).

Piecewise quadratic (P2) elements for $\boldsymbol{v}$, ϕ and ψ and piecewise linear (P1) elements for p and τ_d are used for the spatial discretization on an unstructured

tetrahedral mesh. Upon spatial discretization of (2.16-2.20) the nonlinear algebraic system is cast in the form

$$\boldsymbol{\Lambda} \cdot \left(\frac{\partial \boldsymbol{U}}{\partial t} \right)^{n+1} + \boldsymbol{F}\left(\boldsymbol{U}^{n+1}\right) = 0 \tag{2.26}$$

where $\boldsymbol{U}$ is the solution vector, $\boldsymbol{\Lambda}$ a diagonal matrix with only zeros and ones on the diagonal, depending on the appearance of the corresponding component of $\boldsymbol{U}$ in a time derivative. All other terms are collected in $\boldsymbol{F}$. A fully implicit second-order scheme is applied to discretize the time derivative, upon which (2.26) is solved with an inexact Newton method using backtracking for improved convergence and stability, as described in [53]. The common technique of using the Jacobian matrix within the Newton method for a number of iterations rather than updating it for every iteration is employed to reduce the computational effort.

2.3 Adaptive Mesh Generation

To achieve fine resolution of the interface with realistic mesh sizes, Amphi employs the GRUMMP (Generation and Refinement of Unstructured, Mixed-Element Meshes in Parallel) adaptive meshing scheme [59]. Rather than tracking the interface, the mesh is principally static but coarsened and refined as the interface moves through the domain. As GRUMMP takes a scalar field L_S to control the mesh size using Delaunay refinement, it is natural to use the gradient of the phase-field variable ϕ to determine this scalar field. Specifically, the form used for the 2D algorithm is left unaltered in the 3D version:

$$L_S(x, y, z) = \frac{1}{|\nabla\phi|\frac{\sqrt{2}}{C} + \frac{1}{h_\infty}}, \tag{2.27}$$

where C is a parameter controlling the resolution of the interface and h_∞ is the mesh size in the bulk. At the interface, the thickness of which scales with ϵ, L_S becomes $h_1 \approx C \cdot \epsilon$. So far, simulations have been run with values for C ranging from 0.5 to 1, and proper mesh resolution was achieved for $C = h_1 \leq \epsilon$. As the interface has a thickness of approximately 7.5ϵ, it will be covered by roughly 10 gridpoints. Furthermore, h_∞ is actually split into h_2 and h_3 to allow different mesh densities in the bulk of each fluid, and there is another parameter controlling the sensitivity of the intended mesh density to the distance from the interface.

2.4 Parallelization of Amphi3D

The original sequential implementation of Amphi3D suffers from severe computational limits on its mesh sizes, both in memory and computational effort, and

it is for this reason that a parallel version of the code was designed. To this end, it was decided to employ PETSc, a Portable, Extensible Toolkit for Scientific Computation [4].

The PETSc manual states quite clearly that parallelization through the use PETSc should not take the form where PETSc is called to solve a linear system in parallel in an otherwise sequential code [4], as matrix assembly will take too much time in this scenario. Instead, PETSc, and therefore parallelization, should be involved at least in the matrix assembly.

In brief, the Amphi3D algorithm combines a finite-element flow solver and the GRUMMP adaptive meshing scheme. Combined with a Crank-Nicholson temporal discretization, this leads to a nonlinear system which is solved by an inexact Newton method. Importantly, the nonlinear system is not explicitly available in the code, and it would take considerable effort to write a function representing this nonlinear system. However, such a function is prerequisite to invocation of a PETSc nonlinear solver. Therefore, PETSc has to be involved at a lower level, that is within the inexact Newton method. Each Newton iteration consists of the assembly of a system matrix and the solution of the corresponding linear system, which, as mentioned earlier, is also the lowest level at which PETSc parallelization can be efficient. This motivated the decision to use PETSc, that is to parallelize, at the level of matrix assembly and solution of the linear system within each Newton iteration. Details of the incorporation of PETSc in the parallel code are collected in appendix B.

The solution of the linear systems is handled by KSP, the interface which provides access to all Krylov linear solvers within PETSc. The primary choices to be made are those for a specific Krylov iterative solver and preconditioner. Both the sequential and parallel codes use BiCG (Biconjugate gradient method [10]) and GMRES (Generalized minimal residual method [104]) Krylov solvers. Much more importantly, the sequential ILU preconditioner is not available for parallel application, and a different preconditioner therefore has to be used. To date, this has proven challenging and no preconditioner has been found which provides increased performance over the sequential code, or even the parallel code running on a single processor.

Chapter 3

Parallel Algebraic Preconditioning

The convergence rate of Krylov subspace methods is depends strongly on the properties of the linear system it is applied to. In particular, performance improves as the eigenvalues are more tightly clustered and often as they are closer to unity. Preconditioning is the process of transforming the original linear system $Ax = b$ to an equivalent one, with favorable properties facilitating the convergence of a Krylov subspace method applied to solve the system.

Preconditioning is widely regarded as the most important part in the design of an efficient iterative solution strategy [57, 73] as well as the most difficult to parallelize [13, 124]. In fact, solution of many problems only becomes feasible upon its application. The importance of preconditioning was sharply indicated by Trefethen and Bau in [119]:

> In ending this book with the subject of preconditioners, we find ourselves at the philosophical center of the scientific computing of the future. (...) Nothing will be more central to computational science in the next century than the art of transforming a problem that appears intractable into another whose solution can be approximated rapidly. For Krylov subspace matrix iterations, this is preconditioning.

Preconditioning is usually effected by a matrix called the preconditioner. This nonsingular matrix, M, can be applied to the original linear system to transform it into

$$M^{-1}Ax = M^{-1}b \tag{3.1}$$

Which does not alter the solution x but is intended to improve the convergence rate of the Krylov subspace method if the spectral properties of (3.1) are superior

to those of the original linear system. When M^{-1} is explicitly known, such as for polynomial or sparse approximate inverse preconditioners, M itself is never constructed. On the other hand, M^{-1} is usually never explicitly formed if M is given. System (3.1) is the result of left preconditioning, but right preconditioning, resulting in

$$AM^{-1}y = b, \qquad x = M^{-1}y \tag{3.2}$$

and split preconditioning,

$$M_1^{-1}AM_2^{-1}y = M_1^{-1}b, \qquad x = M_2^{-1}y \tag{3.3}$$

are also commmonly used, the latter intended primarily to preserve symmetry or near-symmetry when it is present in the orginal system. Note that in (3.3), the preconditioner is $M = M_1M_2$. Which of these types of preconditioning is most effective depends primarily on the Krylov subspace method used and the properties of the system matrix. GMRES, for example, tends to favor right preconditioning [13]. The system matrices of the transformed systems, $M^{-1}A$, AM^{-1} and $M_1^{-1}AM_2^{-1}$ are similar and therefore have the same spectrum. As a result, the convergence of the CG method in exact arithmetic is identical for each system if both A and M are symmetric positive definite. On the other hand, the behavior of GMRES and other Krylov subspace methods can depend strongly on left or right positioning of the same preconditioner [123]. A striking example of such dependence can be found on p.66 of [79].

Importantly, the system matrices of the transformed systems are never explicitly calculated. Rather, matrix-vector products and the solution of linear systems of the form $Mz = b$ suffice, and in the case where M^{-1} is known explicitly only matrix-vector products are required.

The choice for a particular preconditioner is necessarily a compromise between three base criteria [13, 122]:

- The Krylov subspace method should converge rapidly for the preconditioned system
- The preconditioner should be inexpensive to construct
- The operator M^{-1} should be inexpensive to apply

The first property aims at a reduction of the number of iterations necessary to solve a particular linear system with prescribed accuracy; this is the raison d'etre of preconditioners. Ideally, $M^{-1} = A^{-1}$, such that the system matrix of the transformed system is the identity and the solution of the system is trivial. However, the construction of such a preconditioners constitutes the extremely costly inversion of the original matrix A, the evasion of which is the very motivation for Krylov

subspace methods and preconditioners to begin with. Therefore, the second and third property voice the concern for excessive overhead before the iterative process can commence and the overhead in each iteration, respectively. Naturally, the balance between these issues is aimed at minimization of the overall computing time. Although most preconditioning techniques are aimed at the approximation $M^{-1} \approx A^{-1}$, or equivalently, at bringing all eigenvalues of the preconditioned linear systems to unity, interesting exceptions exist; see [92] for an example. Situations where a series of linear systems with identical system matrix and different right-hand sides has to be solved constitute a strong shift in this balance, as the expense of preconditioner construction can be amortized by its application to repeated solves. This tends to be the case when the linear systems arise from some variant of Newton's method.

For parallel preconditioners, great care must be taken to limit the amount of cross-process communication, both in its construction and application. The latter suggests that the preconditioner should be as close to block diagonal as possible, with each block corresponding to the unknowns on one processor.

A first classification of preconditioners can be made in terms of generality. Many applications involving PDEs employ preconditioners which are near-optimal for a very narrow set of problems. These are generally based on complete knowledge of the underlying problem; not only in terms of its governing equations, but also with respect to the geometry, the boundary conditions and the details of the discretization. Preconditioners based on a simplified version of the governing equations or lower order discretization belong to this class. Multigrid preconditioners are often employed in this manner [91], but can also be fully algebraic [115].

The primary drawbacks of the problem-specific approach is that it requires the solver developer to have a complete understanding of the problem, and that the often considerable effort in designing the preconditioner can be applied only to a very narrow range of problems as a result of the sensitivity of the effectiveness of these preconditioners to the details of the problem. These arguments have motivated the enduring quest for widely applicable, purely algebraic preconditioning methods which use nothing but the system matrix. Although this approach cannot be expected to rival the quality of application-specific preconditioners, they can achieve reasonable efficiency and offer greater flexibility when details of the underlying problems are altered. They are particularly well-suited for problems with unstructured meshes, and often can be fine-tuned to fit the underlying problem, thus blurring the distinction between general and application-specific preconditioners. Furthermore, specific preconditioners are generally based on existing algebraic methods, as exemplified by [125].

Application-specific preconditioners are beyond the scope of this report, and

the focus of this chapter lies with the three primary classes of algebraic preconditioners: incomplete factorization, sparse appoximate inverses and algebraic multigrid.

3.1 Incomplete decomposition

Incomplete decomposition preconditioning techniques are among the oldest and most frequently used, in both sequential en parallel settings. Guassian elimination generally generates considerable fill-in when decomposing a sparse matrix. Sparsity-preserving pivoting techniques have been developed, but for many classes of linear systems, the resulting sparse direct method tends to be far too costly, primarily in terms of memory, to execute for large sparse matrices. Incomplete decomposition is based on the notion of dropping part of the fill-in which is in some sense unimportant to the quality of the decomposition; creating a preconditioner $M = \tilde{L}\tilde{U}$, with $\tilde{L}$ and $\tilde{U}$ denoting the approximate, or incomplete, factors of A, respectively. The primary distinction between variants of incomplete factorization techniques concerns the criteria upon which fill entries are dropped from the factors.

For symmetric matrices, incomplete Cholesky factorization or IC is used, but this chapter will focus on incomplete LU factorization, ILU, as the matrices of both Amphi3D and the Stokes flow test case at hand are nonsymmetric. A family of ILU preconditioners known as ILU(k), was introduced in [64, 128] and allows fill-in up to level k. A thorough exposition on fill levels is available in [13]. Another popular ILU variant is ILU(τ), which simply drops any fill entries whose absolute value is smaller than the threshold τ. The difficulty of predicting the amount of fill and associated memory requirement is a drawback of both ILU(l) and ILU(τ), and motivated the development of the dual threshold ILUT(τ,p) algorithm [101], which employs a drop tolerance like ILU(τ) but adds the requirement that only p fill-in entries are kept in each row of the incomplete LU factors.

In principle, increasing fill-in improves the convergence rate of the Krylov solver, but there have been observations [49] of non-monotonic improvement of the convergence with increasing fill-in in the preconditioner.

With the introduction of parallel computation, the subject of parallel preconditioning immediately arose, as the prevalent ILU techniques are not readily suited for such application by the inherent sequential nature of Gaussian elimination. Furthermore, the forward and backward solves that have to be conducted at each iteration are highly sequential.

On many occasions [103, 109, 96, 126, 47, 10, 34, 78], the parallel algorithm simply deletes any coupling, that is off-diagonal entries, in the preconditioner matrix. Though this is a simple and highly parallel approach, the quality of the

preconditioners deteriorates rapidly as the number of processors increases.

No-fill ILU can also be parallelized to fair extent through the application of graph coloring techniques, but it too is generally insufficient to achieve a satisfactory rate of convergence in the Krylov subspace solver. Better convergence can be achieved by allowing some fill-in, but this drastically increases cross-process communication.

Another approach to the parallelization of ILU preconditioning is to embed the factorization in domain decomposition. The Additive Schwarz method [113], or ASM, constructs a preconditioner by dividing the problem domain into a number of possibly overlapping subdomains and subsequently combining the subdomain preconditioners. The communication between the subdomains through the overlap can be realized in any of a number of ways [50], or the overlap can be set to zero; reducing the preconditioner to Block Jacobi. Note that although incomplete decomposition techniques are prevalent, Additive Schwarz methods can use any preconditioner for its subdomains. Regardless of the internal preconditioner, however, the convergence of the iterative solver is often poor as a consequence of the relatively crude coupling of the subdomains. One approach to reinforce global coupling is the addition of an extra layer on top of Additive Schwarz in the form of a coarse-grid correction, resulting in a multilevel method [113].

Techniques where reordering is applied with the goal of assigning numbers that are not too far apart to neighbouring grid points or unknowns were first introduced in [52, 121]. Recent versions of this idea can be found in [110, 1, 90, 67, 68] and have been shown to be quite efficient for select problems. The algorithm suggested in [68] is scalable, which means that the work involved in constructing the preconditioner remains constant when the problem size and number of processors are increased with equal factors. The algorithm divides the factorization problem into a number of non-overlapping subproblems of roughly equal size which can be solved in parallel. Each subproblem is divided into edge elements and interior elements; first, only the interior of each subdomain is factored, thereafter rows corresponding to boundary elements are factored. The latter step requires cross-process communication, the limitation of which depends strongly on the possibility of a clean cut, or relatively small total edge length, between the subdomains.

Efficient parallel versions of ILU preconditioning have been achieved, but require considerable sophistication and adaptation, and more importantly, have a narrow area of application with a heavy bias towards 2D problems with regular grids. These issues as well as the inherent instability, due to possible breakdown of the incomplete factorization, have motivated the development of a completely different approach to preconditioning through sparse approximate inverses.

3.2 Sparse approximate inverses

The central idea of sparse approximate inverse preconditioning is that the preconditioner is an explicitly computed approximation to the inverse of the system matrix A; $M^{-1} \approx A^{-1}$. Importantly, the construction of such a preconditioner is fully parallel when the approximation is executed on suitable criteria. Furthermore, there is no inherent risk of breakdown as for ILU and the approach is very robust. Even though the inverse of a sparse matrix is generally dense, and it can even be proven that irreducible sparse matrices have structurally dense inverses [51], many entries of the inverse of a sparse matrix tend to be small in absolute value; creating the opportunity for a close sparse approximation.

Two basic types of approximate sparse inverse preconditioning exist; one where M is a single matrix and one where it is the product of any number of matrices. The latter approach is commonly referred to as factored sparse approximate inverses and is related to the LU factorization of A:

$$M = M_L M_U, \qquad \text{where } M_L \approx L \text{ and } M_U \approx U \tag{3.4}$$

assuming A has an LU decomposition. For both the factored and unfactored cases, two manners of approximation have emerged: Frobenius norm minimization and incomplete (bi-)conjugation.

3.2.1 Frobenius norm minimization

The oldest form of sparse approximate inverses dates back to the seventies [11, 58, 12] and employs the Frobenius norm to compute an approximate inverse M^{-1} [62]:

$$||AM^{-1} - I||_F^2 = \sum_{k=1}^{n} ||(AM^{-1} - I)e_k||_2^2 \tag{3.5}$$

Crucially, minimizing in the Frobenius norm is completely parallel as it can be split into

$$\min_{(M^{-1})_k} ||A(M^{-1})_k - e_k||_2, \qquad k = 1, ..., n \tag{3.6}$$

Each of these minimizations reduces to a small least squares problem once an initial sparsity structure for M^{-1} has been predetermined. The difficulty, then, lies in the algorithm generating a sparsity structure for M^{-1}. This sparsity structure must capture the larger elements in the unknown inverse of A, but must also be constructed at acceptable cost. Whereas earlier papers [81, 75, 76] on Frobenius minimized sparse approximate inverses used fixed sparsity patterns, the algorithm of [62] assumes a very basic initial sparsity structure with which a first version of M^{-1} is constructed and subsequently augments it by adding precisely those

elements which contribute most to the minimization (3.6), up to the point where a maximum number of nonzeros for the row has been reached or the row is close enough to the corresponding row in the inverse of A to meet a certain tolerance. Although all the algorithms within the general framework discussed in this section are sparse approximate inverse methods, the abbreviation SPAI is associated with [62] specifically. This algorithm has proven to be the most successful of its kind and can result in very good convergence of the Krylov solver, but parallelization is far from trivial, as illustrated by the parallel implementations [9, 8], and construction of the preconditioner is very expensive [20].

The latter issue formed the motivation for the introduction of the Minimal Residual method [42], which replaces the exact minimization of (3.6) by a few iterations of a minimal residual method. An added benefit of this approach is that, contrary to SPAI, there is no prescribed initial sparsity pattern required. On the other hand, this technique was observed to be less robust [20]. A yet different class of algorithms has been developed where an effective sparsity pattern is automatically generated, the most notable of which is ParaSAILS, the subject of Section (4.3.3).

A factored, Frobenius-norm minimization approximate inverse preconditioner, FSAI or Factored sparse approximate inverse, was introduced in [75]. It has performed very well forsymetric positive definite (SPD) matrices [14, 76]; retaining the SPD property for the preconditioner of such matrices and therefore enabling the use of the conjugate gradient Krylov method. The preconditioner, $M = M_L M_L^T$, is conceptually constructed by a sparse approximation to the Cholesky factor of the system matrix; however, it is computed using only the values A and the Cholesky factor is never computed. Details involved with the parallel implementation of FSAI can be found in [23, 41, 46, 55]. Notwithstanding the symmetric roots of FSAI, extension to general sparse matrices has been realized. However, its application to nonsymmetric matrices has proven both unreliable and ineffective [15], with AINV (Approximate inverse), which will be described shortly, consistently achieving superior efficiency.

3.2.2 Biconjugation

Biconjugation is a generalization of the Gram-Schmidt process which allows the computation of a triangular factorization of the inverse of A, rather than A itself, using information from A exclusively. The sparse approximation of such a decomposition lies at the heart of AINV [17, 19], an efficient algorithm for factored sparse approximate inverses. The method does, however, share the susceptibility to breakdown [18] of ILU preconditioners. In reaction, an altered algorithm including an evasion of this issue, known as stabilized AINV or SAINV, was developed

independently by [14] and [72].

Although the application of the preconditioner is easily parallelized, its construction is, not unlike ILU, inherently sequential. The rather crude solution of constructing the preconditioner in a sequential manner and subsequently broadcasting it to all other processes as in [22] is simple, but time-consuming and possible only when the problem size is sufficiently small for this to approach to be viable in terms of its memory requirement. Parallel construction of this preconditioner has been achieved through graph partioning [16, 21] and compared favorably to a number of multigrid methods [13, 114]. Further developments of AINV have been presented in [29, 30, 31, 22, 23, 26, 86, 87, 88, 60, 97].

3.3 Algebraic multigrid

Algebraic multigrid methods, or AMG [99], are best understood as a generalization of geometric multigrid methods, which in turn are an extension of geometric two-grid methods. The latter is based on the introduction of a coarse grid in addition to the grid with the resolution of the desired solution. The solution strategy then consists of alternating reduction of the error on the fine grid, known as smoothing, and reduction of the error on the reduced grid, known as coarse grid correction. In algebraic multigrid, the coarse and fine grid distinction is no longer based on the physical domain of the problem, but extracted from the system matrix. As such, there is no direct link between the algebraic multigrid approach and Krylov subspace methods or preconditioners. However, the two approaches, though conceptually and historically distinct, have been combined in various froms; including the use of multigrid solvers within block preconditioners for Krylov solvers [127] and the application of preconditioners designed for Krylov solvers as smoothers for multigrid. In the latter category, incomplete factorization has been applied [129], but recent work in this area has focused on the application of sparse approximate inverses in this manner [33, 32, 117, 116]. Conversely, the difficulty of scalability of preconditioners for Krylov solvers has motivated the quest for variants which include elements of the multigrid approach. Although no particular algorithm has surfaced as the fastest or most robust, a lot of work has been done on multigrid variants of incomplete factorization preconditioners [100, 107, 108, 98, 5, 7, 6, 28, 48, 80, 94, 95, 105, 120, 134] and sparse approximate inverse preconditioners [25, 87, 88, 93].

Although parallelization of algebraic coarsening has proven troublesome, parallel AMG is a young [43] but active field of research [77, 63, 65, 89, 130]. However, little has been achieved in their application to indefinite matrices, or systems arising from three dimensional partial differential equations.

One multigrid method, Finite Element method of Tearing and Interconnecting

or FETI method has been extensively developed in parallel and enjoys considerable popularity and widespread application. However, it is reliant on information of the underlying application and is therefore not a purely algebraic method.

Chapter 4

Stokes flow test case

As alluded to in the introduction and illustrated in the previous chapter, the options for parallel preconditioning of a complex system of equations in algorithms like Amphi3D are plenty. However, finding an effective preconditioner for Amphi3D has proven very difficult. Rather than through trial and error, this project approaches the situation by comparing parallel preconditioners on the much reduced problem of Stokes flow, anticipating that evaluation of preconditioner performance for this problem will provide a solid basis for a more structured, educated and hopefully more fruitful process of preconditioner selection. Instead of writing a new code, a third party Matlab code was selected to serve as the framework for the simulations and generate linear systems to serve as test cases for parallel preconditioners. This code, `NS3D_FEM` (Navier-Stokes 3D, finite element method)[27], does, however, use a different formulation of the problem. This chapter opens with a discussion of that algorithm, followed by a discussion of the details of the linear systems as well as the parallel preconditioners tested on these systems.

4.1 Description of algorithm

4.1.1 The Governing Equations and Boundary Conditions

The problem at hand is steady-state and does not include gravity, but is technically a low Reynolds number Navier-Stokes flow rather than a pure Stokes flow. The starting point is comprised of the usual steady, incompressible Navier-Stokes equations with zero body force:

$$-\nu\nabla^2\boldsymbol{v} + \boldsymbol{v}\cdot\nabla\boldsymbol{v} + \nabla p = 0 \tag{4.1}$$

$$\nabla\cdot u = 0 \tag{4.2}$$

where ν is the kinematic viscosity, $\boldsymbol{v}$ is the velocity vector and p is the pressure. The test case describes flow around a sphere in a cylinder; the domain Ω used in these simulations is a cylinder around the z-axis with a radius of 2 and ranging from $z = -2$ to $z = 7$. The sphere is situated at the origin and has a radius of $\frac{1}{2}$. The inflow boundary $(\partial\Omega)_{in}$ is situated at $z = -2$, the outflow boundary $(\partial\Omega)_{out}$ at $z = 7$ and the no-slip boundary $(\partial\Omega)_{wall}$ is comprised of the wall of the cylinder $x^2 + y^2 = 4$ and the sphere at the origin, $x^2 + y^2 + z^2 = 1/2$. The boundary conditions come to

$$\boldsymbol{v} = \left[0, 0, 1 - \frac{1}{4}\left(x^2 + y^2\right)\right]^T \qquad \text{on } (\partial\Omega)_{in} \tag{4.3}$$

$$\nu\frac{\partial \boldsymbol{v}}{\partial \boldsymbol{n}} + \boldsymbol{n}p = 0 \qquad \text{on } (\partial\Omega)_{out} \tag{4.4}$$

$$\boldsymbol{v} = 0 \qquad \text{on } (\partial\Omega)_{wall} \tag{4.5}$$

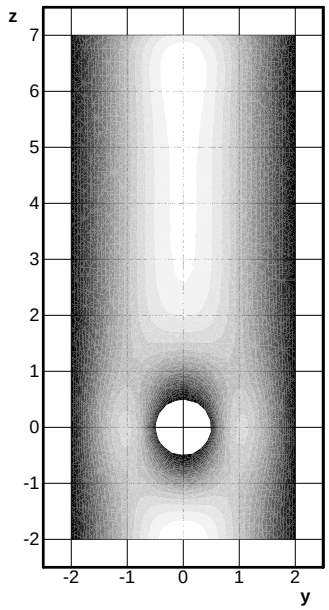

Figure 4.1: A cross section of the domain along $x = 0$. The greyscale indicates velocity in the z-direction, with lighter shades corresponding to higher velocities. $Re = 1$.

4.1.2 Weak Formulation and discretization

Rather than a traditional weak formulation based directly on Equations (4.1) and (4.2), the algorithm of `NS3D_FEM` includes the addition of a penalty term to the continuity equation, restating the system of equations as

$$-\nu\nabla^2\boldsymbol{v} + \boldsymbol{v}\cdot\nabla\boldsymbol{v} + \nabla p = 0 \tag{4.6}$$

$$\nabla\cdot\boldsymbol{v} + \frac{\epsilon}{\nu}p = 0 \tag{4.7}$$

where ϵ is the penalty parameter. Through the derivation of appendix A.2.1, this results in the weak formulation

$$\int_\Omega \nu\nabla\boldsymbol{v} : \nabla\tilde{\boldsymbol{v}} + (\boldsymbol{v}\cdot\nabla\boldsymbol{v})\cdot\tilde{\boldsymbol{v}} - p\nabla\cdot\tilde{\boldsymbol{v}}\, d\Omega = 0, \tag{4.8}$$

$$\int_\Omega (\nabla\cdot\boldsymbol{v} + \frac{\epsilon}{\nu}p)\tilde{p}\, d\Omega = 0. \tag{4.9}$$

The nonlinear term $\nabla\boldsymbol{v}\cdot\tilde{\boldsymbol{v}}$ necessitates the use of a Newton method, as is the case in Amphi3D. This results in a series of iterates $(\boldsymbol{v}_k, p_k), k \in 1, 2, ...$ converging, in principle, to the solution of the weak formulation. The initial guess $(\boldsymbol{v}_0, p_0)$ in the `NS3D_FEM` code satisfies the boundary conditions. Given the iterate $(\boldsymbol{v}_k, p_k)$, the nonlinear residual (R_k, r_k) associated with the momentum part of the weak formulation reads:

$$R_k = \int_\Omega -(\boldsymbol{v}_k\cdot\nabla\boldsymbol{v}_k)\cdot\tilde{\boldsymbol{v}} - \nu\nabla\boldsymbol{v}_k : \nabla\tilde{\boldsymbol{v}} + p_k\nabla\cdot\tilde{\boldsymbol{v}}\, d\Omega \tag{4.10}$$

$$r_k = \int_\Omega -\tilde{p}(\nabla\cdot\boldsymbol{v}_k + \frac{\epsilon}{\nu}p_k)\, d\Omega \tag{4.11}$$

With $\boldsymbol{v} = \boldsymbol{v}_k + \delta\boldsymbol{v}_k$ and $p = p_k + \delta p_k$ the solution of the weak formulation, it follows that

$$D(\boldsymbol{v}_k, \delta\boldsymbol{v}_k, \tilde{\boldsymbol{v}}) + \int_\Omega \nu\nabla\delta\boldsymbol{v}_k : \nabla\tilde{\boldsymbol{v}} - \delta p_k\nabla\cdot\tilde{\boldsymbol{v}}\, d\Omega = R_k(\tilde{\boldsymbol{v}}) \tag{4.12}$$

$$\int_\Omega \tilde{p}(\nabla\cdot\delta\boldsymbol{v}_k + \frac{\epsilon}{\nu}\delta p_k)\, d\Omega = r_k(\tilde{\boldsymbol{v}}) \tag{4.13}$$

where $D(\boldsymbol{v}_k, \delta\boldsymbol{v}_k, \tilde{\boldsymbol{v}})$ is the difference in the nonlinear terms and can be expanded as

$$D(\boldsymbol{v}_k, \delta\boldsymbol{v}_k, \tilde{\boldsymbol{v}}) = \int_\Omega (\delta\boldsymbol{v}_k + \boldsymbol{v}_k)\cdot\nabla(\delta\boldsymbol{v}_k + \boldsymbol{v}_k)\cdot\tilde{\boldsymbol{v}} - (\boldsymbol{v}_k\cdot\nabla\boldsymbol{v}_k)\cdot\tilde{\boldsymbol{v}}\, d\Omega \tag{4.14}$$

$$= \int_\Omega (\delta\boldsymbol{v}_k\cdot\nabla\delta\boldsymbol{v}_k)\cdot\tilde{\boldsymbol{v}} + (\delta\boldsymbol{v}_k\cdot\nabla\boldsymbol{v}_k)\cdot\tilde{\boldsymbol{v}} + (\boldsymbol{v}_k\cdot\nabla\delta\boldsymbol{v}_k)\cdot\tilde{\boldsymbol{v}}\, d\Omega. \tag{4.15}$$

Dropping the quadratic term, (4.15) is plugged back into (4.12), yielding the linear equations

$$\int_\Omega (\delta \boldsymbol{v}_k \cdot \nabla \boldsymbol{v}_k) \cdot \tilde{\boldsymbol{v}} + (\boldsymbol{v}_k \cdot \nabla \delta \boldsymbol{v}_k) \cdot \tilde{\boldsymbol{v}} + \nu \nabla \delta \boldsymbol{v}_k : \nabla \tilde{\boldsymbol{v}} - \delta p_k \nabla \cdot \tilde{\boldsymbol{v}} \, d\Omega = R_k(\tilde{\boldsymbol{v}}), \quad (4.16)$$

$$\int_\Omega \tilde{p}(\nabla \cdot \delta \boldsymbol{v}_k + \frac{\epsilon}{\nu} \delta p_k) \, d\Omega = r_k(\tilde{\boldsymbol{v}}). \quad (4.17)$$

As usual for discretizations of the Navier-Stokes equation, mixed finite elements are used in the spatial discretization, and

$$\boldsymbol{v}_{k+1,h} = \sum_{j=1}^{n_u} \boldsymbol{v}_{k+1}^j \phi_j \quad (4.18)$$

$$p_{k+1,h} = \sum_{j=n_u+1}^{n_u+n_p} p_{k+1}^j \psi_j \quad (4.19)$$

where ϕ_k is the set of vector-valued basis functions for the velocity, and ψ_k the set of pressure basis fuctions. Furthermore n_u holds the number of velocity degrees of freedom, three times the total number of velocity nodes minus those associated with the Dirichlet boundary conditions, and n_p similarly denotes the number of pressure degrees of freedom. Substituting (4.18, 4.19) into (4.16, 4.16) results in the system of linear equations

$$\begin{bmatrix} A & B^T \\ B & C \end{bmatrix} [x] = \begin{bmatrix} f \\ g \end{bmatrix} \quad (4.20)$$

where $x = [\boldsymbol{v}_{k+1}^1, ..., \boldsymbol{v}_{k+1}^{n_u}, p_{k+1}^{n_u+1}, ..., p_{k+1}^{n_u+n_p}]^T$ and

$$A(i,j) = \int_\Omega \{\phi_j \cdot \nabla \boldsymbol{v}_{k,h} + \boldsymbol{v}_{k,h} \cdot \nabla \phi_j\} \cdot \phi_i + \nu \nabla \phi_j : \nabla \phi_i \, d\Omega \quad (4.21)$$

$$B(i,j) = \int_\Omega \psi_i \nabla \cdot \phi_j \, d\Omega \quad (4.22)$$

$$C(i,j) = \int_\Omega \frac{\epsilon}{\nu} \psi_i \psi_j \, d\Omega \quad (4.23)$$

$$f(i) = \int_\Omega -\{\boldsymbol{v}_{k,h} \cdot \nabla \boldsymbol{v}_{k,h}\} \cdot \phi_i - \nu \nabla \boldsymbol{v}_{k,h} : \nabla \phi_i + p_{k,h} \nabla \cdot \phi_i \, d\Omega \quad (4.24)$$

$$g(i) = \int_\Omega (\nabla \cdot \boldsymbol{v}_{k,h} + \frac{\epsilon}{\nu} p_{k,h}) \psi_i \, d\Omega \quad (4.25)$$

$$(4.26)$$

The element matrices and element vectors arising form this system are immediately apparent, as their form is identical to the matrix and right hand side, respectively, of (4.20). In fact, the only difference is that the domain of integration is the element rather than the entire domain, and that the integration is performed by a 5-point Gauss quadrature rule. The use of terahedral Taylor-Hood elements implies that the element matrices are 34×34 and the element vectors have length 34.

Taylor-Hood Elements

The Taylor-Hood element [118] was designed for mixed finite element problems, and the tetrahedral version features 10 nodes for the velocity components, one on each vertex and one in the center of each edge, and 4 pressure nodes, one on each vertex. In three dimensions, then, each element has 34 degrees of freedom in total.

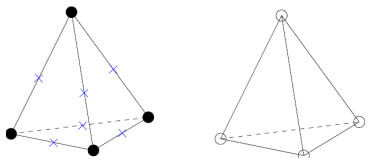

Figure 4.2: The 3D Taylor-Hood element. The dots and crosses in the left tetrahedron represent the vertex and edge velocity nodes respectively; the circles in the right tetrahedron represent the presssure nodes.

4.2 Solution of the linear systems

For the preconditioner tests of this report, the first linear system arising in the Matlab code `NS3D_FEM` is written out as an ascii file and transferred to the Sara Huygens system. Subsequently, a Fortran code including the PETSc libraries is ran which reads the linear system, constructs the preconditioner and executes the Krylov solver. Although this is naturally not a viable option when running actual simulations, it does not affect the timings as the crucial steps, construction of the preconditioner and execution of the Krylov solver, would be identical in an all-Fortran code.

Sara Huygens is an IBM pSeries 575, clustered SMP (Symmetric Multiprocessing) system consisting of 104 nodes, 16 dual core processors (IBM Power6, 4.7 GHz) per node, either 128 GByte or 256 GByte of memory per node. The smaller memory nodes are sufficient for these calculations. Huygens uses simultaneous

multithreading and recommends the usage of 64 MPI tasks per node. Use of this guideline means that jobs using more than 64 MPI tasks use multiple nodes, which implies the addition of slower, cross-nodal communication. Throughout the rest of this report, MPI tasks are referred to as processors for simplicity, although this is not strictly correct.

4.2.1 Properties of the linear systems

Calculations are executed for Reynolds numbers 1, 10 and 50. With the standard grid size, the problem has $4.1 \cdot 10^4$ elements and $5.9 \cdot 10^4$ nodes. By the use of Taylor-Hood elements, this results in $1.6 \cdot 10^5$ velocity and $7.9 \cdot 10^3$ pressure degrees of freedom, a square matrix of size $1.6 \cdot 10^5$ and $1.5 \cdot 10^7$ nonzeros. Grid sizes double and eight times the standard size with correspondingly larger matrix dimensions were also tested.

The matrix is not symmetric, but does have a symmetric nonzero pattern. An important difference between the matrices arising here within `NS3D_FEM` and within Amphi3D is that the diagonal entries corresponding to the pressure degrees of freedom are nonzero as a result of the penalty term (see Section 4.1.2).

4.2.2 Reordering

The first operation performed on the linear systems is Reverse Cuthill-McKee reordering [61], which is aimed at the concentration of all nonzeros around the main diagonal. This is particularly important when solving the system in parallel as it reduces the number of connections between the variables across processors. The implementation of `NS3D_FEM` inherently produces a matrix where the velocity components are already ordered in such a fashion, whereas the pressure nodes are segregated and unordered. The primary effect of the reordering, therefore, is the placement of the pressure degrees of freedom among the velocity degrees of freedom of the same nodes, as illustrated in Figure 4.3.

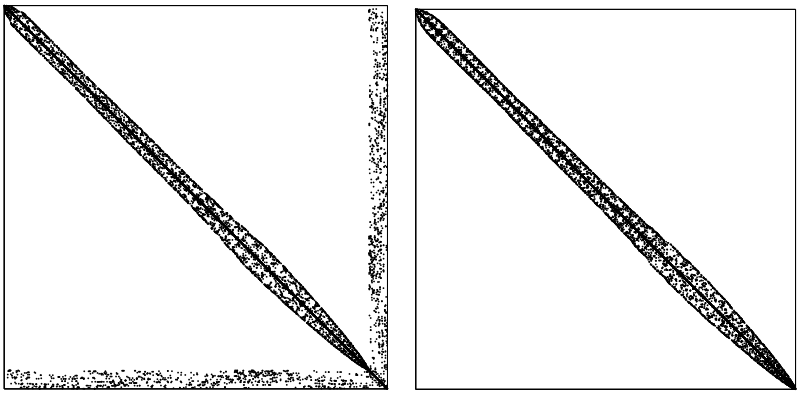

Figure 4.3: The sparsity structure of coefficient matrix before and after Reverse Cuthill-McKee reordering.

4.2.3 Symmetric diagonal scaling

After reordering of the linear system, symmetric diagonal scaling is applied to the linear system $Ax = b$:

$$D^{-1/2} A D^{-1/2} \tilde{x} = D^{-1/2} b \tag{4.27}$$

where $\tilde{x} = D^{1/2} x$, and $D^{1/2}$ and $D^{-1/2}$ are diagonal matrices with entries $D^{1/2}(i,i) = |A(i,i)|$ and $D^{-1/2}(i,i) = |A(i,i)|^{-1}$. It is this scaled linear system that is passed on to the preconditioners and BiCGStab(ℓ).

4.2.4 Use of BiCGStab(ℓ)

In preliminary runs, BiCGStab(ℓ) [111, 112], $\ell = 2$ surfaced as the fastest Krylov iterative solver for these matrices, and all further tests were conducted with this solver.

As the PETSc implementation of BiCGStab(ℓ) only allows left preconditioning, all preconditioners tested are applied from the left. The iterations are said to have converged when the relative residual reaches below $1 \cdot 10^{-6}$; in other words, when $||r||_2 < 1 \cdot 10^{-6}||b||_2$, where r and b denote the residual and the right hand side of the linear system, respectively.

4.3 Parallel Preconditioners

As the symmetric diagonal scaling described in Section 4.2.3 is applied to the system matrices before all solves, they are reasonably conditioned even before a preconditioner is applied. This makes the solution of the system without further preconditioning feasible, and causes its inclusion in the comparisons to be more than a trivial point of reference. Apart from the unpreconditioned system, three preconditioners available through PETSc are tested. Block jacobi is included as it is the simplest of parallel preconditioners. PILUT has been included to represent the incomplete factorization class of preconditioners. This is a legacy code and no longer supported, but is the only preconditioner of this class available through PETSc outside Euclid, which was the first choice but foregone when it consistently crashed during preconditioner construction. PETSc offers access to two parallel approximate inverse preconditioners, SPAI and ParaSAILS (parallel sparse approximate inverse, least squares). The latter was selected as it is both the most efficient, according to a comparison in [40], and the most stable of the two.

4.3.1 Block Jacobi

As mentioned in Section 3.1, Block Jacobi preconditioning simply separates the linear system into several subsystems between which no dependency exists and which are subsequently preconditioned by a local preconditioner. This is enforced by reducing the system matrix to a block diagonal matrix. The most common choice, also used in the calculations of this report, for the number of blocks is one per processor, motivated by the relatively low cost of preconditioning the local part of the matrix. Both inexact and exact factorization are popular methods to precondition the blocks, but exact factorization has been selected here for its simplicity and lack of further parameters. An interesting side effect of this choice is that when running on a single processor, the system is solved exactly.

4.3.2 PILUT

PILUT, introduced in [71], is a parallelization of the ILUT(m,t) algorithm of [101]. This warrants a consideration of the ILU(m,t) algorithm before the parallel aspects of PILUT are discussed.

The sequential algorithm: ILUT(m,t)

In order to limit the cost of both its construction and application, ILUT(m,t) employs a dual dropping strategy. The general algorithm and the point of application for both dropping stages is illustrated in Figure 4.4.

```
1   for i = 1,...,n
2     w = a(i,*)
3     for k = 1,...,i-1
4       if w(k) != 0
5         w(k) = w(k)/a(k,k)
6         Apply first dropping rule to w(k)
7         if w(k) != 0
8           w = w - w(k)*u(k,*)
9         endif
10      endif
11    endfor
12    Apply second dropping rule to w
13    for j = 1,...,i-1
14      L(i,j) = w(j)
15    endfor
16    for j = 1,...,n
17      U(i,j) = w(j)
18    endfor
19    w = 0
20  endfor
```

Figure 4.4: The ILUT(m,t) algorithm

In this algorithm, `w` is a working row used to accumulate linear combinations of sparse rows in the elimination. The first dropping rule, applied in Line 6 of Figure 4.4, simply sets any element of `w` to zero when it is smaller than the relative tolerance t_i, the product of the parameter t and the 2-norm of the `i`th row of `A`. Line 12 features the second dropping rule, which initially one again drops all values below the relative tolerance t_i, but additionally employs the second parameter m to delete all but the m largest elements of the `L` part of the row as well as all but the

m largest elements of the `U` part of the row. This results in $m+1$ elements per row of both `L` and `U` as the diagonal elements are not subject to any dropping criteria. A discussion of proper data structures for efficient sequential implementation of this algorithm is featured in [101, 102].

The parallel algorithm: PILUT

The central idea of the PILUT algorithm of [71] is to separate those rows of the coefficient matrix which are internal to processor's domain, that is have no connections to nonlocal rows, and those that do have such connections. The former category of rows is branded interior, whereas nodes, or rows, within the latter category are known as interface nodes.

The first step of the algorithm consists of each processor factoring its set of interior nodes, which is an entirely sequential process. Note that this is where the connection between PILUT and ILUT(m,t) resides; all the factorizations discussed in this Section are implemented as ILUT(m,t) factorizations. The values for the paramaters m and t used for the calculations of this report are the HYPRE default values of $m = 20$ and $t = 1 \cdot 10^{-4}$, respectively. In the second step, the interior nodes of all processors are eliminated from A, thus forming the reduced matrix A^I featuring only interface nodes. Factoring A^I is subsequently executed by all processors cooperating, and it is in this phase that the communication resides, as well as the distinction between most parallel ILU algorithms.

PILUT performs the parallel factorization of A^I in phases, where during each phase $l = 0, 1, ...$ the processors cooperate to factor a set S_l of rows of A^I_l ($A^I = A^I_0$), and removing these rows from A^I_l to produce A^I_{l+1}. These cycles come to a halt only when all rows of A^I have been factored. As they are expensive in terms of communication, it is important to note that the number of cycles required to perform the entire factorization depend on the maximum number of nonzeros permitted in each row and the threshold of the incomplete fatorization.

The next issue at hand is the construction of the set S_l. PILUT takes the approach of choosing each S_l as a maximal independent set of the rows of A^I_l, thus ensuring that the nodes it represents are not connected in the matrix A^I_l. The algorithm to derive S_l is a parallel formulation [71] of the Luby's algorithm [83].

Subsequently, A^I_l is permuted so that the rows in S_l are at the top, followed by the factorization of those rows by the ILUT(m,t) algorithm. The reduced matrix for the next level, A^I_{l+1}, is then formed with the algorithm of Figure 4.5, which once again is based on ILUT(m,t). This algorithm is applied after the permutation of A^I_l, and eliminates the first `n(l+1)-n(l)+1` unknowns to produce another square matrix A^I_{l+1} of size `n-n(l+1)+1`. Here, `n(l)` is interpreted as the number rows

already factored out of A_l^I, rather than the `l`th element of a vector `n`.

```
1   for i = n(l+1),...,n
2     w = a(i,*)
3     for k = n(l),...,n(l+1)-1
4       if w(k) != 0
5         w(k) = w(k)/a(k,k)
6         Apply first dropping rule to w(k)
7         if w(k) != 0
8           w = w - w(k)*u(k,*)
9         endif
10      endif
11    endfor
12    w = w + L(i,*)
13    L(i,*) = 0
14    a(i,*) = 0
15    Apply modified dropping rule to w
16    for j = i,...,n(l+1)-1
17      L(i,j) = w(j)
18    endfor
19    for j = n(l+1),...,n
20      a(i,j) = w(j)
21    endfor
22    w = 0
23  endfor
```

Figure 4.5: The algorithm used to form successive reduced matrices.

For each row `i` in A_l^I S_l, the algorithm performs linear combinations in Line 8 of Figure 4.5 only with rows that are in S_l, as illustrated by the range for `k` defined in Line 3. Once the construction of `w` is completed, using a treshold dropping rule in Line 6 exactly like that used in Line 6 of 4.4, it is merged with the `i`th row of `L` in Line 12. Line 15 consists of a modified dropping rule, which features both the threshold and maximum nonzeros per row dropping criteria like the second dropping rule of the ILUT(m,t) algorithm, but applies only those elements of `w` corresponding to nodes which have already been factored, that is whose index is smaller than `n(l+1)`. Line 17 sees the elements of `w` that correspond to the `i`th row of `L` copied back, while those corresponding to the unfactored part of the matix are copied back to the `i`th row of A_l^I, which becomes the `i`th row of A_l^I.

Whereas the construction of S_l inherently requires communication, the ILUT factorization in Figure 4.5 is performed separately, with each processor factoring

the locally stored rows within S_l. As these rows are independent by construction, this merely requires the creation of the rows of `U` for each local row. Subsequently, each processor generates the rows of the next level reduced matrix A^I_{l+1} that correspond to the locally stored nodes with the algorithm of Figure 4.5. In particular, for each row `i`, the processor needs to perform linear combinations with all rows `U(k,*)`, for which `A(i,k)` $\neq 0$ and `n(l)` $\leq$ `k` $<$ `l+1`. Some communication is required here as not all `U(k,*)` are necessarily stored locally. However, the non-local rows required by each processor can be determined and sent before performing the associated computations, as the linear combinations they contribute to do not create any fill elements. After this communication has taken place, each processor essentially executes the algorithm of Figure 4.5, updating `L` and creating A^I_{l+1}. The cycles are terminated when all rows in A^I_{l+1} are independent, as this immediately enables their fully parallel elimination.

4.3.3 ParaSAILS

Also implemented as part of the HYPRE package is ParaSAILS, a parallel sparse approximate inverse preconditioner proposed in [40, 41]. As mentioned in Section (3.2.1), ParaSAILS autmatically generates the sparsity pattern for the preconditioner without the requirement of an externally supplied initial sparsity pattern. The basic approach consists taking the sparsity pattern of a positive integer power of A, an idea motivated by the Neumann series expansion of the inverse of A. High powers generally result in good converge, but can be quite expensive to compute. However, ParaSAILS is more sophisticated as the technique consists of reducing the sparsity pattern of A and subsequently taking a power of the resulting sparsity pattern. This approach tends to work best when the square of the reduced sparsity pattern of A is used [40], and this is the default for the HYPRE implementation of ParaSAILS and used for the calculations of this report.

Recall from Section (3.2.1) that ParaSAILS belongs to the class of sparse approximate inverse preconditioners which is based on Frobenius norm minimization. However, the algorithm attempts to construct the preconditioner by seeking M^{-1} such that $M^{-1}A = I$ rather than $AM^{-1} = I$, as in (3.2.1). This is a minor difference, but does result in an altered algorithm, seeking the preconditioner M^{-1} through the minimization of

$$||M^{-1}A - I||_F^2 = \sum_{k=1}^{n} ||{m_k}^T A - {e_k}^T||_2^2 \tag{4.28}$$

where e_k and ${m_k}^T$ are the kth rows of the identity matrix and M^{-1}, respectively. This is easily rearranged and split into n independent minimizations,

$$\min_{m_k} ||{m_k}^T A - {e_k}^T||_2, \qquad k = 1, ..., n \tag{4.29}$$

Which yields the ideal preconditioner $M^{-1} = A^{-1}$ when no additional constraints are placed upon M^{-1}. This would however be extremely costly, which is remedied by the constraint that all ${m_k}^T$ are sparse.

Rather than taking the adaptive approach, where a very sparse initial structure is assumed and augmented until a satisfactory preconditioner has been found, thus going through various iterations of calculating both the sparsity pattern and values of the nonzeros of M^{-1}, the ParaSAILS algorithm constructs a sparsity pattern directly, thus decoupling the calculation of the structure of the preconditioner and its nonzero values. This was motivated by the very high cost of adaptive preconditioner construction, and [40] contains a comparing study of ParaSAILS and SPAI, the most common adaptive algorithm, which indicates that ParaSAILS produces very similar preconditioner quality when compared to SPAI, but is much cheaper to construct.

The general idea is that the sparsity pattern for the preconditioner is the pattern of a power L of the sparsified system matrix, as previously considered in [2, 117, 56, 40], that is the pattern of $\tilde{A}^L$. Sparsification is achieved through dropping all entries of the system matrix whose absolute value lies below a threshold. In the calculations of this report, the default threshold value of 0.1 was used. Although the optimal value for the threshold is in general very problem-dependent, much of this dependency is eliminated by the application of symmetric diagonal scaling. Whereas the threshold effectively selects the most important direct interactions represented by the matrix, taking a power L of $\tilde{A}^L$ ensures that "nearby" interactions are included. For the comparative tests of the report, $L = 2$ was used as this is both recommended in [40, 41] and the default for the HYPRE implementation.

The algorithm can therefore be summarized as follows [41]:

- Threshold A to produce $\tilde{A}$
- Compute the pattern $\tilde{A}^L$ for M^{-1}
- Compute the nonzero entries in M through minimization of $||M^{-1}A - I||_F^2$

An additional post-processing step filtering small values out of the preconditioner is available, intended to speed up the application of the preconditioner in the Krylov iterations. This step, however, is not part of the HYPRE implementation of ParaSAILS by default and was therefore not used for the calculations.

Concerning the thresholding procedure, it should be noted that thresholding is best conducted only after symmetric diagonal scaling has been applied, as is the case in the code used here (see Section(4.2)). At the very least, care must be taken to ensure that the diagonal entries are retained in the sparsity pattern even when their absolute value lies below the threshold.

The subsequent construction of the pattern $\tilde{A}^L$ is conceptually straightforward, but inherently a parallel process and responsible for the bulk of communication required in the set-up of the ParaSAILS preconditioner. Finally, the computation of the nonzero values of M^{-1}, is split such that each processor k solves the least squares problem

$$\min_{m_k} ||{m_k}^T A - {e_k}^T||_2, \qquad k \in 1, ..., n \tag{4.30}$$

by QR factorization. As ${m_k}^T$ is sparse and tends to have its nonzeros concentrated around the kth entry, this phase of the algorithm requires relatively little communication between processors.

Chapter 5

Software issues

5.1 Reordering and scaling

The reordering and scaling, as described in Sections 4.2.2 and 4.2.3 respectively, are both executed within Matlab for the standard grid size. The reordering is performed directly with the Matlab function `symrcm`. However, a laptop with 2 GB of memory was used for Matlab calculations, and the larger matrices which arise from grids double or 8 times the standard size take too much memory to be held in Matlab workspace, despite their sparse format. The assembly of the unpermuted, unscaled matrices and writing them to file is still possible by assembling the matrix in partitions, saving a partition to a file and clearing it from memory before proceeding to the next. However, the unavailability of the entire matrix does complicate its permutation and scaling.

For each matrix arising from either of the larger grid sizes, the permutation vector is generated with a modified version of the code. As assembly of the system matrix is impossible, a logical matrix is assembled instead, which reduces its memory requirement to a fraction of that of the original. This matrix does not contain the nonzero values, but does represent the sparsity structure of the matrix. As the `symrcm` function depends only on the matrix structure, the permutation for the actual system matrix can be obtained by calling `symrcm` upon the logical matrix. The resulting permutation vector is written to a separate file.

A separate Fortran code was written to read the unpermuted and unscaled matrices, as well as the corresponding right hand sides and permutation vectors from file. This code applies the permutation and symmetric diagonal scaling to the matrix and right hand side and subsequently writes the reordered and scaled matrix and vector out to file. The resulting files are equivalent to the files that are written directly from Matlab for the standard grid size.

5.2 Larger grid sizes

Although the techniques of the last section enable preconditioner testing on the larger linear systems associated with the increased grid sizes, the solution of these linear systems with PETSc proved to be very troublesome. Although the memory requirement is no issue on the Sara Huygens system as described in Section 4.2, the attempted solution of these systems consistently resulted in various errors, with or without preconditioning.

The memory requirement prevents a validation of these systems by solving them within Matlab, but checks were performed to verify, for example, that the matrices and right hand sides have a finite 1-norm, that the matrix has no zeros on the diagonal and that its structure is qualitatively equal to that of the smaller systems which are reordered and scaled within Matlab and can be solved with PETSc.

However, problems were encountered with all preconditioners when attempting to solve these larger systems. The issue arising for the unpreconditioned case is as follows. The Fortran code executing the solve with PETSc first instructs PETSc to solve the linear system with BiCGS(L), and subsequently calls another PETSc function to request a flag representing the reason BiCGS(L) has stopped iterating. The regular values for this flag represent either convergence or divergence for various reasons, but the flag returned in this case states BiCGS(L) is still iterating. At this point there is no indication as to the cause of this problem.

Use of the PILUT preconditioner consistently led to a segmentation violation during preconditioner construction, exactly like it did for the smaller grid size cases on many occasions, as illustrated in Tables 6.1 to 6.3. Furthermore, Block Jacobi resulted in a different error during preconditioner construction (`exceeded maximum number of blocks`), and ParaSAILS construction simply never completed, or at least not within a few hours. These issues were encountered across the range of both Reynolds numbers and number of processors. Their cause is unclear at this point, but the problem for the unpreconditioned case seems to indicate that the problem might lie outside of the individual preconditioners, even if the error messages they present are different.

5.3 HYPRE

Segmentation violations are encountered in PILUT both for quite a few of the smaller grid size cases and for all of the larger grid size cases. The reason that the legacy implementation PILUT is, is that segmentation violations occurred in all cases even for the smaller grid sizes when using Euclid, a new parallel ILU implementation which is now recommended over PILUT in the HYPRE manual

[54]. ParaSAILS, also part of HYPRE, in contrast is quite robust for the smaller grid size problems, encountering a problem in just one of the test cases, where the preconditioner construction never ended, as in all the larger grid size cases. Furthermore, ParaSAILS only reacted occasionally to non-default parameter values; identical calls with non-default parameters were often incorrectly executed with default parameters on one occasion, and after logging out and back in would result in correct use of the custom parameters. This, however, did not affect the results presented in Chapter 6 as default parameters were used.

Chapter 6

Results

All results were obtained using BiCGStab(ℓ), $L = 2$, and left preconditioning. The results for linear systems arising from the smallest grid size are collected in Tables 6.1 to 6.3, which lists the performance of each preconditioning method for 1 to 1024 processors. As described in Section 5.2, various issues prevented meaningful results for the larger grid sizes. The tables list the number of iterations of BiCGStab(ℓ) required upon application of the preconditioner, the time required for the construction of the preconditioner, the time spent in iterations of BiCGStab(ℓ) and the total time, summing the two.

The total time for solution without the use of any preconditioner are surprisingly good when compared to the preconditioned solution times; in fact, the fastest total solution time is achieved without preconditioning and using 256 processors for each of the 3 Reynolds numbers. Note, however, that symmetric diagonal scaling (Section 4.2.3) is still applied in the unpreconditioned case, and this does constitute a very basic form of preconditioning. The truly unpreconditioned case, without diagonal scaling, would likely result in far inferior performance.

Although not the case in the `NS3D_FEM` code, it is important to keep in mind that in many applications, including Apmhi3D, multiple solves are performed for each matrix but with different right hand sides. This has a strong influence on the choice of preconditioner, as the preconditioner only needs to be constructed once for such a string of solves. Consequently, these applications will favor more expensive, better preconditioners as the Krylov solve time becomes much more important than the preconditioner construction time. The advantage of solving the unpreconditioned system interestingly dissolves for these applications, as both PILUT and ParaSAILS yield Krylov solve times very similar to the unpreconditioned case for $Re = 1$, PILUT is faster in this respect for $Re = 50$ and ParaSAILS is faster for both $Re = 10$ and $Re = 50$.

These observations, however, consider only the optimal number of processors

for each case. The scalability of the preconditioners is an entirely different issue, and likely has a greater influence on their ultimate potential than absolute performance for these test cases. Starting with the sensitivity of preconditioner setup time to the number of processors, Block Jacobi gets cheaper to construct as the number of processors increases. This is a d1irect consequence of the fact that its construction is entirely parallel, and consists of the exact solution of a linear system the size of which is inversely proportional to the number of processors used(see Section 4.3.1). With that in mind however, the speed-up in this respect is actually disappointing, as best illustrated in the case for $Re = 50$. Illustrative of the nature of parallel ILU preconditioners, the construction cost for PILUT is hardly dependent on the number of processors used. ParaSAILS reacts much better to the addition of processors, showing significantly decreasing setup times even though the dependence is rather erratic.

The Krylov solve times for both the unpreconditioned case and ParaSAILS are shortest for 256 processors. The lack of benefit from the jump to 1024 is likely to be partially caused by the significant increase in communication between nodes this requires (see Section 4.2), and it is therefore not unlikely that the addition of yet more processors would once again speed up the Krylov solves in these cases. The scalability of PILUT is disappointing in terms of the Krylov solve time as well, with the fastest solves achieved using just 16 processors. Block Jacobi, too, is unsurprisingly poor in this respect as it is a very poor preconditioner on a larger number of processors. This is illustrated in particular by the iteration counts for the cases $Re = 1$ and $Re = 50$ on 16 and 64.

ParaSAILS is quite striking in terms of the iteration counts it yields, as these are roughly constant across the board. Whereas Block Jacobi is very good in this respect when using very few processors and horrible when using many, PILUT is fine up to 16 and drops off very dramatically at 64. The unpreconditioned case is independent of the number of processors but suffers for the higher Reynolds numbers. The steady ParaSAILS iteration count is poor for $Re = 1$, but quite good for $Re = 50$. Iterations counts for PILUT seem completely unpredictable in contrast, particularly for $Re = 1$, where good iteration counts are achieved with 1 or 16 processors, but the use 4 results in divergence of BiCGS(L).

Finally, note that the results for Block Jacobi are as expected, with preconditioner setup times independent of the Reynolds number as, the reduction to slow, direct solution of the linear system by LU decomposition when running with one processor and very poor iteration counts when using higher numbers of processors.

Preconditioner	Proc.	Iter.	Setup time	Krylov time	Total time
None	1	814	–	$9.00 \cdot 10^1$	$9.00 \cdot 10^1$
	4	840	–	$2.30 \cdot 10^1$	$2.30 \cdot 10^1$
	16	844	–	$3.84 \cdot 10^1$	$3.84 \cdot 10^1$
	64	808	–	$3.05 \cdot 10^0$	$3.05 \cdot 10^0$
	256	860	–	$2.94 \cdot 10^0$	$2.94 \cdot 10^0$
	1024	836	–	$3.93 \cdot 10^0$	$3.93 \cdot 10^0$
Block Jacobi	1	*err*	–	–	–
	4	34	$1.52 \cdot 10^2$	$1.67 \cdot 10^2$	$3.19 \cdot 10^2$
	16	80	$3.48 \cdot 10^0$	$7.98 \cdot 10^0$	$1.15 \cdot 10^1$
	64	1134	$1.74 \cdot 10^0$	$1.01 \cdot 10^1$	$1.18 \cdot 10^1$
	256	*div*	–	–	–
	1024	*div*	–	–	–
PILUT	1	192	$7.82 \cdot 10^0$	$3.27 \cdot 10^1$	$4.05 \cdot 10^1$
	4	*div*	–	–	–
	16	154	$2.96 \cdot 10^0$	$2.96 \cdot 10^0$	$5.39 \cdot 10^0$
	64	*div*	–	–	–
	256	*err*	–	–	–
	1024	*err*	–	–	–
ParaSAILS	1	*err*	–	–	–
	4	686	$1.95 \cdot 10^2$	$2.64 \cdot 10^1$	$2.21 \cdot 10^2$
	16	674	$5.05 \cdot 10^1$	$4.31 \cdot 10^0$	$5.48 \cdot 10^1$
	64	788	$2.10 \cdot 10^1$	$3.79 \cdot 10^0$	$2.48 \cdot 10^1$
	256	742	$7.45 \cdot 10^0$	$3.17 \cdot 10^0$	$1.06 \cdot 10^1$
	1024	712	$4.00 \cdot 10^0$	$4.67 \cdot 10^0$	$8.67 \cdot 10^0$

Table 6.1: Preconditioner performance for $Re = 1$, on a grid of $4.1 \cdot 10^4$ elements and $5.9 \cdot 10^4$ nodes. The coefficient matrix is a square matrix of size $1.6 \cdot 10^5$ and contains $1.5 \cdot 10^7$ nonzeros.

Preconditioner	Proc.	Iter.	Setup time	Krylov time	Total time
None	1	1330	–	$1.46 \cdot 10^2$	$1.46 \cdot 10^2$
	4	1356	–	$3.70 \cdot 10^1$	$3.70 \cdot 10^1$
	16	1304	–	$5.95 \cdot 10^0$	$5.95 \cdot 10^0$
	64	1332	–	$4.79 \cdot 10^0$	$4.79 \cdot 10^0$
	256	1330	–	$3.43 \cdot 10^0$	$3.43 \cdot 10^0$
	1024	1360	–	$1.02 \cdot 10^1$	$1.02 \cdot 10^1$
Block Jacobi	1	*err*	–	–	–
	4	38	$1.52 \cdot 10^2$	$1.68 \cdot 10^2$	$3.20 \cdot 10^2$
	16	104	$3.55 \cdot 10^0$	$9.30 \cdot 10^0$	$1.29 \cdot 10^1$
	64	*div*	–	–	–
	256	*div*	–	–	–
	1024	*div*	–	–	–
PILUT	1	*div*	–	–	–
	4	*div*	–	–	–
	16	528	$3.27 \cdot 10^0$	$8.26 \cdot 10^0$	$1.15 \cdot 10^1$
	64	7874	$5.03 \cdot 10^0$	$2.01 \cdot 10^2$	$2.06 \cdot 10^2$
	256	*err*	–	–	–
	1024	*err*	–	–	–
ParaSAILS	1	778	$6.39 \cdot 10^2$	$1.04 \cdot 10^2$	$7.43 \cdot 10^2$
	4	838	$1.95 \cdot 10^2$	$3.16 \cdot 10^1$	$2.27 \cdot 10^2$
	16	850	$5.13 \cdot 10^1$	$5.28 \cdot 10^0$	$5.66 \cdot 10^1$
	64	940	$8.70 \cdot 10^1$	$4.20 \cdot 10^0$	$9.12 \cdot 10^1$
	256	832	$2.82 \cdot 10^1$	$2.75 \cdot 10^0$	$3.10 \cdot 10^1$
	1024	920	$9.54 \cdot 10^0$	$5.52 \cdot 10^0$	$1.51 \cdot 10^1$

Table 6.2: Preconditioner performance for $Re = 10$, on a grid of $4.1 \cdot 10^4$ elements and $5.9 \cdot 10^4$ nodes. The coefficient matrix is a square matrix of size $1.6 \cdot 10^5$ and contains $1.5 \cdot 10^7$ nonzeros.

Preconditioner	Proc.	Iter.	Setup time	Krylov time	Total time
None	1	1334	–	$1.47 \cdot 10^2$	$1.47 \cdot 10^2$
	4	1322	–	$3.56 \cdot 10^1$	$3.56 \cdot 10^1$
	16	1314	–	$6.04 \cdot 10^0$	$6.04 \cdot 10^0$
	64	1296	–	$4.70 \cdot 10^0$	$4.70 \cdot 10^0$
	256	1446	–	$3.48 \cdot 10^0$	$3.48 \cdot 10^0$
	1024	1342	–	$5.64 \cdot 10^0$	$5.64 \cdot 10^0$
Block Jacobi	1	–	$2.48 \cdot 10^3$	–	$2.48 \cdot 10^3$
	4	38	$1.52 \cdot 10^2$	$1.68 \cdot 10^2$	$3.20 \cdot 10^2$
	16	94	$3.53 \cdot 10^0$	$8.76 \cdot 10^0$	$1.23 \cdot 10^1$
	64	1428	$2.15 \cdot 10^0$	$1.21 \cdot 10^1$	$1.43 \cdot 10^1$
	256	*div*	–	–	–
	1024	*div*	–	–	–
PILUT	1	404	$7.85 \cdot 10^0$	$6.81 \cdot 10^1$	$7.60 \cdot 10^1$
	4	144	$3.90 \cdot 10^0$	$6.53 \cdot 10^1$	$1.04 \cdot 10^1$
	16	106	$3.00 \cdot 10^0$	$1.69 \cdot 10^0$	$4.69 \cdot 10^0$
	64	8292	$4.49 \cdot 10^0$	$2.12 \cdot 10^2$	$2.16 \cdot 10^2$
	256	*err*	–	–	–
	1024	*err*	–	–	–
ParaSAILS	1	754	$6.37 \cdot 10^2$	$1.01 \cdot 10^2$	$7.38 \cdot 10^2$
	4	772	$1.95 \cdot 10^1$	$2.96 \cdot 10^1$	$2.25 \cdot 10^2$
	16	730	$5.04 \cdot 10^0$	$4.60 \cdot 10^0$	$5.50 \cdot 10^1$
	64	744	$2.12 \cdot 10^0$	$3.66 \cdot 10^0$	$2.49 \cdot 10^1$
	256	770	$7.56 \cdot 10^0$	$3.23 \cdot 10^0$	$1.08 \cdot 10^1$
	1024	844	$4.05 \cdot 10^0$	$5.09 \cdot 10^0$	$9.14 \cdot 10^0$

Table 6.3: Preconditioner performance for $Re = 50$, on a grid of $4.1 \cdot 10^4$ elements and $5.9 \cdot 10^4$ nodes. The coefficient matrix is a square matrix of size $1.6 \cdot 10^5$ and contains $1.5 \cdot 10^7$ nonzeros.

Chapter 7

Conclusions

- There is a lot of room for improvement both in terms of the performance and the stability of parallel preconditioners for Stokes flow. PETSc itself, too, has proven problematic.
- Software issues in general are a major impedience, particularly because of the lack of a clear root for many of the errors encountered
- The limitations of the Block Jacobi approach have manifested very clearly in the divergence of all tests with this rather primitive form of parallel preconditioning using 256 or more processors.
- PILUT performs slightly better in this respect, but its erratic performance and lack of stability complicate the review of its performance. This and the legacy status of the implementation used implies that further research should still consider parallel ILU.
- Although the performance achieved without preconditioning is remarkable for the tests conducted, ParaSAILS arises as the most promising of the preconditioners tested here. It is the fastest method for the two larger Reynolds number cases tested in terms of the time required for the iterative solution of the preconditioned system it yields, and does so while the cost of preconditioner construction significantly declines as more processors are used. Of particular interest is that contrary to the unpreconditioned case the iteration count for the solution of the ParaSAILS preconditioned systems is barely dependent on the Reynolds number. The aim of this project, functioning as a starting point for preconditioner selection for more complex systems of equations, puts a lot of emphasis on the apparent lack of sensitivity of ParaSAILS to the nature of the equations being solved. Further confirmation of the scope of sparse approximate inverse preconditioning in

general is found in its successful application to the solution of dense linear systems [2, 36, 37, 38, 39, 44, 45, 74].

Ultimately, the Stokes flow test case indicates that sparse approximate inverse preconditioning has the most potential for more complex problems even though software issues prevent a clear negative verdict concerning parallel incomplete factorization preconditioners in general.

Appendix A

Derivation of the weak formulations

A.1 Amphi3D

A.1.1 The momentum equation

To derive the weak formulation of the momentum equation, we collect all terms of (2.9) on the left-hand side, multiply with the vector basis function $\tilde{\boldsymbol{v}}$ and integrate over the entire domain:

$$\int_\Omega \left\{ \rho \left(\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \nabla \boldsymbol{v} \right) - \nabla \cdot (-p\boldsymbol{I} + \tau) - G\nabla\phi - \rho\boldsymbol{g} \right\} \cdot \tilde{\boldsymbol{v}} \, d\Omega = 0 \tag{A.1}$$

which, defining $Q = -p\boldsymbol{I} + \tau$ to simplify further manipulation, can be rewritten as

$$\int_\Omega \left\{ \rho \left(\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \nabla \boldsymbol{v} - \boldsymbol{g} \right) - G\nabla\phi \right\} \cdot \tilde{\boldsymbol{v}} - \{\nabla \cdot Q\} \cdot \tilde{\boldsymbol{v}} \, d\Omega = 0 \tag{A.2}$$

Let us focus on the last term, bearing in mind that the colon product reduces to the sum of the elementwise multiplication of its operands for second tensor rank

real tensors:

$$\{\nabla \cdot Q\} \cdot \tilde{\boldsymbol{v}} = \sum_i \tilde{\boldsymbol{v}}_i \left[\sum_j \frac{\partial Q_{ij}}{\partial x_j} \right] \tag{A.3}$$

$$= \sum_j \left[\sum_i \tilde{\boldsymbol{v}}_i \frac{\partial Q_{ij}}{\partial x_j} \right] \tag{A.4}$$

$$= \sum_j \sum_i \left[\frac{\partial}{\partial x_j} \left(\tilde{\boldsymbol{v}}_i Q_{ij} \right) - Q_{ij} \frac{\partial \tilde{\boldsymbol{v}}_i}{\partial x_j} \right] \tag{A.5}$$

$$= \sum_j \left[\frac{\partial}{\partial x_j} \left(\sum_i \tilde{\boldsymbol{v}}_i Q_{ij} \right) \right] - \sum_{i,j} Q_{ij} \frac{\partial \tilde{\boldsymbol{v}}_i}{\partial x_j} \tag{A.6}$$

$$= \nabla \cdot (Q\tilde{\boldsymbol{v}}) - \sum_{i,j} Q_{ij} \left(\nabla \tilde{\boldsymbol{v}} \right)_{ij} \tag{A.7}$$

$$= \nabla \cdot (Q\tilde{\boldsymbol{v}}) - Q : \nabla \tilde{\boldsymbol{v}}_i \tag{A.8}$$

Which, plugged back into (A.2) yields

$$\int_\Omega \left\{ \rho \left(\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \nabla \boldsymbol{v} - \boldsymbol{g} \right) - G \nabla \phi \right\} \cdot \tilde{\boldsymbol{v}} + Q : \nabla \tilde{\boldsymbol{v}}_i \, d\Omega - \int_\Omega \nabla \cdot (Q\tilde{\boldsymbol{v}}) \, d\Omega = 0 \tag{A.9}$$

where the term $\nabla \cdot (Q\tilde{\boldsymbol{v}})$ was separated from the main integral, as it follows from the divergence theorem that

$$\int_\Omega \nabla \cdot (Q\tilde{\boldsymbol{v}}) \, d\Omega = \int_{\partial\Omega} (Q\tilde{\boldsymbol{v}}) \cdot \boldsymbol{n} \, d\partial\Omega \tag{A.10}$$

$$= \int_{(\partial\Omega)_u} (Q\tilde{\boldsymbol{v}}) \cdot \boldsymbol{n} \, d(\partial\Omega)_u + \int_{(\partial\Omega)_\tau} (Q\tilde{\boldsymbol{v}}) \cdot \boldsymbol{n} \, d(\partial\Omega)_\tau \tag{A.11}$$

Both these integral vanish, as the first, over the solid wall part of the boundary, equals zero due to the boundary condition $\boldsymbol{v} = 0$ on $(\partial\Omega)_u$, and the other part is set to zero [133], implying the boundary condition (2.22)

$$(-p\boldsymbol{I} + \tau) \cdot \boldsymbol{n} = 0 \qquad (\partial\Omega)_\tau \tag{A.12}$$

To wrap up, the last term of (A.9) vanishes, and with the expansion of Q the weak formulation ensues (2.16):

$$\int_\Omega \left\{ \left[\rho \left(\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \nabla \boldsymbol{v} - \boldsymbol{g} \right) - G \nabla \phi \right] \cdot \tilde{\boldsymbol{v}} + (-p\boldsymbol{I} + \tau) : \nabla \tilde{\boldsymbol{v}} \right\} \, d\Omega = 0 \tag{A.13}$$

A.1.2 The continuity equation

The weak formulation of the continuity equation $\nabla \cdot \boldsymbol{v} = 0$ follows directly upon its multiplication by $\tilde{p}$ and integration over the domain:

$$\int_{\Omega} (\nabla \cdot \boldsymbol{v}) \tilde{p} \, d\Omega = 0 \tag{A.14}$$

A.1.3 The stress equation

The idea of deriving the weak formulation of a vector equation by means of a dot product with a vector basis function is a simple and effective way to reduce the vector equation to a scalar weak formulation. This approach is naturally extended to real second tensor rank tensor equations when the dot product is replaced by the colon product as described in appendix A.1.1. Thus, the weak formulation of (2.7) follows as the colon product of its terms collected on the left-hand side with $\tilde{\tau}$ and subsequent integration over Ω:

$$\int_{\Omega} \left\{ \tau_d + \lambda_H \tau_{d(1)} - \mu_p \left[\nabla \boldsymbol{v} + (\nabla \boldsymbol{v})^T \right] \right\} : \tilde{\tau} \, d\Omega = 0 \tag{A.15}$$

A.1.4 The Cahn-Hilliard equation

The first half of the decomposed Cahn-Hilliard equation (2.14), once collected on the left-hand side, multiplied by $\tilde{\phi}$ and integrated over Ω, comes to

$$\int_{\Omega} \left\{ \frac{\partial \phi}{\partial t} + \boldsymbol{v} \cdot \nabla \phi - \frac{\gamma \lambda}{\epsilon^2} \Delta(\psi + s\phi) \right\} \tilde{\phi} \, d\Omega = 0 \tag{A.16}$$

The Laplacian term can be simplified further, as

$$-\int_{\Omega} \Delta(\psi + s\phi) \tilde{\phi} \, d\Omega = -\int_{\Omega} \nabla \cdot \left[\nabla(\psi + s\phi) \right] \tilde{\phi} \, d\Omega \tag{A.17}$$

$$= -\int_{\Omega} \nabla \cdot \left[\nabla(\psi + s\phi) \tilde{\phi} \right] - \nabla(\psi + s\phi) \cdot \nabla \tilde{\phi} \, d\Omega \tag{A.18}$$

$$= -\int_{\partial\Omega} \nabla(\psi + s\phi) \tilde{\phi} \cdot \boldsymbol{n} \, d\partial\Omega + \int_{\Omega} \nabla(\psi + s\phi) \cdot \nabla \tilde{\phi} \, d\Omega \tag{A.19}$$

When $G = \lambda(\psi + s\phi)$, as determined in Section (2.2), is substituted into the boundary condition (2.12), the boundary integral vanishes, yielding (2.19):

$$\int_{\Omega} \left[\left(\frac{\partial \phi}{\partial t} + \boldsymbol{v} \cdot \nabla \phi \right) \tilde{\phi} + \frac{\gamma \lambda}{\epsilon^2} \nabla(\psi + s\phi) \cdot \nabla \tilde{\phi} \right] d\Omega = 0. \tag{A.20}$$

Collection on the left-hand side, multiplication with the basis function $\tilde{\psi}$ and integration of the second half of the decomposed Cahn-Hilliard equation (2.15) gives

$$\int_\Omega \left\{\psi + \epsilon^2 \Delta\phi - (\phi^2 - 1 - s)\phi\right\} \tilde{\psi}\, d\Omega = 0 \tag{A.21}$$

Again, focus lies with the second order term

$$\begin{aligned}
\int_\Omega \left\{\epsilon^2 \Delta\phi\right\} \tilde{\psi}\, d\Omega &= \int_\Omega \epsilon^2 \nabla \cdot (\nabla\phi\tilde{\psi}) - \epsilon^2 \nabla\phi\nabla\tilde{\psi}\, d\Omega && \text{(A.22)}\\
&= \int_{\delta\Omega} \epsilon^2 \nabla\phi\tilde{\psi} \cdot \boldsymbol{n}\, d\partial\Omega - \int_\Omega \epsilon^2 \nabla\phi\nabla\tilde{\psi}\, d\Omega && \text{(A.23)}\\
&= -\int_{\delta\Omega} \frac{\epsilon^2}{\lambda} f_w'(\phi)\tilde{\psi} \cdot \boldsymbol{n}\, d\partial\Omega - \int_\Omega \epsilon^2 \nabla\phi\nabla\tilde{\psi}\, d\Omega && \text{(A.24)}
\end{aligned}$$

Where the last line was derived with the boundary condition (2.13). Plugging this back into (A.21) immediately results in (2.20):

$$\int_\Omega \left\{ \left[\psi - (\phi^2 - 1 - s)\phi\right] \tilde{\psi} - \epsilon^2 \nabla\phi \cdot \nabla\tilde{\psi} \right\} d\Omega - \int_{\partial\Omega} \frac{\epsilon^2}{\lambda} f_w'(\phi)\tilde{\psi}\, dS = 0 \tag{A.25}$$

A.2 Stokes flow

A.2.1 The momentum equation

To derive the weak formulation of the momentum equation, all terms of (4.6) are collected on the left-hand side, multiplied by the vector basis function $\tilde{\boldsymbol{v}}$ and integrated over the entire domain:

$$\int_\Omega \left\{-\nu\nabla^2\boldsymbol{v} + \boldsymbol{v}\cdot\nabla\boldsymbol{v} + \nabla p\right\}\cdot\tilde{\boldsymbol{v}}\,d\Omega = 0 \tag{A.26}$$

The first term within the integral can be rewritten using the divergence theorem,

$$\int_\Omega \left\{-\nu\nabla^2\boldsymbol{v}\right\}\cdot\tilde{\boldsymbol{v}}\,d\Omega = -\nu\int_\Omega \nabla\cdot(\nabla\boldsymbol{v}\cdot\tilde{\boldsymbol{v}}) - \nabla\boldsymbol{v} : \nabla\tilde{\boldsymbol{v}}\,d\Omega \tag{A.27}$$

$$= -\int_{\partial\Omega}\nu\frac{\partial\boldsymbol{v}}{\partial\boldsymbol{n}}\cdot\tilde{\boldsymbol{v}}\,d\partial\Omega + \int_\Omega \nu\nabla\boldsymbol{v} : \nabla\tilde{\boldsymbol{v}}\,d\Omega \tag{A.28}$$

and similarly the second term of Equation (A.26) is be rewritten as

$$\int_\Omega \nabla p\cdot\tilde{\boldsymbol{v}}\,d\Omega = \int_\Omega \nabla\cdot(p\tilde{\boldsymbol{v}}) - p\nabla\cdot\tilde{\boldsymbol{v}}\,d\Omega \tag{A.29}$$

$$= \int_{\partial\Omega}\boldsymbol{n}p\cdot\tilde{\boldsymbol{v}}\,d\partial\Omega - \int_\Omega p\nabla\cdot\tilde{\boldsymbol{v}}\,d\Omega \tag{A.30}$$

At this point, Equations (A.28) and (A.30) are plugged back into (A.26), yielding

$$\int_\Omega \nu\nabla\boldsymbol{v} : \nabla\tilde{\boldsymbol{v}} + (\boldsymbol{v}\cdot\nabla\boldsymbol{v})\cdot\tilde{\boldsymbol{v}} - p\nabla\cdot\tilde{\boldsymbol{v}}\,d\Omega + \int_{\partial\Omega}\left\{\boldsymbol{n}p - \nu\frac{\partial\boldsymbol{v}}{\partial\boldsymbol{n}}\right\}\cdot\tilde{\boldsymbol{v}}\,d\partial\Omega = 0 \tag{A.31}$$

However, $\tilde{\boldsymbol{v}}$ vanishes on $(\partial\Omega)_{in}$ and $(\partial\Omega)_{wall}$ by the homogeneous equivalent of boundary condition (4.3) and (4.5), respectively. As furthermore $\boldsymbol{n}p - \nu\frac{\partial\boldsymbol{v}}{\partial\boldsymbol{n}} = 0$ on $(\partial\Omega)_{wall}$ by boundary condition (4.4), the entire boundary integral vanishes. This immediately leads to Equation (4.8).

A.2.2 The penalized continuity equation

The weak formulation of the penalized continuity equation $\nabla\cdot\boldsymbol{v} + \frac{\epsilon}{\nu}p = 0$ follows directly upon its multiplication by $\tilde{p}$ and integration over the domain:

$$\int_\Omega (\nabla\cdot\boldsymbol{v} + \frac{\epsilon}{\nu}p)\tilde{p}\,d\Omega = 0 \tag{A.32}$$

Appendix B

The integration of PETSc in Amphi3D

This appendix presents the primary software issues encountered during the parallelization of Amphi3D with PETSc, which serve as examples of the kind of considerations involved with parallelization through PETSc.

B.1 Initialization, input and output

As Amphi3D is implemented in Fortran 90, the include statements that PETSc necessitates in the sourcefiles require a C preprocessor. This is handled by PETSc through compile rules which are imported into the Amphi3D makefiles by further include statements. The first commands in Amphi3D initialize PETSc; explicitly initializing MPI is not required as PETSc does so automatically. As in the sequential code, an input file with various parameters has to be read by the program, and in the parallel version each process simply reads its local copy of this file. The next step in initializing the calculations is reading the domain boundary data and generating an initial mesh with GRUMMP. As GRUMMP is a sequential code and works with files which are written and read several times throughout the creation of the initial mesh, it was necessary to circumvent the issue of multiple processes writing to one file at the same time by either letting each process develop its own initial mesh or let one process do the honors and have all the other processes wait and then read the initial mesh it produced. The former option was attempted, but consistently led to a slightly different initial mesh on one of the processes. After an extenstive investigation of the GRUMMP source code, this approach was abandoned considering the limited effect the initialization as a whole has on the overall performance of the code. Thus, the code was modified to have only one

process read the boundary data and call GRUMMP to develop the initial mesh. It writes this mesh and some related data to file, upon which these are read by all other processes. To force the other processes to wait on the writing process, an MPI barrier is inserted between the two, which forces all processes to reach that point before moving on. This barrier is one of the few functions used in the code where MPI is interfaced directly, rather than through PETSc.

Amphi3D updates the mesh whenever the interface has moved too far for the mesh to capture it accurately, which generally takes something in the order of 10 timesteps. When this occurs, the situation is handled identically to the initial mesh, with only one process employing GRUMMP to update the mesh and all others waiting for it to finish doing so. Throughout the program, pure output is generated by one process only.

B.2 PETSc datastructures

As PETSc is involved only in solving the linear systems in the code, very few of the original datastructures have been converted to PETSc types. The most important four are the system matrix, a vector for the right-hand side and two solution vectors.

The right-hand side vector and one of the solution vectors are parallel vectors, which PETSc automatically distributes evenly over the processors at their creation. The matrix is of type MPIAIJ, which is PETSc's default type for a parallel matrix. It stores the matrix in CSR format as did the original Amphi3D code, and assigns each process a consecutive range of rows. The number of rows owned by each process can be explicitly supplied, or left for PETSc to determine. In the latter case, PETSc splits the matrix such that each process owns corresponding portions of both the matrix and vectors of the same size which already exist, in this case the right-hand side vector and one of the solution vectors. As the matrices which appear within Amphi3D exhibit no natural sections or decomposition, there is no incentive to alter this distribution of data.

The second solution vector is a local copy on each of the processors of the distributed solution vector, created upon each solution of a linear system. It is necessitated by KSP, PETSc's environment of Krylov Subspace methods, which does not allow a mixture of a distributed matrix with a sequential solution matrix on one hand and the necessity of access to the entire solution vector required by each process to continue the inexact Newton method on the other. The sequential vector is filled with a vector scatter. Though there are possibilities to access off-process data in a distributed vector, those suitable for Fortran 90 only work with select compilers and compatibility issues prevented switching to one of them. Though keeping two versions of the same data seems cumbersome and wasteful of

memory, it was tested and confirmed to take a negligible amount of time and as a sequential solution vector is inherently required, the excess use of memory on each process is limited to its portion of the distributed vector. Vector data generally requires negligible memory compared to the system matrix, and furthermore the local portion of the vector dwindles as the number of processes is increased.

B.3 Matrix memory preallocation

The PETSc manual [4] urges the user to employ memory preallocation when creating large matrices. Though this requires some work to predetermine the amount of memory the matrix will require, it prevents all or most of the dynamic memory allocation, which is inherently a very slow process. To do so, one has to supply the routine creating the matrix with the expected number of nonzeros per row in the diagonal and off-diagonal part of the matrix seperately. On each process, the diagonal part of the matrix is the restriction of the local, rectangular submatrix such that the column range equals the range of local rows, and the off-diagonal part consists of all other columns of the local submatrix.

The expected number of nonzeros can be supplied in the form of two integers, each being an estimate for the number of nonzeros of each type in every row, or in the form of two vectors, allowing different numbers of nonzeros to be specified for each row. The length of these vectors, therefore, is the number of rows that each process owns.

This choice constitutes a trade-off between time spent predetermining the matrix structure on the one hand and time spent in the actual assembly of the matrix. The approach taken was to aim at exact preallocation first and later investigate whether the performance could be improved by shifting the balance towards a lighter preallocation algorithm. Thus, the first step was to devise a scheme to calculate the number of nonzeros in each row and how many off these are in the diagonal and off-diagonal parts of the matrix.

The simplest way to determine the nonzero structure of the matrix would be to assemble it. However, the very purpose here is to determine it before the matrix is created. The approach taken here is to mimic the assembly process and retain only its loops creating and placing the element matrices while omitting all calculations concerned with their values, thus marginalizing the work involved. This scheme is dubbed preassembly and its implementation is shown in Figure B.1.

In the assembly process, the nonzero elements are generally a sum of contributions from different element matrices. Therefore, it does not suffice to count the number of times a matrix element is written in each row. Rather, it is necessary to construct, for each row, a list containing the column numbers of the elements which have so far been classified as nonzero. Then, each time an element which

would be a nonzero is selected, it has to be checked whether it is already in the list. If it is, no further action is taken; otherwise it is appended to the list.

In order to keep track of diagonal and off-diagonal nonzeros in each row, there are two possibilities. The first would be to keep one list for both and to count the number of nonzeros of each type upon completion of the list. The other option is to keep two separated lists for diagonal and off-diagonal nonzeros. The latter approach is taken here as it decreases the average length of the search which is conducted to check whether a nonzero entry is already in the list. This was confirmed to yield a significant performance improvement over keeping a single list. There is no additional memory required for this approach, as both lists can be kept in the same vector, simply letting each start at a different end of the vector. In either case, the length of this vector has to be sufficient to accomodate the largest number of nonzeros in any particular row. It is important to have a reasonably tight estimate the maximum number of nonzeros in order to save memory allocation time. However, an overly conservative estimate will not increase the overall memory requirement of the program as these vectors of the relatively cheap `real*4` data type are deleted before the system matrix is created. The check to ensure the list was large enough is executed once it is completed. As such a list needs to be kept for each row, they can be conveniently combined into a matrix.

In Line 1 of the code extract detailing the preallocation, Figure B.1, this matrix, `nonzeros`, is allocated the number of rows of the matrix that a process owns, `iEnd-iStart`, and a number of columns corresponding to an estimated upper limit, `maxnz`, for the number of nonzeros in each row. Note that `iStart` is the first row owned by a process, but `iEnd` is the last row owned plus one. This is a PETSc convention. The next two lines set `diagnz` and `offdiagnz` to zero. These vectors count the number of diagonal and offdiagonal nonzeros in each row, respectively. Line 4 starts the main loop over all elements. For each element, `FORMiglo` is called to determine which global variable each variable of the current element refers to. Line 6, then, starts the loop over the rows of the element matrices and in Line 7 the local row `i` is translated into a global row `in`. Line 8 applies a permutation of the global variables, like the real assembly, which is designed to concentrate the matrix's nonzeros around the diagonal. Line 9 determines whether the rows in question belongs to the local process, and abandons further calculations if it is not. Lines 10 through 13 do for the columns what Line 6 through 9 did for the rows. However, Line 13 determines whether nonzeros found in the current column would be diagonal or off-diagonal, but does not cancel further calculations either way.

Lines 14 and 15 check whether the column at hand, `jk`, equals any of those previously recorded as nonzeros, stored in the first `diagnz(ik-iStart)` elements of `nonzeros(ik-iStart,:)`. If so, the loop is broken by skipping to Line 19. If,

on the other hand, the column at hand wasn't previously recorded as nonzero, `diagnz(ik-iStart)` is increased by 1 in Line 17 and the column number is placed in `nonzeros(ik-iStart,diagnz(ik-iStart))` in Line 18. Though by Line 20 the diagonal columns have been filtered out, it is still necessary to use an `elseif` statement to ensure that the columns are within the range of the matrix. Columns outside of the matrix are associated with boundary conditions and require no consideration here. Lines 21 through 26 do for the offdiagonal columns what lines 14 through 19 did for the diagonal columns. Note that both use the same matrix `nonzeros` but offdiagonal nonzeros are placed at the end rather than the at start of each row. This code has been tested and consistently resulted in exact preallocated memory.

The variation of the number of nonzeros between rows is considerable, due in part to the mixture of linear (P1) and quadratic (P2) elements used in the finite element formulation (2.2). Therefore, it is no surprise that attempts to calcalute only an average or maximal number of nonzers, in other words use integers instead of vectors to preallocate memory, resulted in decreased performance of the overall code. Another argument for exact preallocation is that the work involved for each process decreases with the number of processors whereas the cost of inexact matrix preallocation during matrix assembly does not.

B.4 Matrix Assembly

The matrix assembly was only slightly modified from the sequential code, in the sense that the generation of the element matrices remains unchanged and only their placement in the system matrix was altered. In fact, it was simplified as the translation between the natural row and column indices are now translated into CSR data by PETSc, whereas this was formerly explicit in the assembly of the system matrix. The code in Figure B.2 shows the assembly, which is largely identical to the preassembly in Figure B.1. Line 1 simply opens the loop over the elements and lines 3 through 13 check whether any of the global variables associated with it correspond to rows of the system matrix owned by the local process. If not, further calculations for this element are skipped by means of the `goto` statement in Line 12 to avoid unnecessary work. If, on the other hand, any or all of the element's variables are relevant to the current process, the values of the element matrix are calculated (this occurs between Line 13 and 14 of Figure B.2). Finally, lines 14 through 21 determine the global position of the variables in the element matrix exactly as in the preassembly, line 21 verifies that it lies within the bounds of the matrix and the values of the element matrix `st` are added to the system matrix `Apet` in line 22. Please note that `ione` is simply an integer with value one and that this is subtracted from the indices as PETSc uses 0-based

```
1   allocate (nonzeros(iEnd-iStart,maxnz))
2   diagnz=0
3   offdiagnz=0
4   do m = 1,nelem
5     call FORMiglo (m, iglo, [...] )
6     do i = 1,nvel
7       in = iglo(i)
8       ik = iperm(in)
9       if( (ik.gt.iStart) .and. (ik.le.iEnd) ) then
10        do j = 1,nvel
11          jn = iglo(j)
12          jk = iperm(jn)
13          if( (jk.gt.iStart) .and. (jk.le.iEnd) )then
14            do k = 1,diagnz(ik-iStart)
15              if( jk.eq.nonzeros(ik-iStart,k) ) goto 11
16            enddo
17            diagnz(ik-iStart) = diagnz(ik-iStart) +1
18            nonzeros(ik-iStart,diagnz(ik-iStart)) = jk
19  11        continue
20          elseif( (jk.gt.0) .and. (jk.le.neq) )then
21            do k = maxnz, maxnz-offdiagnz(ik-iStart)+1, -1
22              if(jk.eq. nonzeros(ik-iStart,k) )goto 12
23            enddo
24            offdiagnz(ik-iStart) = offdiagnz(ik-iStart)+1
25            nonzeros(ik-iStart,maxnz-offdiagnz(ik-iStart)+1) = jk
26  12        continue
27          endif
28        enddo
29      endif
30    enddo
31  enddo
```

Figure B.1: Preassembly for the matrix

indices wheres Fortran is principally 1-based. Attempts to forego calculations concerning specific variables in element matrices which are only partially relevant to the local process were abandoned as this added more computational work in the form of selecting if-statements than it would prevent in not calculating unused values.

```
1    do m = 1,nelem
[...]
2      call FORMiglo (m, iglo, [...] )
3      relevant = .false.
4      do i = 1,nvel
5        in = iglo(i)
6        ik = iperm(in)
7        if( (ik .gt. iStart) .and. (ik .le. iEnd) )then
8          relevant = .true.
9          goto 13
10       endif
11     enddo
12     goto 14
13  13 continue
[...]
14     do i=1,nvel
15       in = iglo(i)
16       ik = iperm(in)
17       if( (ik .gt. iStart) .and. (ik .le. iEnd) )then
18         do j=1,nvel
19           jn = iglo(j)
20           jk = iperm(jn)
21           if(  (jk.gt.0) .and. (jk.le.neq) )then
22            call MatSetValues(Apet,ione,ik-ione,ione,
     &                            jk-ione,st(i,j),ADD_VALUES,ierr)
23           endif
24         enddo
25       endif
26     enddo
27  14 continue
28   enddo
```

Figure B.2: Matrix and vector assembly

Bibliography

[1] J. I. Aliaga, M. Bollhöfer, A. F. Martín, and E. S. Quintana-Ortí. Design, tuning and evaluation of parallel multilevel ILU preconditioners. In *VECPAR*, pages 314–327, 2008.

[2] G. Alléon, M. Benzi, and L. Giraud. Sparse approximate inverse preconditioning for dense linear systems arising in computational electromagnetics. *Numer. Algorithms*, 16, 1997.

[3] D. M. Anderson, G. B. McFadden, and A. A. Wheeler. Diffuse-interface methods in fluid mechanics. *Annual Review of Fluid Mechanics*, 30:139–165, 1998.

[4] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[5] R. E. Bank and R. K. Smith. The incomplete factorization multigraph algorithm. *SIAM J. Sci. Comput.*, 20(1349), 1999.

[6] R. E. Bank and R. K. Smith. An algebraic multilevel multigraph algorithm. *SIAM J. Sci. Comput.*, 23(1572), 2002.

[7] R. E. Bank and C. Wagner. Multilevel ILU decomposition. *Numer. Math.*, 82(543), 1999.

[8] S. T. Barnard, L. M. Bernardo, and H. D. Simon. An MPI implementation of the SPAI preconditioner on the T3E. *Int. J. High Perform. Comput. Appl.*, 13(2):107–123, 1999.

[9] S. T. Barnard and R. L. Clay. A portable MPI implementation of the SPAI preconditioner in ISIS++. In M. H. et al., editor, *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Philadelphia, 1997. SIAM. CD-ROM.

[10] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the solution of linear systems: Building blocks for iterative methods*. SIAM, Philadelphia, 1994.

[11] M. W. Benson. Iterative solution of large scale linear systems. Master's thesis, Lakehead Univ., Thunder Bay, 1973.

[12] M. W. Benson and P. O. Frederickson. Iterative solution of large sparse linear systems arising in certain multidimensional approximation problems. *Util. Math.*, 22(127), 1982.

[13] M. Benzi. Preconditioning techniques for large linear systems: A survey. *Journal of Computational Physics*, 182(2):418 – 477, 2002.

[14] M. Benzi, J. K. Cullum, and M. Tůma. Robust approximate inverse preconditioning for the conjugate gradient method. *SIAM J. Sci. Comput.*, 22(1318), 2000.

[15] M. Benzi, W. Joubert, and G. Mateescu. Numerical experiments with parallel orderings for ILU preconditioners, 1999.

[16] M. Benzi, J. Marín, and M. Tůma. A two-level parallel preconditioner based on sparse approximate inverses. In D. R. Kincaid and A. C. Elster, editors, *Iterative Methods in Scientific Computation IV*, New Brunswick, NJ, 1993. IMACS.

[17] M. Benzi, C. D. Meyer, and M. Tůma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM J. Sci. Comput.*, 17(1135), 1996.

[18] M. Benzi and M. Tůma. An assessment of some preconditioning techniques in shell problems. *Commun. Numer. Methods Eng.*, 14(897), 1998.

[19] M. Benzi and M. Tůma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM J. Sci. Comput.*, 19(968), 1998.

[20] M. Benzi and M. Tůma. A comparative study of sparse approximate inverse preconditioners. *Appl. Numer. Math.*, 30(305), 1999.

[21] M. Benzi and M. Tůma. A parallel solver for large-scale Markov chains. *Appl. Numer. Math.*, 41(135), 2002.

[22] L. Bergamaschi, G. Pini, and F. Sartoretto. Approximate inverse preconditioning in the parallel solution of sparse eigenproblems. *Numer. Linear Algebra Appl.*, 7(99), 2000.

[23] L. Bergamaschi, G. Pini, and F. Sartoretto. Parallel preconditioning of a sparse eigensolver. *Parallel Comput.*, 27(963), 2001.

[24] R. B. Bird, C. F. Curtiss, R. C. Armstrong, and O. Hassager. *Dynamics of Polymeric Liquids, Vol. 2. Kinetic Theory.* Wiley, New York, 1987.

[25] M. Bollhöfer and V. Mehrmann. Algebraic multilevel methods and sparse approximate inverses. *SIAM J. Matrix Anal. Appl.*, 24(1):191–218, 2002.

[26] M. Bollhöfer and Y. Saad. A factored approximate inverse preconditioner with pivoting. *SIAM J. Matrix Anal. Appl.*, 23(692), 2002.

[27] J. Borggaard and T. Burkardt. NS3D FEM: Steady Navier Stokes equations in 3D - Finite Element solution, May 2009. `http://people.sc.fsu.edu/~burkardt/m_src/ns3d_fem/ns3d_fem.html`.

[28] E. F. F. Botta and F. Wubs. Matrix renumbering ILU: An effective algebraic multilevel ILU preconditioner for sparse matrices. *SIAM J. Matrix Anal. Appl.*

[29] R. Bridson and W. P. Tang. Ordering, anisotropy and factored sparse approximate inverses. *SIAM J. Sci. Comput.*, 21(867), 1999.

[30] R. Bridson and W. P. Tang. Refining an approximate inverse. *J. Comput. Appl. Math.*, 22(1527), 2000.

[31] R. Bridson and W. P. Tang. Multiresolution approximate inverse preconditioners. *SIAM J. Sci. Comput.*, 23(463), 2001.

[32] O. Bröker and M. J. Grote. Sparse approximate inverse smoothers for geometric and algebraic multigrid. *Appl. Numer. Math.*, 41(61), 2002.

[33] O. Bröker, M. J. Grote, C. Mayer, and A. Reusken. Robust parallel smoothing for multigrid via sparse approximate inverses. *SIAM J. Sci. Comput.*, 23(1395), 2001.

[34] A. M. Bruaset. *A survey of preconditioned iterative methods.* Longman Scientific and Technical, Harlow, 1995.

[35] J. W. Cahn. Critical point wetting. *The Journal of Chemical Physics*, 66(8):3667–3672, 1977.

[36] B. Carpentieri, I. S. Duff, L. Giraud, and M. M. monga Made. Sparse symmetric preconditioners for dense linear systems in electromagnetics. Technical Report TR/PA/01/35, CERFACS, Toulouse, France, 2001.

[37] K. Chen. On a class of preconditioning methods for dense linear systems from boundary elements. *SIAM J. Sci. Comput.*, 20, 1998.

[38] K. Chen. Discrete wavelet transforms accelerated sparse preconditioners for dense boundary element systems. *Electron. Trans. Numer. Anal.*, 8, 1999.

[39] K. Chen. An analysis of sparse approximate inverse preconditioners for boundary integral equations. *SIAM J. Matrix Anal. Appls.*, 22, 2001.

[40] E. Chow. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM J. Sci. Comput.*, 21(1804), 2000.

[41] E. Chow. Parallel implementation and performance characteristics of sparse approximate inverse preconditioners with a priori sparsity patterns. *Int. J. High-Perform. Comput. Appl.*, 15(56), 2001.

[42] E. Chow and Y. Saad. Approximate inverse preconditioners via sparse-sparse iterations. *SIAM J. Sci. Comput.*, 19(995), 1998.

[43] A. J. Cleary, R. D. Falgout, V. E. Henson, and J. E. Jones. Coarse-grid selection for parallel algebraic multigrid. In *Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 104–115, 1998.

[44] E. Darve. The fast multipole method. I. Error analysis and asymptotic complexity. *SIAM J. Numer. Anal.*, 38, 2000.

[45] E. Darve. The fast multipole method: Numerical implementation. *J. Comput. Phys.*, 160, 2000.

[46] E. de Doncker and A. K. Gupta. Coarse grain preconditioned conjugate gradient solver for large sparse systems. In J. Lewis, editor, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, page 472, Philadelphia, 1995. SIAM.

[47] E. de Sturler. Incomplete block LU preconditioners on slightly overlapping subdomains for a massively parallel computer. *Appl. Numer. Math.*, 19(1-2):129–146, 1995.

[48] J. C. Díaz and K. Komara. Incomplete multilevel cholesky factorizations. *SIAM J. Matrix Anal. Appl.*, 22(895), 2000.

[49] J. J. Dongarra, L. S. Duff, D. C. Sorensen, and H. A. V. Vorst. *Numerical Linear Algebra for High Performance Computers*. SIAM, Philadelphia, PA, USA, 1998.

[50] M. Dryja and O. B. Widlund. Domain decomposition algorithms with small overlap. *SIAM Journal on Scientific Computing*, 15(3):604–620, 1994.

[51] I. S. Duff, A. M. Erisman, C. W. Gear, and J. K. Reid. Sparsity structure and Gaussian elimination. *SIGNUM Newsl.*, 23(2):2–8, 1988.

[52] I. S. Duff and G. Meurant. The effect of ordering on preconditioned conjugate gradients. *BIT*, 29(635), 1989.

[53] S. C. Eisenstat and H. F. Walker. Choosing the forcing terms in an inexact Newton method. *SIAM J. Sci. Comput*, 17:16–32, 1996.

[54] R. D. Falgout and U. M. Yang. Hypre: A library of high performance preconditioners. In *Lecture Notes in Computer Science*, pages 632–641, 2002.

[55] M. Field. a parallel factorised sparse approximate inverse preconditioner with improved choice of sparsity pattern. Technical Report HDL-TR-99-214 Hitachi Dublin Laboratory, Dublin, Ireland.

[56] M. Field. An efficient parallel preconditioner for the conjugate gradient algorithm. Technical Report HDL-TR-97-175 Hitachi Dublin Laboratory, Dublin, Ireland.

[57] J. Ford. A black box at the end of the rainbow: searching for the perfect preconditioner. *Phil. Trans. Royal Soc. London A*, 361(1813):2665–2680, 2003.

[58] P. O. Frederickson. Fast approximate inversion of large sparse linear systems. *Math. Report*, 7, 1975.

[59] L. A. Freitag and C. Ollivier-Gooch. Tetrahedral mesh improvement using swapping and smoothing. *International Journal for Numerical Methods in Engineering*, 40:3979–4002, 1997.

[60] S. Fujino and Y. Ikeda. An improvement of SAINV and RIF preconditionings of CG method by double dropping strategy. pages 142–149, July 2004.

[61] A. George and J. W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, Englewood Cliffs, NJ, 1981.

[62] M. J. Grote and T. Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM J. Sci. Comput*, 18:838–853, 1997.

[63] S. R. Gundolf Haase, Michael Kuhn. Parallel AMG on distributed memory computers. *SIAM Journal on Scientific Computing*, 24(2):410–427, 2000.

[64] I. Gustafsson. A class of first order factorization methods. *BIT*, 18(142), 1978.

[65] V. E. Henson and U. M. Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Appl. Numer. Math.*, 41(155), 2002.

[66] H. H. Hu, N. A. Patankar, and M. Y. Zhu. Direct numerical simulations of fluid-solid systems using the arbitrary Lagrangian-Eulerian technique. *Journal of Computational Physics*, 169:427–462, May 2001.

[67] D. Hysom. *New Sequential and Scalable Parallel Algorithms for Incomplete Factor Preconditioning*. PhD thesis, Old Dominion Univ., Norfolk, Va, 2001.

[68] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM J. Sci. Comput.*, 22(6):2194–2215, 2001.

[69] D. Jacqmin. Calculation of two-phase Navier-Stokes flows using phase-field modeling. *Journal of Computational Physics*, 155(1):96–127, October 1999.

[70] D. Jacqmin. Contact-line dynamics of a diffuse fluid interface. *Journal of Fluid Mechanics*, 402(-1):57–88, 2000.

[71] G. Karypis and V. Kumar. Parallel threshold-based ILU factorization. Technical Report 061, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, 1998.

[72] S. A. Kharchenko, L. Y. Kolotilina, A. A. Nikishin, and A. Y. Yeremin. A robust AINV-type method for constructing sparse approximate inverse preconditioners in factored form. *Numer. Linear Algebra Appl.*, 8(165), 2001.

[73] D. A. Knoll and D. E. Keyes. Jacobian-free Newton-Krylov methods: a survey of approaches and applications. *Journal of Computational Physics*, 193(2):357 – 397, 2004.

[74] L. Y. Kolotilina. Explicit preconditioning of systems of linear algebraic equations with dense matrices. *J. Sov. Math.*, 43, 1988.

[75] L. Y. Kolotilina and A. Y. Yeremin. Factorized sparse approximate inverse preconditionings I: theory. *SIAM J. Matrix Anal. Appl.*, 14(1):45–58, 1993.

[76] L. Y. Kolotilina and A. Y. Yeremin. Factorized sparse approximate inverse preconditionings II: Solution of 3D FE systems on massively parallel computers. *Int. J. High Speed Comput.*, 7(191), 1996.

[77] A. Krechen and K. Stüben. Parallel algebraic multigrid based on subdomain blocking. *Parallel Comput.*, 27(1009), 2001.

[78] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing; design and analysis of algorithms.* Benjamin/Cummings, Redwood City, 1994.

[79] S. L. Lee. Krylov methods for the numerical solution of initial-value problems in differential-algebraic equations. Technical report, Champaign, IL, 1993.

[80] N. Li, Y. Saad, and M. Sosonkina. pARMS: A parallel version of the algebraic recursive multilevel solver. Technical Report UMSI-2001-100, Minnesota Supercomputer Institute, Univ. Minnesota, Minneapolis, 2001.

[81] J. B. Lifschitz, A. A. Nikishin, and A. Y. Yeremin. Sparse approximate inverse preconditionings for solving 3D CFD problems on massively parallel computers. In R. Beauwens and P. de Groen, editors, *Proceedings of the IMACS International Symposium*, pages 83–84, Brussels, 1992.

[82] J. Lowengrub and L. Truskinovsky. Quasi-incompressible Cahn-Hilliard fluids and topological transitions. *Proceedings: Mathematical, Physical and Engineering Sciences*, 454(1978):2617–2654, 1998.

[83] M. Luby. A simple parallel algorithm for the maximal independent set problem. In *STOC '85: Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 1–10, New York, NY, USA, 1985. ACM.

[84] S. Ma. Comparisons of the ILU(0), Point-SSOR, and SPAI preconditioners on the CRAY-T3E for nonsymmetric sparse linear systems arising from PDEs on structured grids. *Int. J. High Perform. Comput. Appl.*, 14(1):39–48, 2000.

[85] S. Ma. A performance comparison of the parallel preconditioners for iterative methods for large sparse linear systems arising from partial differential equations on structured grids. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, E91-A(9):2578–2587, 2008.

[86] G. A. Meurant. Computer solution of large linear systems. *Studies in Mathematics and Its Applications*, 28, 1999.

[87] G. A. Meurant. Numerical experiments with algebraic multilevel preconditioners. *Electron. Trans. Numer. Anal.*, 12(1), 2001.

[88] G. A. Meurant. A multilevel AINV preconditioner. *Numer. Algorithms*, 29(107), 2002.

[89] T. Mifune, T. Iwashita, and M. Shimasaki. A parallel algebraic multigrid solver for fast magnetic edge-element analyses. *ieee transactions on Magnetics*, 41(5), 2005.

[90] M. M. monga Made and H. A. van der Vorst. Parallel incomplete factorizations with pseudo-overlapping subdomains. *Parallel Comput.*, 27(989), 2001.

[91] V. A. Mousseau, D. A. Knoll, and W. J. Rider. Physics-based preconditioning and the Newton-Krylov method for non-equilibrium radiation diffusion. *Journal of Computational Physics*, 160(2):743 – 765, 2000.

[92] M. F. Murphy, G. H. Golub, and A. J. Wathen. A note on preconditioning for indefinite linear systems. *SIAM J. Sci. Comput.*, 21(6):1969–1972, 1999.

[93] Y. Notay. Using approximate inverses in algebraic multilevel methods. *Numer. Math.*, 80(397), 1998.

[94] Y. Notay. A multilevel block incomplete factorization preconditioning. *Appl. Numer. Math.*, 31(209), 1999.

[95] A. Padiy, O. Axelsson, and B. Polman. Generalized augmented matrix preconditioning approach and its application to iterative solution of ill-conditioned algebraic systems. *SIAM J. Matrix Anal. Appl.*, 22(793), 2001.

[96] G. Radicati and Y. Robert. Parallel conjugate gradient-like algorithms for solving sparse non-symmetric linear systems on a vector multiprocessor. *Parallel Comput.*, 11:223–239, 1989.

[97] A. Rafiei and F. Toutounian. New breakdown-free variant of AINV method for nonsymmetric positive definite matrices. *Journal of Computational and Applied Mathematics*, 219(1):72 – 80, 2008.

[98] A. Reusken. A multigrid method based on incomplete Gaussian elimination. *Numer. Linear Algebra Appl.*, 3(369), 1996.

[99] J. W. Ruge and K. Stüben. Algebraic multigrid (AMG). In S. F. McCormick, editor, *Multigrid Methods, Applied Mathematics volume 3*, pages 73–130, Philadelphia, 1987. SIAM.

[100] Y. Saad. Highly parallel preconditioners for general sparse matrices. In *Recent Advances in Iterative Methods*, page 165, New York, 1994. Springer-Verlag.

[101] Y. Saad. ILUT: A dual threshold incomplete LU factorization. *Numer. Linear Algebra Appl.*, 1(387), 1994.

[102] Y. Saad. *Iterative Methods for Sparse Linear Systems, 2nd edition.* SIAM, Philadelpha, PA, 2003.

[103] Y. Saad and M. H. Schultz. Parallel implementation of preconditioned conjugate gradient methods. In W. Fitzgibbon, editor, *Mathematical and Computational Methods in Seismic Exploration and Reservoir Modeling.*

[104] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[105] Y. Saad and B. Suchomel. ARMS: An algebraic recursive multilevel solver for general sparse linear systems. *Numer. Linear Algebra Appl.*, 9(359), 2002.

[106] Y. Saad and H. A. van der Vorst. Iterative solution of linear systems in the 20th century. *J. Comput. Appl. Math.*, 123(1-2):1–33, 2000.

[107] Y. Saad and J. Zhang. BILUM: Block versions of multielimination and multilevel preconditioner for general sparse linear systems. *SIAM J. Sci. Comput.*, 20(2103), 1999.

[108] Y. Saad and J. Zhang. BILUTM: A domain-based multilevel block ILUT preconditioner for general sparse matrices. *SIAM J. Sci. Comput.*, 21(279), 1999.

[109] M. K. Seager. Parallelizing conjugate gradient for the CRAY X-MP. *Parallel Comput.*, 3(1):35–47, 1986.

[110] C. Shen and J. Zhang. Parallel two level block ILU preconditioning techniques for solving large sparse linear systems. *Paral. Comput*, 28:2002, 2002.

[111] G. L. G. Sleijpen and D. R. Fokkema. BiCGStab(ℓ) for linear equations involving unsymmetric matrices with complex spectrum. *ETNA*, 1:11–32, 1993.

[112] G. L. G. Sleijpen and H. A. van der Vorst. An overview of approaches for the stable computation of hybrid BiCG methods. *Appl. Numer. Math.*, 19(3):235–254, 1995.

[113] B. F. Smith, P. E. Bjrstad, and W. D. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge Univ. Press, Cambridge, 1996.

[114] R. D. Smith, J. K. Dukowicz, and R. C. Malone. Parallel ocean general circulation modeling. *Phys. D*, 60(38), 1992.

[115] K. Stüben. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 128:281–309, 2001.

[116] M. Sunhaloo, A. Gopaul, R. Boojhawon, and M. Bhuruth. Sparse approximate inverse smoothing for multigrid solution of nine-point approximations for convection-diffusion problems. In *Proceedings of the 2005 International Conference on Scientific Computing*, pages 52–58, 2005.

[117] W. P. Tang and W. L. Wan. Sparse approximate inverse smoother for multigrid. *SIAM J. Matrix Anal. Appl.*, 21(1236), 2000.

[118] H. P. Taylor, C. A numerical solution of the Navier-Stokes equations using the finite element technique. *Computers and Fluids*, 1(1):73–100, 1973.

[119] L. N. Trefethen and D. Bau. *Numerical Linear Algebra.* SIAM, Philadelphia, 1997.

[120] A. van der Ploeg, E. F. F. Botta, and F. W. Wubs. Nested grids ILU-decomposition (NGILU). *J. Comput. Appl. Math.*, 66(515), 1996.

[121] H. A. van der Vorst. Large tridiagonal and block tridiagonal linear systems on vector and parallel computers. *Parallel Comput.*, 5(45), 1987.

[122] H. A. van der Vorst. High performance preconditioning. *SIAM J. Sci. Stat. Comput.*, 10(6):1174–1185, 1989.

[123] H. A. van der Vorst. *Iterative Krylov methods for large linear system.* Cambridge University Press, 2003.

[124] C. Vuik, R. R. P. van Nooyen, and P. Wesseling. Parallelism in ILU-preconditioned GMRES. *Parallel Computing*, 24(14):1927 – 1946, 1998.

[125] J. S. Warsa, T. A. Wareing, and J. E. Morel. Solution of the discontinuous P1 equations in two-dimensional Cartesian geometry with two-level preconditioning. *SIAM J. Sci. Comput.*, 24(6):2093–2124, 2002.

[126] T. Washio and K. Hayami. Parallel block preconditioning based on SSOR and MILU. *Numerical Linear Algebra with Applications*, 1(6):533–553, 1994.

[127] A. J. Wathen. Preconditioning and fast solvers for incompressible flow, 2004. Numerical Analysis Group Research Report, Oxford University, NA-04/08.

[128] J. Watts. A conjugate gradient truncated direct method for the iterative solution of the reservoir simulation pressure equation. *Soc. Petrol. Eng. J.*, 21(345), 1981.

[129] G. Wittum. On the robustness of ILU-smoothing. *SIAM J. Sci. Stat. Comput.*, 10(699), 1989.

[130] U. M. Yang. Parallel algebraic multigrid methods - high performance preconditioners. In A. M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, pages 209–236. Springer-Verlag, 2006.

[131] P. Yue, J. J. Feng, C. Liu, and J. Shen. A diffuse-interface method for simulating two-phase flows of complex fluids. *Journal of Fluid Mechanics*, 515(-1):293–317, 2004.

[132] P. Yue, C. Zhou, and J. J. Feng. Can the Cahn-Hilliard model quantitatively describe moving contact lines? *J. Fluid Mech.*, submitted, 2008.

[133] P. Yue, C. Zhou, J. J. Feng, C. F. Ollivier-Gooch, and H. H. Hu. Phase-field simulations of interfacial dynamics in viscoelastic fluids using finite elements with adaptive meshing. *J. Comput. Phys.*, 219(1):47–67, 2006.

[134] J. Zhang. Sparse approximate inverse and multilevel block ILU preconditioning techniques for general sparse matrices. *Appl. Numer. Math.*, 35(67), 2000.

[135] C. Zhou, P. Yue, J. J. Feng, C. Ollivier-Gooch, and H. H. Hu. 3D phase-field simulations of interfacial dynamics in viscoelastic fluids with variable contact angles using finite elements with adaptive meshing. *J. Comput. Phys.*, submitted, 2009.