Design of a parallel hybrid direct/iterative solver for CFD problems

Thies, Jonas; Wubs, Fred

Published in:
EPRINTS-BOOK-TITLE

Publication date:
2011

2011 Seventh IEEE International Conference on eScience

Design of a parallel hybrid direct/iterative solver for CFD problems

Jonas Thies
Centre for Interdisciplinary Mathematics
Uppsala University
Sweden
Email: jonas@math.uu.se

Fred Wubs
Johann Bernoulli Institute for
Mathematics and Computing Science
University of Groningen
The Netherlands
Email: f.w.wubs@rug.nl

***Abstract*—We discuss the parallel implementation of a hybrid direct/iterative solver for a special class of saddle point matrices arising from the discretization of the steady Navier-Stokes equations on an Arakawa C-grid, the $\mathscr{F}$-matrices.**

The two-level method described here has the following properties: (i) it is very robust, even hat comparatively high Reynolds Numbers; (ii) a single parameter controls fill and convergence, making the method straightforward to use; (iii) the convergence rate is independent of the number of unknowns; (iv) it can be implemented on distributed memory machines in a natural way; (v) the matrix on the second level has the same structure and numerical properties as the original problem, so the method can be applied recursively. The implementation focusses on generality, modularity, code reuse and recursiveness. The solver is implemented using building blocks of the Trilinos libraries. We show its performance on a parallel computer for the Navier-Stokes equations.

I. Introduction

Presently, a typical computational fluid dynamics (CFD) problem may involve millions of unknowns, representing velocities and pressures on a grid, which have to be determined by linearizing the discrete equations and solving a large sparse linear system of equations. Robust numerical methods are needed to achieve high fidelity. Therefore one often resorts to direct (sparse) solvers. In general such a method does not fail as long as the used precision is enough to handle the posedness of the problem. However, there are several disadvantages to direct methods. Both the amount of memory and computing time increase quickly as the problems become larger. Direct methods are also inherently difficult to parallelize, as compared to iterative approaches.

The computational cost of direct methods for 3D partial differential equations (PDEs) grows with the square of the number of unknowns. Iterative methods are therefore preferred for very large applications. They perform a finite number of iterations to yield an approximate solution; in theory the accuracy achieved increases with the number of iterations performed. However, iterative methods are often not robust for difficult problems arising from the discretization of mixed parabolic/hyperbolic PDEs. The iteration process may stall or diverge, and the final approximation may be inaccurate. Furthermore they often require custom numerics such as preconditioning techniques to be efficient.

The hybrid direct/iterative approach presented here seeks to combine the robustness of direct solvers with the memory and computational efficiency of iterative methods. We perform a non-overlapping domain decomposition of the grid, and eliminate the interior velocities using a direct method. For the remaining variables a Schur complement problem has to be solved, which we do by a Krylov subspace method preconditioned by a novel incomplete factorization preconditioner.

We give a brief overview of the algorithm in Section II. Section III contains a description of the parallel implementation. At the end of the paper, we present numerical experiments concerning the convergence of the new method. As the implementation presented here is parallel and orders of magnitude faster than the MATLAB code used in [1], we can now study the method in much more detail (cf. Section IV-A). We also investigate the parallel performance and indicate the bottlenecks of the present implementation (Sections IV-C and IV-D). In Section V we compare the results to those found in literature for other methods applied to similar problems, and we conclude by proposing some improvements in Section VI.

II. Algorithm

We consider the problem of solving the equations

$$Kx = b, \tag{1}$$

where $K \in R^{(n+m)\times(n+m)}$ $(n \geq m)$ is a saddle point matrix that has the form

$$K = \begin{bmatrix} A & B \\ B^T & 0 \end{bmatrix}, \tag{2}$$

with $A \in R^{n\times n}$, $B \in R^{n\times m}$. For the Stokes problem discretized on a C-grid (Fig. 1), K is a so-called $\mathscr{F}$-matrix (A is symmetric positive definite and B has row sum 0 and at most two entries per row [2]). Recently, a direct method for the solution of $\mathscr{F}$-matrices was proposed [3]. It reduces fill and computation time while preserving the structure of the equations during the elimination. A hybrid direct/iterative method based on this approach was presented in [4], [1]. It has the advantage that the ordering it defines for the matrix exposes parallelism on each level: all the subdomain matrices can be factored independently using sequential sparse direct solvers, and the Schur complement can be constructed with a minimal amount of communication in an assembly process. The difficult task

978-0-7695-4597-4/11 $26.00 © 2011 IEEE
DOI 10.1109/eScience.2011.60

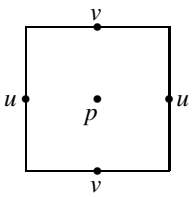

Figure 1. Positioning of velocity (u,v) and pressure (p) in the C-grid.

of parallel preconditioning is aided by the algorithm, which yields a block-diagonal preconditioner with dense blocks and a significantly reduced sparse linear system. In this paper we focus on the aspect of parallelism and present an implementation of the solver intended for distributed memory machines. A number of design choices is made, and we illuminate some of the issues that arise using a structured C-grid Navier-Stokes code as an example.

The ingredients of the algorithm are the following

1) Perform a non-overlapping domain decomposition
2) Detect the velocity separators
3) Pick for every subdomain a P node to be kept in the Schur complement
4) Eliminate all interior variables of the subdomains and construct the Schur complement for the velocity separators and the selected pressures.
5) Perform a Householder transformation on each separator. This decouples most of the V-nodes from pressure nodes.
6) Identify V_Σ nodes (separator velocities that still connect to two pressures).
7) Drop all connections between non-V_Σ nodes and V_Σ nodes, and between non-V_Σ nodes in different separator groups. The resulting matrix is block-diagonal with the 'reduced Schur complement' defined by the V_Σ and P-nodes.
8) Iterate on the Schur complement using the matrix of the previous step as preconditioner. This preconditioner is easily applied using LU decompositions of all non-V_Σ blocks and the reduced system.

A. Computational complexity

We assume that a direct method with optimal complexity is used for the solution of the relevant linear systems, so in 3D if the number of unknowns is $\mathcal{O}(N)$, the work is $\mathcal{O}(N^2)$, as with Nested Dissection. We have $N = \mathcal{O}(n^3)$ unknowns, where n is the number of grid cells in one direction. We keep the subdomain size constant and denote the number of unknowns per subdomain by $S = s^3$ (s is the separator length), so separator groups have $\mathcal{O}(s^2)$ velocities. Per subdomain there will therefore be $\mathcal{O}(s^2)$ non-V_Σ- and $\mathcal{O}(1)$ V_Σ nodes. The amount of work required per subdomain is as follows:

1) $\mathcal{O}(S^2)$ for the subdomain elimination;
2) transformation on faces with H: $\mathcal{O}(s^4)$;
3) factorization of non-V_Σ nodes: $\mathcal{O}((s^2)^3) = \mathcal{O}(S^2)$.

The total over all domains is $\mathcal{O}(N/S)\mathcal{O}(S^2) = \mathcal{O}(NS)$, so in this part the number of operations increases linearly with S (e.g. by a factor 8 if s is doubled).

The solution of the reduced problem (V_Σ nodes) requires $\mathcal{O}((N/S)^2)$ operations. Here doubling s will decrease the work by a factor 64. So in total the work per iteration is

$$\mathcal{O}(NS) + \mathcal{O}((N/S)^2).$$

The number of iterations is independent of N for constant S, and proportional to $\log(1+S)$ for constant N. So if we double s, a fixed number of iterations is added.

If we can solve the reduced problem iteratively by applying our method recursively until the problem has a fixed grid-independent size, the cost per iteration becomes $\mathcal{O}(NS\log(1+S))$ and thus linear in the number of unknowns N, but we focus on the two-level case up to now.

III. Parallel implementation

A. Basic design of the solver

The Trilinos package `Ifpack` suggests subdividing the process of solving a linear system of equations into three distinct phases: *Initialization*, *Computation* and *Solution*. The initialization step can be done as soon as the sparsity pattern of the matrix is known, it may involve memory allocation or computing a suitable ordering for a factorization. The compute phase is executed once the numerical values of the matrix entries are known and typically consists of an (incomplete) factorization of the matrix. The solve phase is executed whenever a right-hand side is provided and returns the solution vector. We will adopt this structure as it can be exploited during a simulation. For the two-level hybrid solver we identify these phases as:

1) **Initialization phase:** compute the ordering.
   - Partition the variables into subdomains,
   - detect and group separator variables,
   - initialize the direct solvers for interior variables (subdomain solvers).
2) **Compute phase.**
   - Compute subdomain solvers,
   - construct the Schur complement S on the separators,
   - construct a preconditioner for S.
3) **Solve phase.**
   - Solve for the interior variables using the subdomain solvers,
   - solve for the separator variables using a preconditioned Krylov subspace method.

Apart from the decision to divide the program into three stages, we have the following requirements on the implementation:

- generality - the code should be usable for a range of problems such as the Poisson equation, Darcy's law or (Navier-)Stokes, and possibly other applications such as the ocean equations studied in [5]. To achieve this we implement a number of standard approaches for different variable types, as discussed in Section III-B.
- Modularity - parts of the algorithm (e.g. the partitioning method or the direct solver) should be easy to replace.

- Code reuse - we want to make use of existing code as much as possible. The Trilinos libraries offer many versatile classes that we exploit in our implementation. That way, parts of the code that are not too specialized are 'outsourced' to a software library that is well-maintained and documented.
- Recursiveness - once a multi-level algorithm has been designed it should be easy to extend the existing implementation.

It goes without saying that efficiency and parallel performance are also among our primary concerns. We will now proceed to look at the input data and the three phases of solving a linear system using our algorithm and discuss the important implementation issues.

B. Input format and variable types

The implementation should be as generally applicable as possible, so we do not decide on a specific CFD code, discretization or even PDE a priori. Instead we want to write a solver that takes a matrix and some information on the structure of the problem and returns a solution whenever we pass in a right-hand side vector. We will only assume that the problem is defined on some kind of grid, and that each grid-cell has the same number of unknowns. The unknowns are ordered per grid-cell, e.g. for a 2D Stokes problem we may have an ordering $(u_1, v_1, p_1, u_2, v_2, p_2, ...)$, where the subscript is the cell index. Such a data layout can be obtained from an existing code by introducing 'dummy equations' at boundaries (to make sure that every grid cell has the same number of variables) and reordering the matrix. This initial ordering assigns a unique global index (*GID*) to each variable, and we will make sure that the GID is preserved everywhere in the code so that it can be used to identify a variable's type or the subdomain to which it belongs. We also require information about the variable types in the problem, e.g. variable 3 in each grid cell is a pressure here. This information is used to define the ordering during the initialization phase. Presently we support three types of variables:

- "Laplace" - the variable connects to variables of the same type in neighboring grid cells and a partitioner will be used to define the subdomains. Separators for this variable type will be identified and retained in the next Schur complement. We assume that at least one of the variables in the problem is of this type so that we can base the partitioning on this variable. For the Navier-Stokes equations, the velocities u and v are "Laplace" variables.
- "Uncoupled" - the variable is partitioned along with the other variables, but no separators are needed. These variables are all eliminated as subdomain interior variables.
- "Retain X" - these variables are basically uncoupled, yet one has to retain at least X of them in the interior of each subdomain. For our prototype problem, the pressure has type "Retain 1" so that the subdomain matrices stay non-singular.

240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	[12–15]
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	[8–11]
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	[4–7]
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	[0–3]
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

Figure 2. Simple cartesian partitioning of a Laplace problem on a structured 16×16 grid. There are four partitions and 16 subdomains, four per partition.

Using this input format it is very easy to adapt the solver to different equations, for instance one may add a tracer equation to the Navier-Stokes equations by increasing the number of degrees of freedom (dof) per cell by 1 and setting the variable type for the tracer to "Laplace". Obviously, the CFD code used must be adjusted to provide the correct Jacobian and residual function.

C. Initialization phase

In this phase we need to partition the variables into subdomains and define an ordering such that we can consistently access different classes of variables (i.e. interior variables, separator variables). Finally we can construct and initialize objects to hold the LU-decomposition of the interior variables. This is done in three phases to ensure modularity.

1) Partitioning: We first perform a coarse-grained partitioning for parallelism, giving np partitions, where np is the number of tasks in a parallel computation. Each task can then partition its owned variables into a number of subdomains, such that each variable in the global matrix belongs to exactly one subdomain on one partition (i.e. in the memory of one task).

Both the coarse-grained partitioning and the local decomposition into subdomains can be done either based on the graph of the matrix (using graph partitioning algorithms like the ones implemented in METIS [6]), or geometrically. In this paper we focus on structured grid problems and geometric partitioning into rectangular subdomains (cartesian partitioning).

For a scalar 2D problem (like the Laplace equation discretized using a 5-point stencil), the initial partitioning may look like the one shown in Figure 2, with partitions divided by thick lines and subdomains by thin or thick lines. Each number indicates the GID of the variable at that position in the grid.

We provide a virtual interface class `BasePartitioner` that can be used to obtain a unique subdomain ID for each global variable index (GID). A concrete implementation is

I_8	I_8	I_8	${}_{8}S_{9}$	I_9	I_9	I_9	${}_{9}S_{10}$	I_{10}	I_{10}	I_{10}	I_{10}
I_8	I_8	I_8	${}_{8}S_{9}$	I_9	I_9	I_9	${}_{9}S_{10}$	I_{10}	I_{10}	I_{10}	I_{10}
I_8	I_8	I_8	${}_{8}S_{9}$	I_9	I_9	I_9	${}_{9}S_{10}$	I_{10}	I_{10}	I_{10}	I_{10}
I_8	I_8	I_8	${}_{8}S_{9}$	I_9	I_9	I_9	${}_{9}S_{10}$	I_{10}	I_{10}	I_{10}	I_{10}
${}^{8}_{4}S$	${}^{8}_{4}S$	${}^{8}_{4}S$	${}^{8}_{4}S^{9}_{5}$	${}^{9}_{5}S$	${}^{9}_{5}S$	${}^{9}_{5}S$	${}^{9}_{5}S^{10}_{6}$	${}^{10}_{6}S$	${}^{10}_{6}S$	${}^{10}_{6}S$	${}^{10}_{6}S$
I_4	I_4	I_4	${}_{4}S_{5}$	I_5	I_5	I_5	${}_{5}S_{6}$	I_6	I_6	I_6	I_6
I_4	I_4	I_4	${}_{4}S_{5}$	I_5	I_5	I_5	${}_{5}S_{6}$	I_6	I_6	I_6	I_6
I_4	I_4	I_4	${}_{4}S_{5}$	I_5	I_5	I_5	${}_{5}S_{6}$	I_6	I_6	I_6	I_6
${}^{4}_{0}S$	${}^{4}_{0}S$	${}^{4}_{0}S$	${}^{4}_{0}S^{5}_{1}$	${}^{5}_{1}S$	${}^{5}_{1}S$	${}^{5}_{1}S$	${}^{5}_{1}S^{6}_{2}$	${}^{6}_{2}S$	${}^{6}_{2}S$	${}^{6}_{2}S$	${}^{6}_{2}S$
I_0	I_0	I_0	${}_{0}S_{1}$	I_1	I_1	I_1	${}_{1}S_{2}$	I_2	I_2	I_2	I_2
I_0	I_0	I_0	${}_{0}S_{1}$	I_1	I_1	I_1	${}_{1}S_{2}$	I_2	I_2	I_2	I_2
I_0	I_0	I_0	${}_{0}S_{1}$	I_1	I_1	I_1	${}_{1}S_{2}$	I_2	I_2	I_2	I_2

Figure 3. Interior nodes (I) and separators (S) for the 2D example problem. Nodes with the same indices belong to a group.

the `CartesianPartitioner` that only considers information on the problem (such as grid-size, dimension, dof/cell) and provides the partitioning shown in Fig. 2. For problems with several dof per cell, the same partition number is assigned to each variable in a grid cell.

For ease of notation, let us introduce a number of short-hands. If subdomain s belongs to partition p, we write $s \in p$. Likewise, we write $i \in s$ if variable i has been assigned to subdomain s. Both partitions and subdomains have unique non-negative integer ID's, so that relational operators like $s < t$ are meaningful if s and t are subdomain ID's. If matrix row i has a nonzero entry in column j, we say that variable i couples to variable j.

2) Detecting and grouping separators: A variable $i \in s$ is called *interior* if it couples only to variables $j \in t$ where $t <= s$. Any interior variable is associated with exactly one subdomain s. A variable i is a *separator* variable of s if it couples to variables in s and at least one subdomain $t > s$. This includes variables $i \notin s$, so that separator variables are associated with two or more subdomains. Separator variables are grouped according to variable (u, v etc.) and the subdomains they connect to.

Caveat: In the case of a 5-point stencil in 2D (7-point in 3D), the variables in the corners of a subdomain are not identified as separator nodes of the diagonally displaced subdomain. In our example, node 27 becomes a separator of subdomain 5 because it connects to nodes 28 and 39, which are separator nodes in the first place. To avoid this complication, we perform the detection of separators for using an auxiliary graph (a 9- (27-)point stencil in 2D (3D)). . An alternative that works for more general problems is the hierarchical interface decomposition used in [7], [8] for scalar problems.

Figure 3 shows how variables are grouped into interior and separator groups. Each separator group is shared by two or more subdomains. The detection and grouping of separators can be based on any non-overlapping partitioning provided by a `BasePartitioner` object. To aid the implementation of the solver, we introduce another base class here, the `BaseOverlappingPartitioner`, that allows accessing all variables of a given local subdomain group-wise, i.e. all interior variables or all variables forming a separator of that subdomain. By 'accessing' we mean retrieving the GID of a variable, which can then be used to find the entry in a matrix or vector object. Our concrete implementation uses the detection and grouping algorithm described above, with a minimal overlap between partitions so that it can work in parallel.

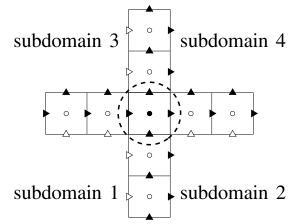

Figure 4. The marked cell where the four separators meet is called a full conservation cell: all its velocities are part of separators, so the pressure has to be retained in the Schur complement to avoid a singular matrix for the interior of subdomain 1.

3) Domain decomposition for Stokes problems: For the Stokes problem on a C-grid, each cell has two (three) velocity components in 2D (3D), located at the cell edges (faces), and a pressure (located in the cell center). Here A in (2) is a Laplace operator for each velocity component. As we do not make approximations in the B-part, grid-independent convergence is achieved if we use the scalar partitioning discussed above for each of the velocity components. The grouping of variables only has to make sure that each separator group has variables of only one type (e.g. u-velocities or v-velocities). This holds even if there are couplings due to convection in the Navier-Stokes Jacobian matrix.

All pressures will be put in the 'interior' group of their subdomain except for one in each subdomain, which forms its own group (we arbitrarily choose the first in each subdomain). Next, we identify any interior pressures that do not connect to any interior velocity by searching the original graph of the problem. Such a pressure belongs to a so-called full conservation cell (cf. Figure 4) and leads to a singular subdomain matrix unless we move it into a group of its own. Any velocities it connects to are also moved to groups of their own. This makes sure that the reduced Schur complement has the desired form of two dense pressure-columns with opposite sign per group. To keep this step independent of the type of problem solved, we introduce a tag "Retain Isolated", which we use to identify the variables for which this special treatment is required. The grouping of separator variables is done per variable type. For the 2D Stokes-equations on the 16×16 grid, for example, the first separator with grid cells $[4, 16, 28]$ leads to two separator groups, one for u and one for v in those cells.

D. Compute phase

Once these data structures are available, we can proceed to eliminate the interior variables. Each interior variable belongs

to exactly one subdomain and each subdomain to one task in a parallel computation. We can therefore use a sequential sparse direct solver for this task. Any of the sparse solvers interfaced by the `Amesos` package can be used for this purpose, which ensures modularity in this part. We note, however, that the choice is not critical as the subdomain matrices are typically small.

1) Constructing the Schur complement: We now have a partial LU-decomposition of our global matrix K: all internal subdomain variables have been eliminated. The remaining unknowns are the separators and - for Stokes problems - retained pressures. The Schur complement defined by this set of unknowns is given by the expression

$$\mathbf{S}_{ij} = K_{ij} - \sum_k K_{ik}(K_k)^{-1}K_{kj}, \tag{3}$$

where i and j indicate the sets of separator groups , and k indicates the sets of interior variables (K_k denoting the diagonal block associated with the interior variables of subdomain k). The sum is sparse because each separator has contributions from at most eight subdomains (in 3D). The sum is implemented in a blocked fashion: We loop over all overlapping subdomains (k-index) and perform an LU-solve $B_{kj} = K_k^{-1}K_{kj}$) and a matrix/matrix product, $S'_{ij} = K_{ik}B_{kj}$. The resulting matrices S' are subtracted from K_{ij} to form the Schur complement.

When using $\mathbf{S}$ in a Krylov sequence we do not have to construct it explicitly but could use the defining equation 3, i.e. matrix-vector products with K_{ij}, K_{ik} and K_{kj} and a sparse matrix solve with K_k whenever applying S to a vector. This makes the method somewhat cheaper during setup and both more expensive and more scalable during the solve phase. In our experiments constructing S was the better option, but a break-even point may be reached when using many processors, where inter-processor communication is the bottleneck rather than the floating-point operations. Both approaches are implemented.

2) Construction of the preconditioner : The preconditioner is constructed by applying an orthogonal transformation from the left and right to each separator group in the Schur complement, and then applying a dropping strategy where all connections between V_Σ and non-V_Σ nodes and between non-V_Σ nodes in different separator groups are dropped (the first element in each separator group is a V_Σ-node). Our program automatically treats pressure variables for Stokes-type problems correctly as each pressure is in a separator group of size 1 so that the corresponding orthogonal transformation is the identity operator. The present implementation is as follows, assuming that the Schur complement has been completely constructed:

- the orthogonal transformation is explicitly constructed as a block-diagonal sparse matrix representing the vector $w = v/\sqrt{v^T v}$ in the Householder transformation $I - 2vv^T/v^T v$,
- the transformation is applied using a series of sparse matrix-matrix products and additions to compute $(I-2w^T w)A(I-2w^T w)$,
- the dropping strategy is implemented by extracting the remaining blocks from the transformed matrix. The blocks corresponding to non-V_Σ nodes within a group are small and dense and solved using LAPACK. The block corresponding to V_Σ nodes (the reduced Schur complement) is sparse and distributed among all tasks. It can either be solved sequentially or in parallel using a sparse direct solver from the `Amesos` library.

This implementation is simple and performs fairly well in practice, although improvements can certainly be made.

E. Solve phase

We now formally have a partial LDU-decomposition of the system matrix K:

$$\begin{bmatrix} K_{II} & K_{IS} \\ K_{SI} & K_{SS} \end{bmatrix} = \begin{bmatrix} I_I & 0 \\ K_{SI}K_{II}^{-1} & I \end{bmatrix} \begin{bmatrix} K_{II} & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} I_I & K_{II}^{-1}K_{IS} \\ 0 & I_S \end{bmatrix} \tag{4}$$

where an index I denotes interior variables and S the remaining separator variables. From the block diagonal matrix K_{II} we construct an LU factorization $L_I U_I$ where also L_I and U_I are block diagonal triangular matrices so that linear systems with K_{II} are easily solved in parallel. We use the LDU- rather than an LU formulation because it doesn't require access to the individual factors L_I and U_I. That way the implementation is independent of the actual solver used for the interior variables, which may not even involve a factorization. A complete solve of $Ax = b$ is performed in the following steps:

- solve $y_I^{(1)} = (L_I U_I)^{-1} b_I$,
- compute $y_S = b_S - K_{SI} y_I^{(1)}$,
- solve $x_S = \mathrm{S}^{-1} y_S$ iteratively,
- compute $z_I = K_{IS} x_S$,
- solve $y_I^{(2)} = (L_I U_I)^{-1} z_I$,
- compute $x_I = y_I^{(1)} - y_I^{(2)}$.

In contrast to an LU-formulation, this requires two linear systems with the interior variables to be solved. This is acceptable since this step is fast compared to the solution of the Schur complement and is trivially parallel.

Application of the preconditioner: During the Krylov subspace method for the Schur complement S, we have to apply the inverse of the preconditioning matrix, $x_S = P^{-1} y_S$. As P is actually an approximation of the transformed matrix $H^T S H$, we also have to apply the orthogonal transforms to the vectors before and after the preconditioner. The inverse preconditioner is applied by solving a series of dense linear systems for the non-V_Σ nodes (using previously computed LU-factorizations) and a distributed sparse linear system for the V_Σ nodes. The transformations and dense solves can all be executed in parallel.

IV. Numerical experiments

In [4], [1] we have already demonstrated robustness of the new method for the driven cavity test problem at Reynolds Numbers up to 8 000 and a grid size of 512×512. In this section we will first investigate the convergence behavior of the GMRES method and then compute steady states at even

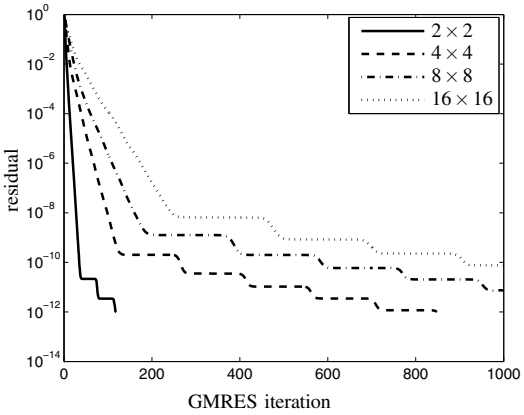

Figure 5. Convergence behavior depending on the subdomain size (driven cavity problem, 512×512 grid, $\mathrm{Re} = 2\ 000$).

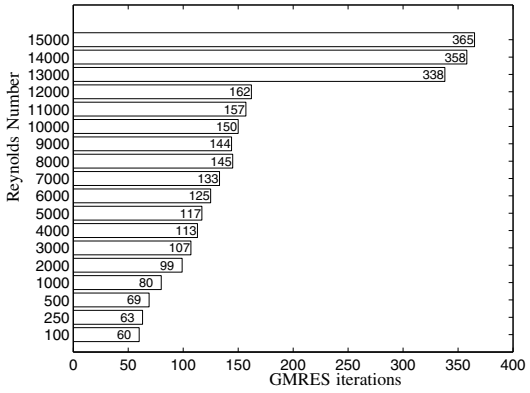

Figure 6. Number of classical GMRES iterations at high Reynolds Numbers (subdomain size 4×4, convergence tolerance 10^{-8})

higher Reynolds numbers. We then investigate the parallel performance for a 3D problem.

All experiments were performed on the Huygens machine at the computing center SARA in Amsterdam. (104 nodes with 16 dual core IBM Power6 processors running at 4.7 GHz, 128 GByte of shared main memory per node). We use MPI for all communication purposes, even inside a node where shared memory techniques could be used.

A. Convergence behavior

In [4], [1] we showed that the number of GMRES iterations is independent of the grid-size for the driven cavity test problem (in fact it decreases because the diffusion term becomes more dominant when the grid is refined). We now solve the same problem using our new implementation. We choose a fixed grid-size of 512×512 and start from the solution at $\mathrm{Re} = 1\ 000$. To compute the solution at $\mathrm{Re} = 2\ 000$ to an accuracy of 10^{-10}, 4 Newton steps are performed. The convergence tolerance of the GMRES solver for the Schur complement is set to 10^{-12} and we allow a maximum of 1 000 iterations. The implicit residual norm $||P^{-1}r||_2/||P^{-1}r_0||_2$ (P denoting the preconditioner) of the classical right-preconditioned GMRES method is shown in Figure 5 for the first Newton step. The convergence is very smooth in the beginning but at some point the residual norm starts forming 'plateaus'. For larger subdomain sizes, these plateaus occur earlier and are longer. This is another argument for not using too large subdomains - we already saw in the complexity analysis in Section II-A that for the hypothetical multi-level method the subdomain size should be chosen as small as possible. Problems may occur if we want to use a restarted GMRES method (i.e. GMRES(m)). In contrast to GMRES, global convergence of GMRES(m) is not ensured: we observed stagnation if m is chosen smaller than the length of the longest plateau that is encountered. For most experiments in this paper, we consider comparatively low Reynolds Numbers so that an accuracy of $10^{-6} - 10^{-8}$ is sufficient to resolve the steady state.

B. Robustness at high Reynolds Numbers

When increasing the Reynolds Number Re in the driven cavity problem, the solution becomes more and more convection dominated and the Jacobian more and more ill-conditioned. At $\mathrm{Re} \approx 8000$, eigenvalues start crossing the imaginary axis so that the solution becomes unstable and the Jacobian indefinite. In Figure 6 we show the number of GMRES iterations on the Schur complement in the first Newton step, depending on Re. In the last few steps the number of iterations increases sharply because the method encounters a plateau. Nevertheless, convergence is achieved within a relatively small number of iterations and does not stagnate. The simulation takes about ten minutes using 16 cores of a Huygens node.

C. Computational cost of the three phases

In order to identify the parts of the program where most time is spent, we will solve the 3D driven cavity problem, a straight-forward generalization of the problem investigated above where no-slip boundary conditions are applied at five walls of a cube of edge length 1, and a Dirichlet-condition $u = 1, v = w = 0$ at the top. We use a stretched $32 \times 32 \times 32$ grid and a subdomain size of $4 \times 4 \times 4$. We compute steady states at increasing Reynolds Numbers of $\mathrm{Re} = 50, 100, 167, 258, 380, 545$, and 767. At this parameter value, the steady state is already highly unstable in the three-dimensional case. A total of eight linear systems has to be solved. The resulting profile of the code on up to 32 cores is shown in Figure 7. Here KLU is used for the subdomain factorizations, and MUMPS [9] (version 4.9.2) for the reduced system. PT-SCOTCH [10] is used to compute a fill-reducing ordering for the reduced problem. In all cases at least 96% of the total runtime is spent on solving the linear systems with the Jacobian (the remaining tasks are collectively denoted by 'Other' here). This justifies neglecting the continuation and Newton process from now on (the relative increase of this portion as the number of cores increases is explained by the

fact that the model itself is not parallelized, i.e. the entire Jacobian and right-hand side are computed on each process). The initialization phase is cheap and seems to scale reasonably well, and as it has to be performed only once per simulation, we do not further consider it.

The compute phase clearly dominates, but its portion decreases as the number of cores is increased. This indicates high computational costs in this phase but also reasonably good scalability. On the other hand, the solve phase becomes more dominant as the number of cores is increased: This indicates a lack of scalability. We will look at the compute and solve phases in more detail in the next section to identify the weak spots of the code.

D. Parallel performance

We use the 3D test problem described above and investigate the performance for subdomain size 8 on a $64 \times 64 \times 64$ grid. Unfortunately we can not go beyond this problem size because of memory limitations: the CFD code that provides the Jacobian is not parallelized.

Table I contains the data for the 'compute' phase and the 'solve' phase. For each phase it shows the total time in the first row. The remaining rows are the subroutines that contribute significantly to that phase. The figures in these columns do not necessarily add up to 100 because the timing results are averaged over several instances of the phase. For a varying number of cores we show the wall-clock time and the parallel speed-up as compared to the first column. Note that we start off with 8 cores. Running with one core was simply not possible.

1) Compute Phase: The subdomain factorization yields super-linear speed-up in most cases (Table Ia). For the construction of the Schur complement S this is only achieved for small subdomains: larger subdomains mean that larger dense blocks have to be summed into the sparse matrix, which leads to increased communication. The compute phase is dominated by initialization and computation of the preconditioner for S.

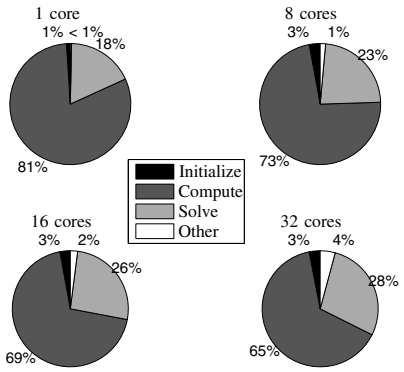

Figure 7. Profiling of a continuation run on up to 32 cores. We distinguish the three phases of the linear solver and show the percentage of the total runtime spent in each phase.

The initialization phase is not critical as it could be moved into the global initialization phase (and thus be performed only once).

The computation of the preconditioner is detailed in the indented part of Table Ia. The dominant subroutine here is the application (from the left and right) of the Householder transform to the matrix S. A better implementation of this operation could be thought of, but it seems to scale reasonably well as it is. Once the transformed matrix is constructed, the factorization of the non-V_Σ blocks scales linearly in most cases. The factorization of the reduced (V_Σ-) matrix is relatively fast but scales poorly.

2) Solve Phase: The results for the solution phase in Table Ib are analogous to those for the compute phase: the subdomain solve scales super-linearly in most cases, and most time is spent in applying the preconditioner. The entry 'Other' comprises the Krylov method, matrix-vector products and other basic linear algebra operations. These parts of the solver seem to scale fairly well. When applying the preconditioner, the orthogonal transformation of the vectors and the solution of the dense sub-blocks scale well whereas the solution of the reduced problem does not. Several factors cause this poor performance of the parallel direct solver: The reduced problem may be too small to be handled efficiently by a large number of cores. Similar to common practice in multigrid methods one might use a subset of the processors here. Furthermore, the fill-reducing ordering is computed based on the graph of the complete matrix. This is likely to lead to pivoting as the *p*-nodes are encountered. The ordering proposed in [3] may help to overcome this issue.

Table I. 3D Driven cavity - grid size 64, subdomain size 8.

(a) Compute phase

	time [sec]				speed-up			
# cores	8	16	32	64	8	16	32	64
compute	301	161	95	50	1	1.9	3.2	6.1
subd. fact.	30	13	7.1	3.5	1	2.4	4.3	8.6
construct S	77	47	23	13	1	1.7	3.4	6.1
init. prec	102	53	37	15	1	1.9	2.8	6.6
comp. prec	91	49	28	18	1	1.9	3.2	5.1
block fact.	12	6.2	3.8	1.6	1	1.9	3	7
transform S	81	42	25	12	1	1.9	3.2	6.7
coarse fact.	7.1	6.4	6.2	6.2	1	1.1	1.2	1.2

(b) Solve Phase

	time [sec]				speed-up			
# cores	8	16	32	64	8	16	32	64
solve phase	92	46	25	42	1	2	3.7	2.2
subd. solve	32	14	6.8	3.9	1	2.2	4.7	8.4
apply prec	6.6	4.8	3.8	30	1	1.4	1.7	.22
apply trans	1.4	.92	.48	.24	1	1.5	2.9	5.6
solve blocks	1.1	.64	.33	.18	1	1.7	3.3	6.1
solve coarse	3.7	3.2	2.9	30	1	1.1	1.3	.12
other	53	27	15	7.7	1	1.9	3.6	6.9

V. Comparison with results in literature

In the previous section we showed results concerning the robustness and performance of the solver developed in [4], [1] and its implementation presented here. A legitimate question is how our method compares to other approaches.

In [11] a number of segregated preconditioning techniques (PCD, SIMPLE and LSC) are compared using Trilinos implementations. A finite element discretization with full Newton linearization of the driven cavity problem is used, both in 2D and 3D. Therefore, this study lends itself for a direct comparison. Results are shown for Reynolds Numbers up to 1 000 (in 2D); the convergence tolerance of 10^{-5} for the linear solves is slightly larger than the 10^{-6} used for our 3D experiments. At $\mathrm{Re} = 10$ (i.e. for the Stokes problem with a mild nonsymmetric perturbation) the PCD method achieves a grid independent convergence rate. Their fastest method (PCD) requires about 40 minutes to compute the steady state solution at $\mathrm{Re} = 1\ 000$ on a 512^2 mesh using 64 Intel CPUs running at 3.6 GHz. The machine we use is faster (4.7 GHz/core) and has more RAM, but the fact that we could compute an unstable steady state at $\mathrm{Re} = 15\ 000$ on 16 CPUs in about 10 minutes shows that our algorithm is more robust and our implementation is competitive in 2D. In 3D their PCD method requires about 28 minutes to compute a steady state at $\mathrm{Re} = 100$ on a 64^3 grid using 8 CPUs. Using our method on 8 Huygens cores, the steady state can be computed in less than 10 minutes, but in terms of memory efficiency and scalability our implementation is currently not competitive in 3D. When the bottlenecks identified earlier on are removed, it should be significantly faster and applicable to larger problems (128^3 as in [11] and beyond).

The segregated approach has the advantage that it can readily be applied to unstructured grid problems because AMG preconditioners are used for the two inner solves. In [12] aggregation-based AMG methods are investigated for nonsymmetric fluid dynamics problems, where the importance of the choice of transfer operators is emphasized. Multigrid methods can typically not be applied to the complete linear system, though, making the use of segregated preconditioners necessary. While we have not investigated the issue theoretically, the fact that we can compute steady states at much higher Reynolds Numbers (where the systems are highly indefinite and non-normal) is strong numerical evidence of the importance of a fully coupled approach.

VI. Discussion

We have presented a parallel implementation of the robust hybrid solver from [4], [1] based on Trilinos. By using object-oriented programming techniques, the code is kept flexible and extensible. The availability of a fast parallel implementation allowed us to perform more detailed experiments than before, revealing that the convergence curves of the preconditioned GMRES method are uniform at the beginning, followed by a series of plateaus as the residual norm becomes small, leading to possible stagnation of the restarted GMRES method. A Krylov method based on Householder transforms may help to overcome this issue [13]. Together with the results on the complexity of the method from Section II-A we conclude that the subdomain size should be chosen as small as possible.

On the other hand, the solution of the reduced problem is the main parallelization pitfall in the present implementation. Improved fill-reducing orderings may help in this case, in particular using an ordering based on the 'compressed graph' $F(A) + F(BB^T)$. For large problems, recursive application of the method becomes essential to sufficiently reduce the problem size before applying a direct solver. This would allow choosing short separators on every level until the reduced problem can be solved efficiently by a sequential method. The program structure is such that this can be implemented easily once the algorithm is fully understood.

Besides investigating the performance of the implementation we have also demonstrated that the solver is robust and efficient at Reynolds Numbers up to 15 000 for the Jacobian of the 2D driven cavity problem. To our best knowledge, results for these highly indefinite and non-normal matrices were not reported in literature before.

Acknowledgment

This research was supported by the Netherlands Organisation for Scientific Research (NWO).

References

[1] F. W. Wubs and J. Thies, "A robust two-level incomplete factorization for (Navier-)Stokes saddle point matrices," *SIAM J. Matrix Anal. Appl.*, in press.

[2] M. Tůma, "A note on the LDL decomposition of matrices from saddle-point problems," *SIAM J. Matrix Anal. Appl.*, vol. 23, 2002.

[3] A. C. de Niet and F. W. Wubs, "Numerically stable LDLT-factorization of F-type saddle point matrices," *IMA Journal of Numerical Analysis*, vol. 29, no. 1, pp. 208–234, 2009. [Online]. Available: http://dx.doi.org/10.1093/imanum/drn005

[4] J. Thies and F. W. Wubs, "A robust parallel ILU solver with grid-independent convergence for the coupled steady incompressible Navier-Stokes equations," in *Proc. ECCOMAS CFD 2010*, J. C. F. Peireira and A. Sequeira, Eds., 2010, CDROM paper 1421.

[5] J. Thies, F. W. Wubs, and H. A. Dijkstra, "Bifurcation analysis of 3D ocean flows using a parallel fully implicit ocean model," *Ocean Modelling*, vol. 40, pp. 287–297, 2009.

[6] G. Karypis and V. Kumar, "Parallel multilevel *k*-way partitioning scheme for irregular graphs," *SIAM Rev.*, vol. 41, no. 2, pp. 278–300, 1999.

[7] J. Gaidamour and P. Hénon, "A parallel direct/iterative solver based on a Schur complement approach," *Computational Science and Engineering, IEEE International Conference on*, vol. 0, pp. 98–105, 2008.

[8] P. Hénon and Y. Saad, "A parallel multistage ILU factorization based on a hierarchical graph decomposition," *SIAM J. Matrix Anal. Appl.*, vol. 28, pp. 2266–2293, 2006.

[9] P. R. Amestoy, I. S. Duff, J. Koster, and J. Y. L'Excellent, "MUMPS: A multifrontal massively parallel solver," *ERCIM News*, vol. 50, pp. 14–15, jul 2002, european Research Consortium for Informatics and Mathematics (ERCIM), http://www.ercim.org.

[10] C. Chevalier and F. Pellegrini, "PT-SCOTCH: a tool for efficient parallel graph ordering," *Parallel Computing*, vol. 34, pp. 318–331, 2008.

[11] H. C. Elman, V. E. Howle, J. Shadid, R. Shuttleworth, and R. S. Tuminaro, "A taxonomy and comparison of parallel block multi-level preconditioners for the incompressible Navier-Stokes equations," *J. Comp. Phys.*, vol. 227, no. 3, pp. 1790–1808, 2008.

[12] M. Sala and R. S. Tuminaro, "A new Petrov-Galerkin smoothed aggregation preconditioner for nonsymmetric linear systems," *SIAM J. Scientific Computing*, vol. 31, no. 1, pp. 143–166, 2008.

[13] M. Sosonkina, L. T. Watson, and R. K. Kapania, "A new adaptive GMRES algorithm for achieving high accuracy," *Numer. Linear Algebra Appl*, vol. 5, pp. 275–297, 1998.