A parallel solver for PDE systems and application to the incompressible Navier-Stokes equations *

Eero Vainikko and Ivan G. Graham

25th March 2003

Department of Mathematical Sciences,
University of Bath, Bath BA2 7AY, UK

Keywords: Parallel iterative methods, domain decompostion, partial differential equations, incompressible Navier Stokes equations.

Abstract

The DOUG (Domain Decomposition on Unstructured Grids) package was first developed 1996-98 as a parallel solver for scalar elliptic PDEs discretised with finite elements. In this paper we describe its extension to unsymmetric elliptic systems of PDEs, highlighting software design and parallelisation issues. As an application we discuss the performance of the extended package applied to the incompressible Navier-Stokes equations, discretised with mixed finite elements. In particular we focus on discontinuous pressure elements, which are important in many practical applications. We also indicate briefly the application of this solver to Navier-Stokes stability assessment.

Acknowledgement. We thank Andrew Cliffe for kindly providing us with the test problems and the illustrations in §4 and also for very many useful discussions.

1 Introduction

The DOUG package [8, 12] is a parallel solver, based on domain decomposition, for discretised steady state PDEs on unstructured grids in 2D or 3D domains. It has been used in a number of applications, e.g. [3], but up till now was restricted to scalar problems, and had been principally applied in the symmetric case.

The first purpose of this paper is to describe the extension of the DOUG code to discrete systems of (unsymmetric) PDEs, given in block format, and with blocks given in either element or assembled format. We describe in detail the parallelisation of the FGMRES iterative method for these systems, focussing on parallel implementation of general block systems.

The second purpose is to describe the application of the code to the solution of systems arising from (the linearisation of) the steady state incompressible Navier-Stokes equations discretised using mixed finite element methods. The main additional issue is the design of

*Supported by UK Engineering and Physical Sciences Research Council Grant GR/M59075.

preconditioners which are powerful enough to solve large systems for high Reynolds number. We investigate this issue for discretisations based on discontinuous pressure elements, which are widely used in industrial applications. We compare several preconditioners, in terms of their iteration counts and parallel performance and select the best one for solving systems of this type. This section of the paper is an extension of the recent paper [9, 10] which considered only continuous pressure elements.

The third purpose of the paper is to indicate briefly the application of our parallel PDE solver to the computation of eigenvalue problems for the Navier-Stokes equations, linearised about a computed flow. This application is also discussed in more detail in the recent papers [9, 10]. The kernel of the algorithm is the repeated solution of the (shifted) linearised Navier-Stokes equations, for which we are able to use the block-enabled `DOUG` code described above.

We illustrate the performance of the code on the computation of some challenging flow problems and we give results illustrating the parallel performance of the algorithm on a AMD Athlon Beowulf cluster.

2 `DOUG` package and extension to block systems

2.1 Iterative methods for large nonsymmetric systems

Iterative methods are often used for solving linear systems of equations

$$A\mathbf{x} = \mathbf{b} \tag{1}$$

with large sparse matrices A. The most common are Krylov subspace methods such as BICGSTAB or GMRES [20]. The Krylov subspace method of our choice is the inner-outer Flexible GMRES (FGMRES) method [19]. Our choice is motivated by the fact that the problems discussed below are relatively hard, and although the preconditioner used reduces the number of iterations of the *standard* GMRES method substantially, this number is typically in the hundreds instead of the tens usually achieved for easier e.g. symmetric problems. The drawback of the standard preconditioned GMRES method is that it needs to store all Krylov vectors for the purpose of orthogonalisation of the next search direction. If the number of iterations is large, the available memory becomes a serious restriction to the overall implementation with large matrices.

In the standard (right) preconditioned GMRES method for (1) with right preconditioner P, the problem is rewritten as

$$AP^{-1}\tilde{\mathbf{x}} = \mathbf{y} \tag{2}$$

where the solutions of (1) and (2) are related by $P\mathbf{x} = \tilde{\mathbf{x}}$. Iterative solution of (2) requires repeated multiplication by A, together with solves of systems of the form $P\mathbf{z} = \mathbf{r}$. The standard GMRES algorithm computes the approximate solution which minimises the residual of (2) in the Krylov space

$$\mathrm{span}\left\{\mathbf{r}_0, AP^{-1}\mathbf{r}_0, ..., (AP^{-1})^{k-1}\mathbf{r}_0\right\}, \tag{3}$$

where $\mathbf{r}_0$ is the initial residual. To compute the GMRES solution requires the computation and storage of an orthonormal basis of the Krylov space, which becomes costly as the number of iterations k gets large. Moreover since the method is based on residual minimisation, the preconditioner P has to be kept fixed throughout the solution process.

The Flexible GMRES method [19] allows the preconditioner P to vary, from iterate to iterate, at the expense of having to store not only an orthonormal basis $\{\mathbf{v}_1, ..., \mathbf{v}_k\}$ for the Krylov space (3) but also the solutions $\mathbf{z}_j$ of $P\mathbf{z}_j = \mathbf{v}_j, \quad j = 1, ..., k$. However this extra expense pays off in practice if we use FGMRES as the outer iteration and take P to be the action of an inner solver chosen as the standard preconditioned GMRES method, up to some tolerance. With this combination the number of outer iterations is greatly reduced so the total storage requirement and the total number of orthogonalisation operations is reduced. It turns out that the extra calls to the preconditioner and matrix-vector multiplications with A do not cancel out this advantage. This is especially true in the case of parallel implementations as the orthogonalisation procedure requires the computation of dot products which are often communication bottlenecks in parallel algorithms.

Thus, for the parallel implementation of any iterative solver, the following parallel operations are needed: (i) Vector updates, i.e., $\mathbf{x} = \mathbf{y} + \mathbf{z}$. (ii) Dot products, i.e., $\mathbf{x} \cdot \mathbf{y}$. (iii) Matrix-vector multiplications, i.e., $\mathbf{x} = A\mathbf{y}$. (iv) Preconditioning step, i.e. solving $P\mathbf{z} = \mathbf{r}$. These operations are given here in the order of communication and computational complexity with the vector updates needing no communication at all and the application of the preconditioner being the most demanding part.

2.2 Domain Decomposition

Domain Decomposition (DD) is a generic technique for generating parallel preconditioners for Krylov subspace methods like CG, BiCGSTAB and (F)GMRES. DD methods can be divided roughly into two sets of methods: Substructuring methods and Additive Schwarz methods (see e.g. [22]). Substructuring methods are based on the reduction of the system onto subdomain interfaces while in Additive Schwarz methods subdomain solvers are used. Usually the subdomains have some overlap, although the Additive Average method [1] is an example of an Additive Schwarz method without overlap. In this work we discuss only Additive Schwarz methods where the subdomains have small overlap. This "one level method" is usually accelerated by using a global solve on a coarser grid - the "two level method".

The computational package `DOUG` (Domain Decomposition on Unstructured Grids) [8, 12] is a general parallel iterative solver for symmetric and unsymmetric systems of discretised steady state PDEs on 2D and 3D domains. The code has been developed at the University of Bath since 1996 ([12, 8, 3]). It employs a two level technique and is applicable to general PDEs on unstructured grids. Another implementation of parallel Domain Decomposition methods for general linear systems is the pARMS project [17].

2.3 Problem partitioning

The first public release of the `DOUG` code was for solving linear systems (1) arising from finite element discretisations of scalar elliptic PDEs. The user had to provide the stiffness matrix A and load vector $\mathbf{b}$ in elemental form, which is natural only with finite element discretisations. In the current `DOUG` version, matrices can be supplied also in assembled form, and thus it can be used for systems arising from finite difference and finite volume discretisation schemes as well.

Suppose that the given sparse system (1), corresponding for the moment to a scalar PDE, is to be solved, and let Λ denote the set of *freedoms* for this problem (i.e. Λ is an index set for the rows and columns in A and the vector $\mathbf{x}$ has an entry for each element in Λ.

In most PDE applications, Λ can also be identified with a set of linear functionals of the approximate solution of the PDE.) In `DOUG` the division of the freedoms Λ into subdomains is automatic and is achieved by partitioning a suitable "dual graph" into disconnected subsets. The subsets of the dual graph induce a subdivision of Λ into subsets with small overlap (for example a line of freedoms in the case of 2D problems and a surface of freedoms in the 3D case). The partitioning of the dual graph is achieved using the graph partitioning software package METIS [13], which strives to produce well-balanced partitions, while minimising the number of broken edges.

When the matrix is given in elemental form, the dual graph is constructed as follows: The nodes are taken to be the finite elements and the edges connect each pair of elements which touch. Partitioning the dual graph yields pairwise disjoint subsets of the set of elements. The union of each subset of elements forms a geometrical subdomain Ω_i, say, of the original PDE domain, Ω, say. Denoting the set of freedoms in $\overline{\Omega}_i$ by Λ_i, we thus obtain an overlapping partition $\Lambda_i : i = 1, \ldots, s$ of the original set of freedoms, Λ. Any given freedom lies in some one of the sets Λ_i and it lies in the intersection $\Lambda_i \cap \Lambda_j$ if and only if it is contained in both $\overline{\Omega}_i$ and $\overline{\Omega}_j$.

If the matrix is specified in assembled form, effective partitioning of Λ can be computed in an algebraic way by first constructing a "primal graph" with nodes corresponding to freedoms in the linear system. An edge is drawn between freedoms p and q if the corresponding entry $A_{p,q}$ of the given matrix A is non-zero. Sparsity implies that each freedom is connected to a relatively small number of other freedoms. Then the dual graph has its nodes taken to be the edges of the primal graph, and edges in the dual graph connect pairs of edges of the primal graph which have a common freedom. Graph partitioning applied to the dual graph again yields pairwise disjoint subsets, Ω_i say, of the set of all edges, Ω say, of the primal graph. The union of the freedoms connected by the edges in Ω_i forms a subset Λ_i of all the freedoms. Again we have a partition of the original set of freedoms Λ. Any given freedom lies in some one of the sets Λ_i and it lies in the intersection $\Lambda_i \cap \Lambda_j$ if and only if it lies on edges contained in both Ω_i and Ω_j.

In any case, because of the properties of the graph partitioning software, we can assume that the set of freedoms Λ is partitioned into s subdomains Λ_i , $i = 1, \ldots, s$ of approximately equal sizes. This domain decomposition is the basis of the parallelisation of the operations 1 - 4 above.

The DOUG code uses a master-slave model to implement this domain decomposition. The master node is responsible for problem subdivision, the slave nodes do subdomain computations. Each of the slave processors works out datastructures for neighbour communication of the freedoms on the subdomain overlaps. The slaves have knowledge of the position in the global numbering for each interface freedom and are also sent information about other processors that they share the interface freedoms with. At each stage in the iterative method, the current values of the interface freedoms are replicated on neighbouring subdomains. For operations like dot products, to ensure that each freedom is referenced only once during read operations, a uniqueness flag is set to 1 on a single one of the slaves which owns that freedom and 0 on all the other slaves which own it. The domain decomposition allows the parallel computation of vector updates and dot products in an obvious way.

For the matrix-vector multiplications and for the preconditioning step, it is useful to have in mind two slightly different decompositions of the matrix A. For the matrix-vector operations, it is useful to consider local matrices $\tilde{A}_i$ with the same dimension as A, but with non-zeros only in the rows and columns corresponding to the freedoms in Λ_i and chosen so that:

$$A\mathbf{x} = \sum_{i=1}^{s} \tilde{A}_i \mathbf{x} \quad \text{for all vectors} \quad \mathbf{x} \ .$$

These can be obtained for example by setting

$$(\tilde{A}_i)_{p,q} = A_{p,q}/n(p,q) \ , \quad \text{when } p \text{ and } q \in \Lambda_i \ ,$$

where $n(p,q)$ denotes the number of subdomains Λ_i which contain both of the freedoms p and q. (Hence $n(p,q)$ is different from 1 only when p and q both lie in more than one Λ_i.) In the case of matrices defined elementwise, we may alternatively define $\tilde{A}_i$ to be simply the assembled stiffness matrices corresponding to natural boundary conditions on the subdomain Ω_i.

For the preconditioning step we introduce slightly different local matrices A_i, chosen simply to be the submatrices of A corresponding to rows and columns in Λ_i. (In the case of finite element matrices given in elemental form, these have an easy interpretation as stiffness matrices corresponding to the underlying PDE discretised on Ω_i, and subject to essential boundary conditions on the extension of Ω_i obtained by adding to it all the elements which touch it.) The local matrices A_i have dimension $\#\Lambda_i \times \#\Lambda_i$ and their inverses are used in the preconditioners described in the following sections.

Obviously the local matrices A_i and $\tilde{A}_i$ need not both be stored, since their non-zero entries differ only on the interfaces of the Λ_i. In the case of element matrices, the action of $\tilde{A}_i$ is realised by adding element matrices in Ω_i. The action of the inverses A_i^{-1} requires information about the element matrices for all the elements containing the freedoms in Λ_i. For interface freedoms, the missing element information is obtained by exchanging data with the neighbours and then summing up before factorising A_i.

The most significant programming task is the parallelisation of the preconditioning step 4. This we now describe in some detail.

2.4 One-level preconditioner

The one-level Additive Schwarz preconditioner may be written schematically, as

$$P^{-1}\mathbf{x} = \sum_{i=1}^{s} R_i^T A_i^{-1} R_i \mathbf{x} \ , \tag{4}$$

where R_i takes a global vector $\mathbf{x}$ to its subvector of entries corresponding to freedoms in Λ_i and R_i^T takes a vector defined on Λ_i and extends by zero to a vector on Λ. Computation of the action of P^{-1} thus involves independent subdomain solves with each A_i followed by additions of neighbor subdomain solutions for freedoms which lie on the intersection of more than one subdomain.

In the present version of `DOUG`, the subdomain problems are solved directly using a multifrontal solver for sparse matrices. In our case we use UMFPACK [5]. Such direct solvers perform well up to certain matrix sizes, especially for 2D PDEs. For this reason, and to allow for flexible use of the code on different parallel platforms, after distribution of the submatrices to processors, each partition Λ_i gets possibly further split locally to obtain an optimal subdomain size n_s. The value of n_s depends on the computer architecture and the rate at which the sparse direct solver works. While $n_s = 1000$ was used in 1996, $n_s = 1400$ was found experimentally to be near optimal in 2002 for scalar elliptic problems. For Navier Stokes problems the optimal value was found to be approximately 4000. Hence on any given platform each processor may be solving in serial one or more subdomain problems using a sparse direct solver as part of the preconditioning step.

Communication appears in the dot product operations (in an obvious way) and, after each Ax and preconditioner application, it is needed for updating the corresponding values on freedoms which lie in more than one Λ_i. In order to ensure good scalability it is necessary to overlap communication with computation throughout the algorithm. We will describe in detail how this is done in the case of Ax operations (for PDE systems) in Section 2.6. Similar techniques can be used for the preconditioning step.

We remark that two extensions to the present `DOUG` approach could easily be made. First the subdomains Λ_i could be extended further, to provide a more generous overlap with their neighbours. The result will generally be a more powerful preconditioner, but requiring more commnunication. Second, the exact multifrontal solves with the A_i could be replaced with inexact solves, say by several steps of an iterative method.

2.5 Two-level preconditioner

The two-level method involves the construction and solution of a coarse version of the original system, in addition to the subdomain solves described above. The inclusion of the coarse solve is essential in the PDE case if the preconditioner is to behave robustly with respect to growth in the total number of freedoms (e.g. [2]).

One option for creating a coarse problem is to apply a coarsening algorithm to the original mesh and rediscretise the PDE on the coarsened mesh. However, since the coarsened mesh may not cover the same spatial domain as the fine mesh there are some technical difficulties in the interpretation and implementation of boundary conditions [2]. Another alternative (the one we use here) is to assemble the coarse matrix by suitably interpolating the fine matrix. This has the advantage that the boundary conditions already built into the fine matrix are inherited by the coarse matrix.

To create the coarsened problem, the present version of the `DOUG` code requires further geometric information on the given problem. (Purely algebraic variants could also be implemented, for example the AMG techniques in [18], but we have not implemented these here. Since all our applications come from finite element methods, the geometric procedure described below is possible).

Thus, let us first assume that we are solving a scalar PDE specified on a domain Ω and approximated in some finite element space $\mathcal{S}^h(\Omega)$ of piecewise polynomials with respect to some mesh on Ω. We assume that we are given enough information on Ω so that we can construct a bounding polygonal or polyhedral domain Ω^e which overlaps Ω (i.e. $\Omega \cup \partial\Omega \subset \Omega^e$). We triangulate Ω^e with a coarse mesh (i.e. coarser than the mesh underlying the space $\mathcal{S}^h(\Omega)$), and introduce a finite element space $\mathcal{S}^H(\Omega^e)$ on the coarse mesh. Let Λ^h and Λ^H denote the sets of freedoms for the spaces $\mathcal{S}^h(\Omega)$ and $\mathcal{S}^H(\Omega^e)$, respectively. The basic requirement for the coarsening procedure is the construction of a suitable linear restriction operator $R_0 : \Lambda^h \to \Lambda^H$. Its transpose R_0^T (defined by the transpose of its matrix representation) then forms a prolongation from $\Lambda^H \to \Lambda^h$. The coarse approximation of A is then taken to be $A_0 := R_0 A R_0^T$ and, for the two-level method, the preconditioner consists of inverting A_0 and adding the result to each of the subdomain solves with A_i. Schematically

$$P^{-1} = \sum_{i=0}^{s} R_i^T A_i^{-1} R_i \tag{5}$$

which should be compared with (4).

The construction of effective coarse grids for difficult PDE problems is a topic of considerable importance in present research. A fundamental requirement on R_0 is that

$$\ker R_0^T = \{0\} \ , \tag{6}$$

i.e. there is no nontrivial function on the coarse mesh which becomes zero when prolonged to the fine mesh. When the original matrix A in (1) is symmetric positive definite (SPD) the assumption (6) trivially implies that the coarsened matrix A_0 is also SPD. For certain classes of symmetric positive definite PDE problems and certain families of finite elements, much more can be said about the theoretical performance of (5), in terms of condition number bounds for the preconditioned matrix AP^{-1} (see, e.g. [2]). However for general unsymmetric PDEs this theory is more limited. For example in the unsymmetric case the fundamental requirement (6) is not sufficient even to guarantee non-singularity of the coarse matrix A_0. In the current version of `DOUG`, the coarse matrix problem is solved directly, again using the multifrontal solver. In many of the experiments in §3.2, coarse problems with singular A_0 are solved. In our experience the coarsening procedure is still an effective part of the preconditioner, even in the singular case. It remains an open question whether the implementation of possibly complicated criteria to avoid singularity in the coarse problem would substantially improve the performance of the preconditioner in practice.

In typical applications, the matrix R_0 is computed as follows. Suppose that we have a linear interpolation operator $I^{H\to h} : \mathcal{S}^H(\Omega^e) \to \mathcal{S}^h(\Omega)$. (Usually this would be provided by the interpolation procedure associated with the finite element space $\mathcal{S}^h(\Omega)$.) Suppose that $\{\phi_p^h : p \in \Lambda^h\}$ and $\{\phi_q^H : q \in \Lambda^H\}$ are bases for the finite element spaces $\mathcal{S}^h(\Omega)$ and $\mathcal{S}^H(\Omega^e)$ respectively. Then the matrix R_0 is obtained as the matrix of coefficients in the expansion

$$I^{H\to h}\phi_q^H = \sum_{p\in\Lambda^h} (R_0)_{q,p}\phi_p^h \ , \quad q \in \Lambda^H.$$

Hence $(R_0)_{q,p}$ is found by evaluating the the pth fine freedom of the qth coarse basis function. The matrix R_0 is then typically sparse and for each p, only local information about the elements of the fine mesh containing freedom p are needed to compute $(R_0)_{q,p}$.

In the `DOUG` code, continuous piecewise linear basis functions on triangles in 2D (respectively tetrahedra in 3D) are used for the coarse basis functions ϕ_q^h. It is easy to give conditions sufficient to ensure that (6) is satisfied, for example this holds in the case of Lagrange families of elements, provided each coarse grid element contains three non-collinear (respectively four non-coplanar) fine grid freedoms. The coarsening procedure can be applied to a wide variety of finite elements on the fine grid. In the applications below we shall be working for example with piecewise biquadratic basis functions on rectangular elements.

In many applications the fine mesh is refined in regions where the underlying problem has difficult behaviour. For best performance of the preconditioner, the coarse mesh typically needs to mirror qualitatively this refinement. This is especially true when solving unsymmetric problems with standard (symmetric) Galerkin finite element methods (as we shall do in

§3.2). In this case refinement in the fine mesh is needed just to obtain a *stable* scheme (even before accuracy is considered). In order to generate qualitatively appropriate coarse meshes we employ adaptive locally regular refinement of an initial regular (very) coarse mesh covering the whole domain Ω. The coarse mesh is initially constructed by covering Ω with uniform coarse squares (respectively cubes in 3D). These are recursively refined into smaller squares by subdivision into 4 (resp. 8) until either (i) each coarse square contains fewer than some threshold $\overline{f}$ of fine grid freedoms or (b) an overall complexity limit for the coarse mesh is attained. The hierarchically refined squares (cubes) are then subdivided into triangles (tetrahedra) and continuous piecewise linear approximation is applied using the standard procedure for handling hanging nodes (see [12]). The resulting interpolation matrix R_0^T is formed by the master. The local parts relevant to each subdomain are is distributed to the slaves and the rest of the procedure is as described above.

The slaves are each responsible for assembling part of the coarse matrix A_0, while the master assembles the whole matrix and solves it. In the present version this is again done by a serial direct solver.

In the implementation, the communication required for distribution of the subdomains A_i to the slaves is overlapped with the computation needed for the formation of the coarse matrix A_0. By the time each slave processor has obtained the interpolation matrices R_0^T and after arrival of each entry of the fine matrix, the corresponding components of A_0 are updated. As a result, soon after the A_i's are distributed the local parts of the coarse matrix A_0 have been assembled on the slaves and will be sent to the master for merging together into the matrix A_0.

2.6 Block systems

All the recent computer architectures exploit hierarchical memory structures with automatic cache management. If some data units in the memory are likely to be accessed close together in time, an efficient implementation can be achieved if these units of data are stored closely together in the memory address space to help easier cache pre-fetching. If in the case of block matrices, each of the blocks is likely to be referenced separately, it is best to store the arrays in the memory blockwise.

We will illustrate here our memory organisation structure in the case of matrix-vector multiplication for block systems, where the stiffness matrices are assumed defined via element matrices. We will assume that matrix A is an $n_b \times n_b$ block matrix of the form

$$A = \begin{bmatrix} \mathcal{A}^{11} & \cdots & \mathcal{A}^{1n_b} \\ \vdots & \ddots & \vdots \\ \mathcal{A}^{n_b1} & \cdots & \mathcal{A}^{n_bn_b} \end{bmatrix},$$

with the $\mathcal{A}^{lm}$ being sparse matrix blocks. We have to compute $A\mathbf{x}$, where $\mathbf{x} = (\mathbf{x}^{1^T}, \ldots, \mathbf{x}^{n_b T})^T$, and each $\mathbf{x}^m \in \Lambda^m$, where Λ^m is the set of freedoms associated with the mth block. For $\ell \neq m$ $\mathcal{A}^{\ell,m}$ is in general a sparse rectangular matrix. To compute $A\mathbf{x}$ we have to compute $\mathcal{A}^{\ell,m}\mathbf{x}^m$ for all ℓ, m. On the assumption that the matrices $\mathcal{A}^{\ell,m}$ are defined via element matrices, we can apply the procedure outlined in §2.3 to obtain (possibly overlapping) decompositions Λ_i^m of each Λ^m. These lead to "local" matrices $\tilde{\mathcal{A}}_i^{\ell,m}$ (with the same dimension as $\mathcal{A}^{\ell,m}$ but with

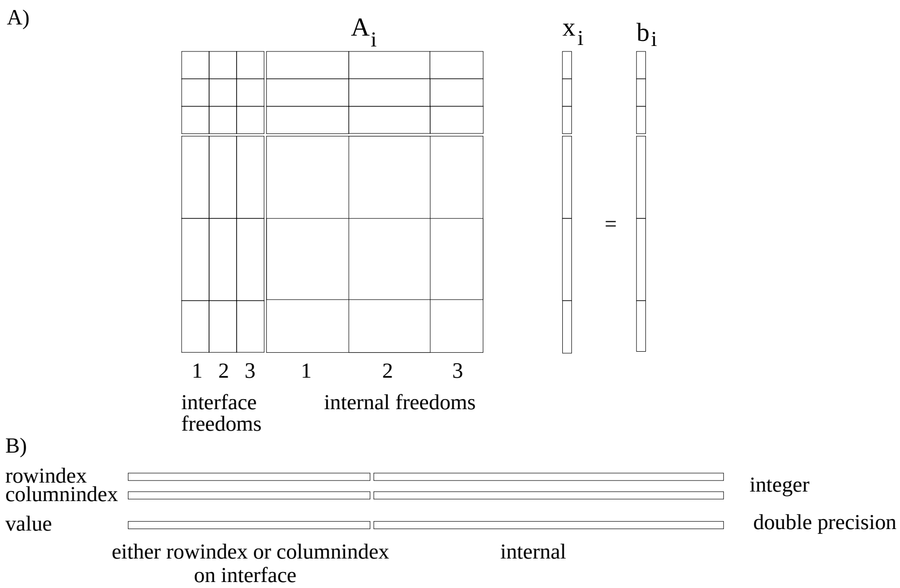

Figure 1: Array preordering: A) rhs-vector, solution and the working arrays, for nblocks = 3; B) sparse matrix storage scheme.

non-zero entries only in rows and columns corresponding to freedoms in $\overline{\Omega}_i$) such that

$$\mathcal{A}^{\ell,m}\mathbf{x}^m = \sum_i \tilde{\mathcal{A}}_i^{\ell,m}\mathbf{x}^m \ .$$

The implementation of this then requires the computation of

$$\tilde{\mathcal{A}}_i^{\ell,m}\mathbf{x}_i^m \ ,$$

where with a small abuse of notation, we can strip out the zeros and think of the row and column indices of $\tilde{\mathcal{A}}_i^{\ell,m}$ as lying in Λ_i^ℓ and Λ_i^m. As mentioned in §2.4, it is important to handle carefully the required communication on the shared freedoms.

To achieve an efficient Ax operation, one would like to begin with the operation of matrix-vector multiplication on the overlapping regions, then start the nonblocking communication doing the multiplication on the inner freedoms in the meantime and end by updating the interface freedom values on arrival of the corresponding messages from the neighbours. This implies also that the interface freedoms must be clustered together in the memory storage. On the other hand, all the matrix values associated with the interface freedoms get accessed at first. Therefore, in our implementation, all the freedoms are preordered into $2n_b$ array blocks and each matrix block $\mathcal{A}_i^{\ell,m}$ gets split into four submatrices

$$\mathcal{A}_i^{\ell,m} = \left[\begin{array}{cc} (\mathcal{A}_i^{\ell,m})_{BB} & (\mathcal{A}_i^{\ell,m})_{BI} \\ (\mathcal{A}_i^{\ell,m})_{IB} & (\mathcal{A}_i^{\ell,m})_{II} \end{array}\right]$$

with B referring to freedoms corresponding to the subdomain boundaries and I denoting inner freedoms. Putting this all together gives us $2n_b \times 2n_b$ matrix stucture on each subdomain. We illustrate this in case $n_b = 3$ in Fig.1 A). To store the matrix A_i we use the standard sparse triple format consisting of two integer arrays giving nonzero row and column indices and the corresponding matrix values in a floating point array. The blocking structure implies that all the submatrix A_i elements, associated with any interface freedoms get stored first, followed by the elements on strictly internal freedoms as shown in Fig.1 B).

3 Navier Stokes Problems

3.1 Linearisations

We shall consider the computation of fluid flows governed by the steady-state incompressible Navier-Stokes equations:

$$\left.\begin{array}{rcl} -\epsilon\Delta\mathbf{w} + \mathbf{w}\cdot\nabla\mathbf{w} + \nabla q &=& \mathbf{f} \\ \nabla\cdot\mathbf{w} &=& 0\ . \end{array}\right\} \tag{7}$$

The solution consists of the vector velocity $\mathbf{w}$, and the scalar pressure q, both defined on a 2D or 3D computational (usually finite) domain, subject to suitable boundary conditions. Here Δ is the Laplace operator, ∇ is the gradient operator and $\nabla\cdot$ denotes divergence. The examples given below are scaled so that the viscosity parameter ϵ is equal to $1/Re$, where Re is the Reynolds number. (For further details see, for example [11].)

Let $\mathbf{w}$ be any velcity field satisfying the incompressibility condition $\nabla\cdot\mathbf{w} = 0$. Then we will be interested in the fast solution of the linearisations of (7) :

$$\left.\begin{array}{rcl} -\epsilon\Delta\mathbf{u} + \mathbf{w}\cdot\nabla\mathbf{u} + \mathbf{u}\cdot\nabla\mathbf{w} + \nabla p &=& \mathbf{f} \\ \nabla\cdot\mathbf{u} &=& 0\ , \end{array}\right\} \tag{8}$$

for the pair $(\mathbf{u}, p)$, saistfying homogeneous boundary conditions. System (8) arises naturally if, for example, Newton's method is used to solve (7). In addition (and this is the prime motivation of our particular work), (8) arises when it is desired to assess the stability of a computed solution $(\mathbf{w}, q)$ of (7). The one needs to solve the eigenvalue problem

$$\left.\begin{array}{rcl} -\epsilon\Delta\mathbf{u} + \mathbf{w}\cdot\nabla\mathbf{u} + \mathbf{u}\cdot\nabla\mathbf{w} + \nabla p &=& \lambda\mathbf{u} \\ \nabla\cdot\mathbf{u} &=& 0\ , \end{array}\right\} \tag{9}$$

for eigenvalue λ and eigenvector $(\mathbf{u}, p)$. The iterative solution of (9) typically requires repeated solves of the linearised problem (8) or shifted versions of it. (See [4, 9, 10] and the references given there for further details.)

3.2 Examples

In this paper we will be concerned with the computation of two challenging examples of the system (8).

(i) Flow around a cylinder. This is a 2D problem in a domain depicted in Figure 2. The applied velocity is $(1,0)^T$ on the top, bottom and left-hand end of the domain, $(0,0)^T$ on the cylinder and $(\mathbf{u}, p)^T$ satisfies the ouflow conditions $\epsilon\frac{\partial u_1}{\partial x} - p = 0 = \frac{\partial u_2}{\partial x}$ at the right-hand end. The domain occupies the region $[-20, 40] \times [-10, 10]$. The circular inclusion (the "cylinder") is of radius 1 with centre at the origin.

It known that the steady state solution of (7) becomes unstable (to a Hopf bifurcation) for Reynolds number lying in the range $23 < Re < 24$. We choose as one test problem, the

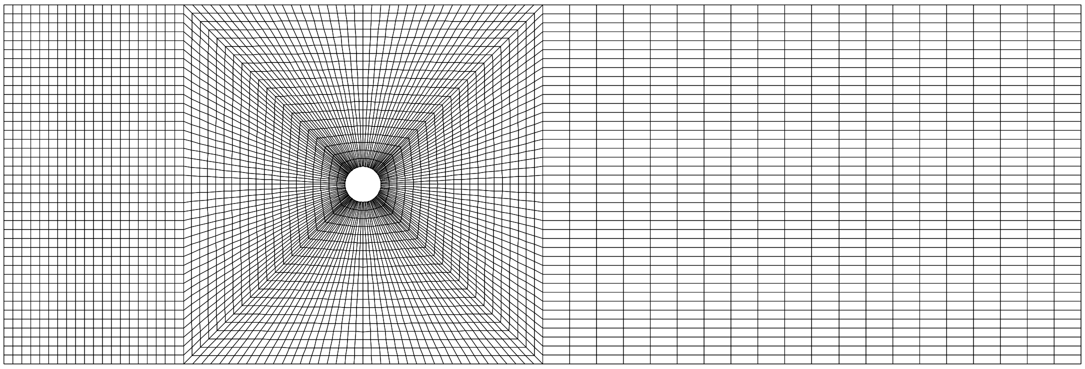

Figure 2: Domain with typical grid for flow around the cylinder problem

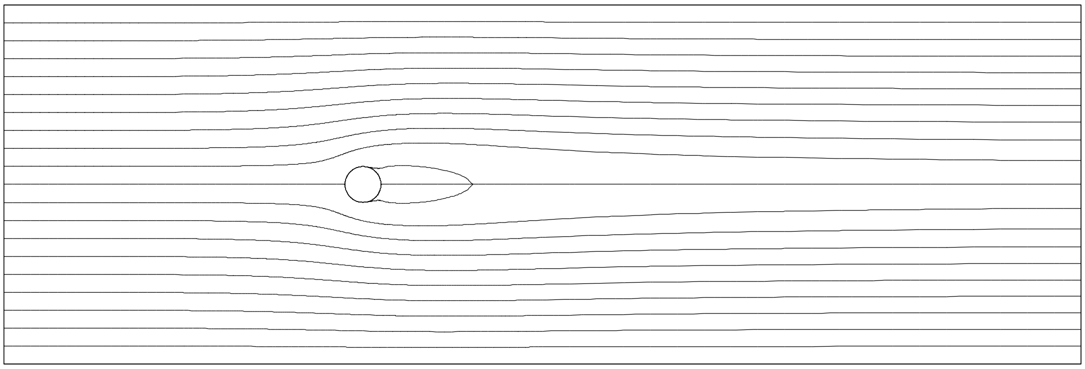

Figure 3: Streamlines for flow around the cylinder problem, $Re = 25$

solution of the linearised problem (8) with $\mathbf{w}$ chosen to be the steady state of (7) and for various Re lying in the range $1 \leq Re \leq 25$. The streamlines of the solution for $Re = 25$ are depicted in Figure 3. The assessment of stability of this solution requires the repeated solution of shifted versions of this system (see [9, 10]).

(ii) Flow in an expanding pipe. This is a 3D problem in an expanding circular pipe. The problem can be written in cylindrical coordinates, with z the axis of symmetry of the pipe, r the radius of the pipe and θ the azimuthal angle. Axisymmetric flows $(\mathbf{w}, q)$ (with all components independent of θ) may be found by solving on a mesh in the $z - r$ plane. A typical mesh is shown in Figure 4. In this example the pipe has a sudden 1:3 expansion and the computational domain, in the $z - r$ plane, is $\{[0, 1] \times [-L_1, 0]\} \cup \{[0, 3] \times [0, L_2]\}$, where L_1 is the length of the inlet section of the pipe and L_2 is the length of the rest of the pipe. Usually the solutions are not very sensitive to the value of L_1. Since the length of the recirculation zone downstream of the expansion is proportional (approxiately) to Re, L_2 should also be proportional to Re and so can be large for high Reynolds number flows. In the computed examples below, $L_1 = 25$ and $L_2 = 100$. (Figures 4,5,6 are scaled in the z direction so that they fit on the page.) The boundary conditions are: (i) $w_r = 0$ and $w_z = 1 - r^2$ on the inlet; (ii) $w_r = w_z = 0$ on the surface of the pipe: ; (iii) $w_r = \frac{\partial w_z}{\partial r} = 0$ on the axis of symmetry; (iv) $\epsilon \frac{\partial w_r}{\partial z} - q = \frac{\partial w_z}{\partial z} = 0$ at the outlet . Streamlines for steady-state axisymmetric solutions of (7) for $Re = 25$ and $Re = 50$ are depicted in Figures 5 and 6.

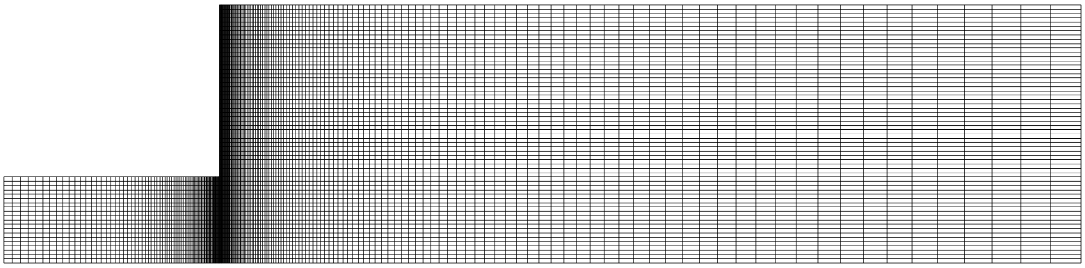

Figure 4: Cross-section of domain in $z-r$ plane with typical grid for expanding pipe problem

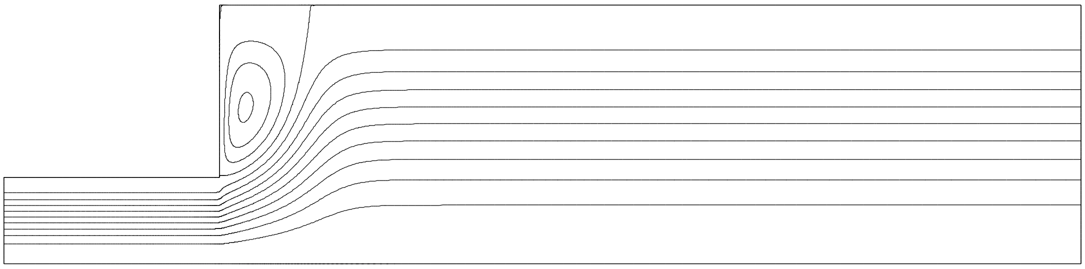

Figure 5: Streamlines for expanding pipe problem, $Re = 25$

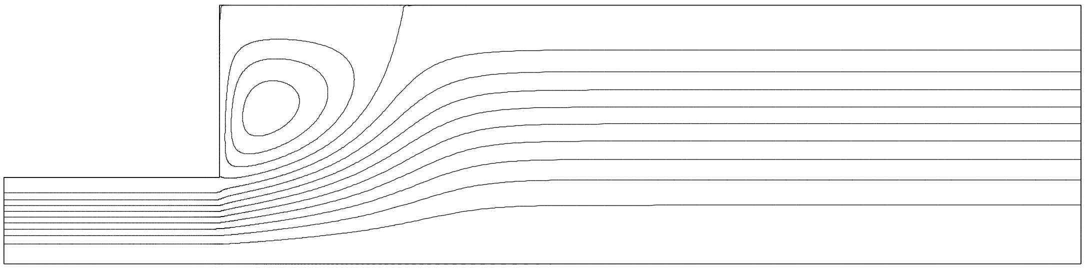

Figure 6: Streamlines for expanding pipe problem, $Re = 50$

It is an outstanding unsolved problem to compute the value of the Reynolds number at which the axisymmetric steady state flow for this problem becomes unstable (to perturbations which may have variation in the θ direction). Although this is a $3D$ eigenvalue problem, a separation of variables technique can reduce it to an infinite sequence of $2D$ eigenvalue problems in the $z-r$ plane, for each of the Fourier modes (with respect to θ) of the eigenfunction of the full $3D$ problem. For our test problem in this case we take the eigenvalue problem corresponding to the Fourier mode which is constant in θ. In cylindrical polar coordinates, this takes the form:

$$-\epsilon\left[\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u_r}{\partial r}\right)+\frac{\partial^2 u_r}{\partial z^2}-\frac{u_r}{r^2}\right] + \left[w_r\frac{\partial u_r}{\partial r}+u_r\frac{\partial w_r}{\partial r}+w_z\frac{\partial u_r}{\partial z}+u_z\frac{\partial w_r}{\partial z}\right]+\frac{\partial p}{\partial r}=\lambda u_r\ , \qquad (10)$$

$$-\epsilon\left[\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u_z}{\partial r}\right)+\frac{\partial^2 u_z}{\partial z^2}\right] + \left[w_r\frac{\partial u_z}{\partial r}+u_r\frac{\partial w_z}{\partial r}+w_z\frac{\partial u_z}{\partial z}+u_z\frac{\partial w_z}{\partial z}\right]+\frac{\partial p}{\partial z}=\lambda u_z\ , \qquad (11)$$

$$\frac{1}{r}\frac{\partial}{\partial r}(ru_r)+\frac{\partial u_z}{\partial z}=0 \qquad (12)$$

Here $\mathbf{w}$ is the axisymmetric steady state flow and $(\mathbf{u}, p)$ is to be found subject to homogeneous boundary conditions: (i) $u_r = u_z = 0$ at the inlet; (ii) $u_r = \frac{\partial u_z}{\partial r} = 0$ at the surface of pipe; (iii) $u_r = u_z = 0$ on the axis of symmetry ; (iv) $\epsilon\frac{\partial u_r}{\partial z} - p = \frac{\partial u_z}{\partial z} = 0$ at the outlet. This is again in the form (8), but in $r-z$ coordinates.

3.3 Mixed Finite Elements

There are many discretisation techniques for (8) which can be applied to the two examples outlined in §3.2. In the case of *mixed finite elements*, different families of piecewise polynomials are used for approximating the velocity $\mathbf{u}$ and for the pressure p. For the above examples, mixed finite element approximation yields the algebraic problem:

$$A\mathbf{x} = \mathbf{y}, \qquad (13)$$

where $\mathbf{x} = (\mathbf{U}_1^T, \mathbf{U}_2^T, \mathbf{P}^T)^T$ is a vector of n degrees of freedom approximating $(u_1, u_2, p)^T$, and the matrix A takes the form:

$$A = \begin{bmatrix} F_{11} & F_{12} & B_1^T \\ F_{21} & F_{22} & B_2^T \\ B_1 & B_2 & 0 \end{bmatrix}\ . \qquad (14)$$

Here the matrix F represents the discretisation of $-\epsilon\Delta\mathbf{u} + \mathbf{w}\nabla\mathbf{u} + \mathbf{u}\nabla\mathbf{w}$, B represents the discretisation of $\nabla.\mathbf{u}$, and the transpose of this matrix represents the discretisation ∇p. (For more details see, for example, [11].) In the more general 3D case, the upper hand matrix F becomes a 3×3 block matrix. Overall, A is unsymmetric, sparse and in real applications n will be very large ($10^5 - 10^9$ is common).

Some elements of our parallelisation technique for (13) apply equally well to discretisation by any family of mixed finite elements. In the `DOUG` code, the computation of matrix-vector products is achieved using the general procedure for PDE systems outlined in §2.6. However the design of preconditioners is somewhat more element dependent. In this paper we shall concentrate on preconditioners which are appropriate when the finite elements used to approximate the pressure p are discontinuous - this is a popular choice in practice. In particular all the numerical results below are for quadrilateral Q_2P_{-1} elements (continuous biquadratic velocities, and discontinuous quadratic pressures) - see [11] for details of this element. The flows illustrated in Figures 3, 5, 6 are computed using this element.

3.4 Parallel Preconditioners

In this section we outline the two preconditioners which will be compared in the results section.

(i) Parallel Block Preconditioner. This consists of a parallelisation of a particular example of a class of preconditioners for (13) (see, e.g. [21]), the action of which is obtained by (approximate) solution of (discretisations of several) scalar PDEs. These can be motivated by the observation of Elman and Silvester [7] that the matrix

$$P = \begin{bmatrix} F_{11} & F_{12} & B_1^T \\ F_{21} & F_{22} & B_2^T \\ 0 & 0 & -X \end{bmatrix},$$

where

$$X = \mathbf{B}\mathbf{F}^{-1}\mathbf{B}^T, \quad \mathbf{F} = \begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} B_1 & B_2 \end{bmatrix}.$$

is a perfect right preconditioner for A in the sense that AP^{-1} has a minimal polynomial of degree two, thus guaranteeing convergence of GMRES in at most two iterations. However, this "perfect" preconditioner requires assembling and inverting the Schur complement X and inverting the 2×2 block matrix $\mathbf{F}$ and is not a practical choice. Elman [6] suggested the more practical approximation for X^{-1}:

$$X_B^{-1} = (\mathbf{B}\mathbf{B}^T)^{-1}(\mathbf{B}\mathbf{F}\mathbf{B}^T)(\mathbf{B}\mathbf{B}^T)^{-1}. \tag{15}$$

With Elman's approximation, action of the whole block preconditioning step, Solve: $P(\mathbf{u}, p)^T = (\mathbf{w}, r)^T$ is achieved with the following algorithm:

(i) Solve $(\mathbf{B}\mathbf{B}^T)p = r$,
(ii) compute $s = -(\mathbf{B}\mathbf{F}\mathbf{B}^T)p$,
(iii) solve $(\mathbf{B}\mathbf{B}^T)p = s$,
(iv) apply $\mathbf{v} = \mathbf{w} - \mathbf{B}^T p$,
(v) solve $\mathbf{F}\mathbf{u} = \mathbf{v}$.

We assemble the matrix $\mathbf{B}\mathbf{B}^T$ by a sparse matrix multiplication algorithm and the product remains sparse. The steps (ii) and (iv) require matrix-vector multiplication with blocks of the discrete PDE system and their parallelisation is obtained using the techniques described in §2.6. Then the steps (i) and (iii) involve solution of sparse systems (with matrix in assembled form). In our algorithm, these solves are replaced by one application of the Additive Schwarz preconditioner detailed above. Finally step (v) requires the solution of a 2×2 (or 3×3) PDE system, with coefficient matrix F. In our implementation we approximate this by setting the off-diagonal matrices in F to 0 and replacing the diagonal blocks by one application of the Additive Schwarz preconditioner. This yields a cheap highly parallel version of the ideal preconditioner of [7]. This preconditioner is labelled "Block" in the tables below.

(ii) Parallel Whole Preconditioner. In the Parallel Block Preconditioner above, the domain decomposition is introduced only after the system is decoupled. An alternative approach is to apply the domain decomposition directly to the whole coupled system. This leads to the solution of a sequence of coupled problems, but reduced to smaller subdomains and is the basis of the Parallel Whole Preconditioner. We shall describe this only in the case of the Q_2P_{-1} elements which we use for the computations in this paper. In this case the velocity is approximated by continuous piecewise quadratic functions with freedoms at the corners, mid-points of the sides and centroids of the quadrilateral elements. The pressure is approximated by discontinuous linear functions on the same elements, with three freedoms taken to be the moments of the function against the three standard basis functions $\{1, x, y\}$ in each element. Quadratic isoparametric mapping is used to handle the geometry in the computational domain for both velocity and pressure elements.

Assuming that the matrices are specified in elemental form, the partitioning procedure appropriate to matrices given in this form is applied to the mesh, resulting in a set of geometric subdomains Ω_i of the original PDE domain Ω as described in §2.3. Let Λ^v denote the set of freedoms for the x-component of the velocity. Assume (as will usually be the case) that Λ^v is also the set of freedoms for the y-component of the velocity and also let Λ^p denote the set of freedoms for the pressure. Using the partitioning, these sets are then divided into subsets Λ_i^v and Λ_i^p. The sets Λ_i^v overlap at freedoms which lie on the boundaries of the Ω_i, as described in §2.3. However, since the pressure elements are discontinuous, the sets Λ_i^p are pairwise disjoint. We introduce the restriction operators R_i^v and R_i^p corresponding to this partitioning, as described in §2.3.

With this machinery we can define restrictions of the matrix A to the ith subdomain by

$$A_i = \begin{bmatrix} (R_i^v)^T & & \\ & (R_i^v)^T & \\ & & (R_i^p)^T \end{bmatrix} \begin{bmatrix} F_{11} & F_{12} & B_1^T \\ F_{21} & F_{22} & B_2^T \\ B_1 & B_2 & 0 \end{bmatrix} \begin{bmatrix} R_i^v & & \\ & R_i^v & \\ & & R_i^p \end{bmatrix}. \qquad (16)$$

The (one level) Parallel Whole Preconditioner for the matrix A (14) is then defined by inserting (16) in (4) and setting $R_i = \mathrm{diag}\{R_i^v, R_i^v, R_i^p\}$. This is the preconditioner which is labelled Whole in the results section below. At its present stage of development this is just a one level method: Each application of the preconditioner requires the solution of the full Navier-stokes equations on each subdomain. We are presently investigating the design of coarsening procedures to turn this into a two level method. Nevertheless, as we will see in the numerical examples this simple one-level method is very competitive with the two-level Parallel Block Preconditioner in the case of Q_2P_{-1} elements.

We would like to add here that important alternative approximations for the inverse of the Schur complement X have been derived by Green's function arguments in [15]. In principle these involve discretisation of a second-order differential operator on the underlying pressure space and perform very much better than X_B^{-1} defined in (15) on test problems with continuous pressure elements. A practical parallel version of this approximation was used in [9, 10], for discretisations based on Q_2Q_1 elements. As far as we are aware, a robust extension of the approach of [15] to general discontinuous pressure elements is not yet known.

4 Numerical examples

We have made some numerical experiments to compare the Block preconditioner with the Whole preconditioner, to measure the parallel performance of the given methods and to study two different strategies for computing eigenvalues for the two computational examples of the flow around a cylinder and the flow in an expanding pipe - examples (i) and (ii) given in Section 3.

4.1 Linear system solves

In the first series of experiments we were comparing the performance of the Block preconditioner with the Whole preconditioner with different numbers of degrees of freedom and different Reynolds numbers for the cylinder and pipe problems. One key point about these two problems (K.A. Cliffe, Private Communication) is that while the cylinder problem produces physically reasonable results for a relatively small number of degrees of freedom, the pipe problem requires many more freedoms due to the more complicated nature of the flow: any less than about 100K freedoms is fairly meaningless. Thus we present results for increasing problem sizes for the cylinder problem, but give only a single problem size for the pipe.

In Table 1 we give timings for a single solve of (13) (where the flow field $\mathbf{w}$, which enters the matrix F in (14), is taken to be the steady state solution of the Navier-Stokes equations, which was pre-computed). Thus the timings give an idea of the cost of a single Newton iteration for the Navier-Stokes system near to convergence and also of the cost of a single inverse iteration in an eigenvalue solver for the Navier-Stokes operator linearised about the steady-state. Our stopping criterion is that the initial residual norm should be reduced by 10^{-6}. In Table 1 we also present the corresponding numbers of calls to the preconditioner for each run (given in brackets). The number of slave nodes was 12 in these experiments. We were using the FGMRES method (see §2.1), preconditioned with the left-precondtioned GMRES method, where the preconditioner was either Block or Whole. The maximal number of inner iterations was 100 for the cylinder problem and 200 for the pipe problem.

Table 1: *Comparing Block and Whole preconditioners on the flow past a cylinder and the expanding pipe problem. Wall clock time (secs.) for a single solve is given followed by the number of calls to the corresponding preconditioner, in brackets.*

time (#it.)	Block			Whole		
#Dof \ Re	1.0	5.0	25.0	1.0	5.0	25.0
Flow around a cylinder						
8190	34.3 (394)	34.9 (400)	41.8 (490)	6.98 (78)	7.44 (81)	18.7 (253)
18622	108.9 (886)	94.9 (800)	114.4 (900)	22.7 (196)	19.5 (166)	42.8 (388)
33278	195.3 (1400)	170.0 (1200)	154.2 (1100)	33.3 (293)	26.0 (207)	39.6 (380)
52158	258.3 (1500)	259.3 (1500)	227.4 (1300)	69.3 (600)	54.7 (458)	73.0 (653)
75262	363.4 (1700)	403.2 (1900)	342.3 (1600)	113.7 (872)	79.1 (560)	98.0 (700)
102590	577.1 (2200)	926.1 (3600)	501.1 (1900)	168.7 (110)	113.0 (676)	128 (800)
Flow in an expanding pipe						
96400	(>160000)	(>80000)	31265.5 (100600)	1022.6 (5124)	967.1 (4600)	665.3 (3053)

As we can see, the Whole preconditioner performs much better compared to the Block preconditioner in all cases, with up to an order of magnitude improvement in solution time for the cylinder problem, and a much greater improvement in the case of the expanding pipe problem, where the Block preconditioner can fail. We expect that with an addition of an appropriate coarse grid operator, the Whole preconditioner should perform even better than this.

We also notice that, for the cylinder problem (and for both preconditioners), although there is some increase in the number of iterations as the mesh is refined, the actual solution times show close to linear growth in the number of degrees of freedom of the problem. (This may be due to the fact that the parallel efficiency increases as the number of freedoms grows.)

Finally, we also observe that in the pipe problem the number of iterations reduces when the Reynolds number is increased, at least for the range of Reynolds numbers tested. In the case of the cylinder problem there is also a reduction when going from $Re = 1.0$ to $Re = 5.0$, but after that an increase. We expect eventually an increase in the number of iterations will occur in the pipe problem too, but we may postulate that the increase is slower to appear in the pipe problem because the critical value of Re (where the steady flow becomes unstable) is expected to be much higher for the pipe problem than for the cylinder. (It is about $Re = 24$ for the cylinder.)

Table 2: *Timings and relative speedups for the flow past a cylinder and the expanding pipe problem.*

	Flow around a Cylinder			Flow in an expanding pipe		
	Block prec., 52158 DoF			Whole prec., 96400 DoF		
#Slaves	#it	time(s)	rel.s/o	#it.	time(s)	rel.s/o
1	1100	1507.0	-	3200	7237.7	–
2	1000	712.8	2.11	3400	4051.0	1.79
4	1100	379.2	3.97	3400	2291.3	3.16
8	900	193.6	7.78	2800	761.7	9.50
12	1000	174.1	8.66	4411	946.3	7.65
16	1000	175.5	8.59	3600	728.8	9.93
20	900	147.8	10.20	3800	774.5	9.35

In Table 2 we illustrate the parallel performance by listing the wall clock time for a single solve for both example problems. The results are presented in Table 2 where we give the number of iterations, followed by time in seconds and relative speedups compared with the single slave case. As we can see, the growth in the relative speedup varies somewhat due to the fact that the number of iterations varies with the number of slaves. Good speed-ups are obtained up to 8 slaves, with some decrease after that. It is expected that with larger problem sizes the speed-up will improve for higher numbers of slaves. Note that the timings include the startup overhead, including reading and distributing the input data by the master processor and the subproblem factorisations with the multifrontal solver. Again, better speedups may be obtained for a method where the start-up did not contain such a large proportion of the task, such as, for example a whole Newton process for the Navier-Stokes equations.

4.2 Eigenvalue solves

In this final section, we indicate briefly the application of the solvers which we have developed to our target application, namely the assessment of stability of steady state solutions $\mathbf{w}$ of (7) by solving the eigenvalue problem (9). The discretisation of this with mixed finite elements is of the form:

$$A\mathbf{x} = \lambda M\mathbf{x} \tag{17}$$

where the mass matrix M has the following form:

$$M = \begin{bmatrix} M_u & 0 & 0 \\ 0 & M_u & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

and M_u is the mass matrix arising from the velocity finite element space. This application is discussed in some detail in [9, 10] and the references given there, where it is explained that the core of a typical iterative method (based on Arnoldi's method, inverse iteration or related methods) is the solution of shifted systems

$$(A - \sigma M)\mathbf{y} = \mathbf{x} \ , \tag{18}$$

where the shift σ lies near a part of the spectrum where the critical eigenvalues are thought to lie.

We consider two algorithms, both of which use PARPACK – the parallel version of the ARPACK package – which is based on the Implicitly Restarted Arnoldi Method [16] and is suitable for large scale eigenvalue problems. The software provides the option to compute a few (k) eigenvalues with user specified features of a general $n \times n$ matrix. We call PARPACK in the "shift-invert" mode and with the computational mode 'LI' – as we know from some preliminary investigations that the desired eigenvalues have large imaginary part. PARPACK calls for many solutions of (18), and for these we are using the `DOUG` solver described above. The preconditioner for A is used throughout as the preconditioner for (18), independent of the value of σ.

Our first strategy is the straightforward application of PARPACK. We input a shift σ, ($\sigma = 0$ in this case) a required relative eigenvalue tolerance, *etol* and a relative tolerance, *tol* for the shifted systems (18), which we take to be of the same order as *etol*.

Our second strategy is to use PARPACK to obtain a coarse approximation to the desired eigenvalue and eigenvector, and then feed these approximations to an inexact inverse iteration method, with shift converging to the desired approximate eigenvalue. So our second strategy is PARPACK with refinement by inverse iteration with a fine relative tolerance for the eigenvalue approximation. We update the shift using the Rayleigh Quotient of the current approximation of the eigenvector, but keep the solve tolerance *tol* fixed during the inverse iteration phase of the algorithm. Note that since the shift is complex, we need to use the complex version of the FGMRES method. However, as all the matrices are real, we can avoid complex arithmetic during the preconditioning step by calling the same preconditioner twice – once with the real part and secondly with the imaginary part of the right-hand side vector.

In Table 3 we compare the two strategies for computing eigenvalues. Here $k = 20$, and we were using the Whole preconditioner. For the first strategy we took $tol = etol = 10^{-6}$. For the second strategy, we chose $tol = 0.01$, $etol = 0.05$ during the PARPACK phase and $tol = 0.01$ and $etol = 10^{-6}$ in the inverse iteration phase. As we can see, for this problem the combined strategy turns out to be far more efficient.

Up till now software for solving the eigenvalue problem for the Navier-Stokes equations has been based on iterative eigenvalue algorithms with exact solves for the shifted systems (18). This is unfeasible for large systems, such as those arising from the expanding pipe problem.

Table 3: *Computing eigenvalues with PARPACK and the combined method of refining rough PARPACK eigenvalues by the inverse iteration method.*

Strategy:	1. PARPACK	2. PARPACK combined with inverse iteration		
#DoF	Total time	PARPACK time	Inverse it. time	Total time
Flow past a cylinder				
33278	2270.6	471.1	53.4	524.5
52158	3732.4	714.9	260.6	975.5
75262	5282.6	1114.7	261.0	1375.7
Epanding pipe problem				
96400	71952.0	13566.8	2773.6	16340.4

The main aim of this work is to produce iterative methods for (18) which are powerful enough to be used in computing eigenvalues for difficult problems such as this.

References

[1] P. E. Bjørstad, Maksimillian Dryja and Eero Vainikko, Robust Additive Schwarz Methods on Unstructured Grids, Domain Decomposition in Science and Engineering, Proceedings from the 9th International Conference on Domain Decomposition Metods, Bergen, John Wiley & Sons, 1997, pp. 597 - 605.

[2] T. F. Chan, B. Smith and J. Zou, Overlapping Schwarz methods on unstructured meshes using non-matching coarse grids, Numer. Math. **73** (1996), pp. 149 - 167.

[3] K. A. Cliffe, I.G. Graham, R. Scheichl and L. Stals, Parallel computation of flow in heterogeneous media modelled by mixed finite elements, Journal of Computational Physics **164** (2000), pp. 258 -282.

[4] K. A. Cliffe, A. Spence and S. J. Tavener, The numerical analysis of bifurcation problems with application to fluid mechanics, Acta Numerica (2000), pp 39 – 131.

[5] T. A. Davis and I. S. Duff, "A combined unifrontal/multifrontal method for unsymmetric sparse matrices", ACM Transactions on Mathematical Software, vol. 25, no. 1, pp 1-19, 1999.

[6] H. C. Elman, Preconditioning for the steady-state Navier-Stokes equations with low viscosity, SIAM J. Sci. Comp. **20** (1999), pp. 1299-1316.

[7] H. C. Elman and D. J. Silvester, Fast nonsymmetric iterations and preconditioning for Navier-Stokes equations, SIAM J. Sci. Comp. **17** (1996), pp. 33-46.

[8] I.G. Graham, M.J. Hagger, L. Stals and E. Vainikko, The `DOUG` package, `http://www.maths.bath.ac.uk/~parsoft/doug`.

[9] I.G. Graham, A. Spence and E. Vainikko, Parallel Iterative Methods for Navier-Stokes Equations and Application to Stability Assessment, in EupoPar2002 Parallel Processing, B. Monien and R. Feldmann (Eds.), Lecture Notes in Computer Science 2400, Springer-Verlag, Berlin, 2002.

[10] I.G. Graham, A. Spence and E. Vainikko, Parallel Iterative Methods for Navier-Stokes Equations and Application to Eigenvalue Calculation, submitted to Concurrency and Computation: Practice and Experience, 2002. (`http://www.maths.bath.ac.uk/~igg/publications.html`)

[11] P. M. Gresho and R. L. Sani, Incompressible Flow and the Finite Element Method. Advection-diffusion and isothermal laminer flow. John Wiley & Sons, 1998.

[12] M. J. Hagger, Automatic domain decomposition on unstructured grids DOUG, Advances in Computational Mathematics **9** (1998), pp. 281 - 310.

[13] G. Karypis and V. Kumar, METIS: Unstructured graph partitioning and sparse matrix ordering system, Department of Computer Science, University of Minnesota Report, Minneapolis, Aug. 1995.

[14] D. Kay and D. Loghin, A Green's function preconditioner for the steady-state Navier-Stokes equations. Oxford Numerical Analysis Group Research Report NA-99/06, 1999.

[15] D. Kay, D. Loghin and A. Wathen. A preconditioner for the steady-state Navier-Stokes equations. SIAM J. Sci. Computing, to appear.

[16] R. B. Lehoucq, D. C. Sorensen, C. Yang ARPACK user's guide: Solution of large scale eigenvalue problems with implicitly restarted Arnoldi methods. http://www.caam.rice.edu/software/ARPACK [12 December 2002].

[17] Z. Li, Y. Saad and M. Sosonkina. pARMS: a parallel version of the algebraic recursive multilevel solver. Report umsi-2001-100. Minnesota Supercomputer Institute. University of Minnesota, Minneapolis, MN, 2001.

[18] J. W. Ruge and K. Stüben, "Algebraic Multigrid", Multigrid Methods, Frontiers in Applied Math., pp. 73– 130, SIAM, 1987.

[19] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. SIAM J. Sci. Comput., **14** (1993), pp. 461 - 469.

[20] Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM J. Sci. Stat. Comput., **7** (1986), pp. 856 - 869.

[21] D.Silvester, H.Elman, D.Kay, A.Wathen, Efficient preconditioning of the linearized Navier-Stokes equations for incompressible flow, J. Com. Appl. Math. **128** (2001), pp. 261-279.

[22] B.F.Smith, P.E.Bjørstad and W.D.Gropp, Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations. Cambridge University Press, Cambridge, 1996.