Using the VBARMS method in parallel computing

Bruno Carpentieri[*] Jia Liao[*] Masha Sosonkina[†] Aldo Bonfiglioli[‡]
Sven Baars[*]

Abstract

The paper describes an improved parallel MPI-based implementation of VBARMS, a variable block variant of the pARMS preconditioner proposed by Li, Saad and Sosonkina [NLAA, 2003] for solving general nonsymmetric linear systems. The parallel VBARMS solver can detect automatically exact or approximate dense structures in the linear system, and exploits this information to achieve improved reliability and increased throughput during the factorization. A novel graph compression algorithm is discussed that finds these approximate dense blocks structures and requires only one simple to use parameter. A complete study of the numerical and parallel performance of parallel VBARMS is presented for the analysis of large turbulent Navier-Stokes equations on a suite of three-dimensional test cases.

Keywords: Linear systems, incomplete LU factorization preconditioners, graph compression techniques, parallel performance, distributed-memory computers.

1 Introduction

The initial motivation for this study is the design of robust preconditioning techniques for solving sparse block structured linear systems arising from the finite element / finite volume analysis of turbulent flows in computational fluid dynamics applications. Over the last few years we have developed block multilevel incomplete LU (ILU) factorization methods for this problem class, and we have found them very effective in reducing the number of GMRES iterations compared to their pointwise analogues [8]. This class of preconditioners can offer higher parallelism and robustness than standard ILU algorithms especially for solving large problems, thanks to their multilevel mechanism. Exploiting existing block structures in the matrix can help reduce numerical instabilities during the factorization and achieve higher flops to memory ratios on modern cache-based computer architectures. Sparse matrices arising from the solution of systems of partial differential equations often exhibit perfect block structures consisting of fully dense (typically small) nonzero blocks in their sparsity pattern, e.g., when several unknown physical quantities are associated with the same grid point. For example, a plane elasticity problem has both x- and y-displacements at each grid point; a Navier-Stokes system for turbulent compressible flows would have five distinct variables (the density, the scaled energy, two components of the scaled velocity, and the turbulence transport variable) assigned to each node of the physical mesh; a bidomain system in cardiac electrical dynamics couples the intra-and extra-cellular electric potential at each ventricular cell of the heart. Upon numbering consecutively the ℓ distinct variables associated with the same grid point, the

[*]Institute of Mathematics and Computing Science - University of Groningen, 9747 AG Groningen, The Netherlands - e-mail: `b.carpentieri@rug.nl`, `j.liao@rug.nl`, `s.baars@rug.nl`
[†]Department of Modeling, Simulation & Visualization Engineering - Old Dominion University, Norfolk, VA 23529 - e-mail: `msosonki@odu.edu`
[‡]Scuola di Ingegneria - University of Basilicata, Potenza, Italy - e-mail: `aldo.bonfiglioli@unibas.it`

permuted matrix has a sparse block structure with nonzero blocks of size $\ell \times \ell$. The blocks are fully dense if variables at the same node are mutually coupled.
Our recently developed variable block algebraic recursive multilevel solver (shortly, VBARMS) can detect fine-grained dense structures in the linear system automatically, without any user's knowledge of the underlying problem, and exploit them efficiently during the factorization [8]. Preliminary experiments with a parallel MPI-based implementation of VBARMS for distributed memory computers, presented in a conference contribution [7], showed the robustness of the proposed method for solving some larger matrix problems arising in different fields. In this paper, capitalizing on those results, we introduce a new graph-based compression algorithm to construct the block ordering in VBARMS, which extends the method proposed by Ashcraft in [1] and requires only one simple to use parameter (Section 2); we describe in Section 3 a novel implementation of the block partial factorization step that proves to be noticeably faster than the original one presented in [8]; finally, in Section 4, we assess the parallel performance of our parallel VBARMS code for solving turbulent Navier-Stokes equations in fully coupled form on large realistic three-dimensional meshes; in the new parallel implementation, we use a parallel graph partitioner to reduce the graph partitioning time significantly compared to the experiments presented in [7].

2 Graph compression techniques

It is known that block iterative methods often show faster convergence rate than their pointwise analogues in the solution of many classes of two- and three-dimensional partial differential equations (PDEs). When the domain is discretized by cartesian grids, a regular partition may also provide an effective matrix partitioning. For example, in the case of the simple Poisson's equation with Dirichlet boundary conditions, defined on a rectangle $(0, \ell_1) \times (0, \ell_2)$ discretized uniformly by $n_1 + 2$ points in the interval $(0, \ell_1)$ and $n_2 + 2$ points in $(0, \ell_2)$, upon numbering the interior points in the natural ordering by lines from the bottom up, one obtains a $n_2 \times n_2$ block tridiagonal matrix with square blocks of size $n_1 \times n_1$; the diagonal blocks are tridiagonal matrices and the off-diagonal blocks are diagonal matrices. For large finite element discretizations, it is common to use substructuring, where each substructure of the physical mesh corresponds to one sparse block of the system. However, if the domain is highly irregular or the matrix does not correspond to a differential equation, finding the best block partitioning is much less obvious. In this case, graph reordering techniques are worth considering.
The PArameterized BLock Ordering (PABLO) method proposed by O'Neil and Szyld is one of the first matrix partitioning algorithms specifically designed for block iterative solvers [15]. The algorithm selects groups of nodes in the adjacency graph of the coefficient matrix such that the corresponding diagonal blocks are either full or very dense. It has been shown that classical block stationary iterative methods such as block Gauss-Seidel and SOR methods combined with the PABLO ordering require fewer operations than their point analogues for the the finite element discretization of a Dirichlet problem on a graded L-shaped region, as well as on the 9-point discretization of the Laplacian operator on a square grid. The complexity of the PABLO algorithm is proportional to the number of nodes and edges in both time and space.
Another useful approach for blocking a matrix A is to find block independent sets in the adjacency graph of A [21]. A block independent set is defined as a set of groups of nodes (or unknowns) having the property that there is no coupling between nodes of any two different groups, while nodes within the same group may be coupled. Independent sets of unknowns in a linear system can be eliminated simultaneously at a given stage of Gaussian Elimination. For this reason, this type of oredering is extensively adopted in linear solvers design. Independent sets may be computed by using simple graph algorithms which traverse the vertices of the adjacency graph of A in the natural order $1, 2, \ldots, n$, mark each visited vertex v and all of its nearest neighbors connected to v by an edge, and add v and each visited node that is not already marked to the current independent set partition [18]. Upon renumbering nodes one partition

after the other, followed as last by interface nodes straddling between separate partitions, one obtain a permutation of A in the form

$$PAP^T = \begin{pmatrix} D & F \\ E & C \end{pmatrix}, \tag{1}$$

where D is a block diagonal matrix. The nested dissection ordering by George [10], mesh partitioning, or further information from the set of nested finite element grids of the underlying problem [2, 3, 6] can be used as an alternative to the greedy independent set algorithm described above. Additionally, the numerical values of A may be incorporated in the ordering to produce more robust factorizations [21].
However, finite element and finite difference matrices often possess also fine-grained block structures that can be exploited in iterative solvers. If there is more than one solution component at a grid point, the corresponding matrix entries may form a small dense block and optimized codes can be used for dense factorizations in the construction of the preconditioner and dense matrix-vector products in the sparse matrix-vector product operation for better performance, see e.g. [8, 11, 19, 23, 24]. A block incomplete LU factorization (ILU) method is one preconditioning technique that treats small dense submatrices of A as single entities, and the VBARMS method discussed in this paper can be seen as its natural multilevel generalization. An important advantage of block ILU versus conventional ILU is the potential gain obtained from using optimized level 3 basic linear algebra subroutines (BLAS3). Column indices and pointers can be saved by storing the matrix as a collection of blocks using the variable block compressed sparse row (VBCSR) format, where each value in the CSR format is a dense array. On indefinite problems, computing with blocks instead of single elements enables us a better control of pivot breakdowns, near singularities, and other sources of numerical instabilities. These facts have been assessed in our previous contribution [8].
The method proposed by Ashcraft in [1] is one of the first compression techniques for finding dense blocks in the sparsity pattern of a matrix. The algorithm searches for sets of rows or columns having the exact same pattern. From a graph viewpoint, it looks for vertices of the adjacency graph (V, E) of A having the same adjacency list. These are also called *indistinguishable nodes* or *cliques*. The algorithm assigns a *checksum* quantity to each vertex, e.g., using the function

$$chk(u) = \sum_{(u,w)\in E} w, \tag{2}$$

and then sorts the vertices by their checksums. This operation takes $|E| + |V| \log |V|$ time. If u and v are indistinguishable, then $chk(u) = chk(v)$. Therefore, the algorithm examine nodes having the same checksum to see if they are indistinguishable. The ideal checksum function would assign a different value for each different row pattern that occurs but it is not practical because it may quickly lead to huge numbers that may not even be machine-representable. Since the time cost required by Ashcraft's method is generally negligible relative to the time it takes to solve the system, simple checksum functions such as (2) are used in practice [1].
Sparse unstructured matrices may sometimes exhibit *approximate dense blocks* consisting mostly of nonzero entries except only a few zeros inside the blocks. By treating these few zeros as nonzero elements, with a little sacrifice of memory, a block ordering may be generated for an iterative solver. Computing approximate dense structures may enable us to enlarge existing blocks and to use BLAS3 operations more efficiently in the iterative solution, but it may also increase the memory costs and the probability to encounter singular blocks during the factorization [8]. Two important performance measures to gauge the quality of the block ordering computed are the *average block density* (*av_bd*) value, defined as the amount of nonzeros in the matrix divided by the amount of elements in the nonzero blocks, and the *average block size* (*av_bs*) value, which is the ratio between the sum of dimensions of the square diagonal blocks divided by the number of diagonal blocks. From our computational experience, high average block density values around 90% are necessary to prevent the occurrence of singular blocks during the factorization.

2.1 The angle-based method

Approximate dense blocks in a matrix may be computed by numbering consecutively rows and columns having a similar nonzero structure. However, this would require a new checksum function that preserves the proximity of patterns, in the sense that close patterns would result in close checksum values. Unfortunately, this property does not hold true for Ashcraft's algorithm in its original form. In [19], Saad proposed to compare angles of rows (or columns) to compute approximate dense structures in a matrix A. Let C be the pattern matrix of A, which by definition has the same pattern as A and nonzero values equal to one. The method proposed by Saad computes the upper triangular part of CC^T. Entry (i, j) is the inner product (the cosine value) between row i and row j of C for $j > i$. A parameter τ is used to gauge the proximity of row patterns. If the cosine of the angle between rows i and j is smaller than τ, row j is added to the group of row i. For $\tau = 1$ the method will compute perfectly dense blocks, while for $\tau < 1$ it may compute larger blocks where some zero entries are padded in the pattern. To speed up the search, it may be convenient to run a first pass with the checksum algorithm to detect rows having an identical pattern, and group them together; then, in a second pass, each non-assigned row is scanned again to determine whether it can be added to an existing group. In practice, however, it may be difficult to predict the average block density obtained using a given value of τ . For example, the experiments reported in Table 1 show that $\tau = 0.58$ returns a block density of 86.37% for the VENKAT01 matrix and of 45.06% for the STACOM matrix.

Matrix	$\tau = 0.56$	$\tau = 0.57$	$\tau = 0.58$	$\tau = 0.59$	$\tau = 0.60$
STACOM	25.63	25.68	45.06	50.83	52.02
K3PLATES	37.78	38.73	58.62	58.70	59.16
OILPAN	50.08	50.09	50.23	50.23	90.65
VENKAT01	29.71	29.71	86.37	86.37	86.37
RAE	26.40	26.48	49.48	50.71	51.96

Matrix	$\tau = 0.64$	$\tau = 0.65$	$\tau = 0.66$	$\tau = 0.67$	$\tau = 0.68$
RAEFSKY3	63.32	63.32	63.32	95.23	95.23
BMW7ST_1	49.29	50.11	50.66	68.85	74.00
S3DKQ4M2	64.29	64.29	64.29	97.52	97.52
PWTK	57.05	57.31	57.48	94.23	94.75

Table 1: Average block density value (%) obtained from the angle compression algorithm for different values of τ.

The cost of Saad's method is closer to that of checksum-based methods for cases in which a good blocking already exists, and in most cases it remains inferior to the cost of the least expensive block LU factorization, i.e., block ILU(0).

2.2 Graph-based compression

We revisited Saad's angle-based method to develop a new compression algorithm that computes a block ordering having an average block density *av_bd* not smaller than a user-specified value μ. This may simplify the parameter selection procedure. The method proceeds in two steps. First, using the checksum algorithm it groups rows having equal nonzero structure and builds the quotient graph $G/\mathcal{B} = (V_\mathcal{B}, E_\mathcal{B})$. The quotient graph $\mathcal{G}/\mathcal{B}$ is constructed by coalescing rows with identical pattern into one supervertex (or supernode) Y_i of $V_\mathcal{B}$. We can write

$$V_\mathcal{B} = \{Y_1, \dots, Y_p\}, \quad E_\mathcal{B} = \{(Y_i, Y_j) \mid \exists v \in Y_i, w \in Y_j \text{ s.t. } (v, w) \in E\}$$

where $G = (V, E)$ is the graph of A. An edge connects two supervertices Y_i and Y_j if there exists an edge of G connecting a vertex connecting a vertex in Y_i to a vertex in Y_j. If A is unsymmetric, we assume to operate on the symmetrized graph of $A + A^T$; thus the edge orientation is not important. Afterwards, the algorithm merges pairs of supervertices (Y, X), for X adjacent to Y in $G/\mathcal{B}$, provided that the density of the rows that are involved in this particular merge after this operation does not drop below μ. Otherwise, the algorithm will stop to prevent near-singularities during the block factorization. The total size of the rows and columns spanned by this new block is

$$T = 2 \cdot |adj(Y) \cup adj(X)| \cdot |Y \cup X| - |Y \cup X|^2 ,$$

which is the amount of nonzero rows and columns times the size of the supervertex minus the square block on the diagonal which we count twice since we count both columns and rows. The nonzeros spanned by the new block is

$$N = 2 \cdot \sum_{Z \in Y \cup X} |adj(Z)| - \sum_{Z \in Y \cup X} |adj(Z) \cap (Y \cup X)|,$$

which is the amount of adjacent nodes per node inside the supervertex minus the amount of nodes inside the diagonal block, which is again counted twice. The complete graph-based algorithm is sketched in Algorithm 1. It requires only one simple to use parameter μ. If we desire a block ordering having a block density around 60%, we simply set $\mu = 0.6$. In contrast, a correct tuning of τ may require to run the full solver to see if a singular block is encountered during the factorization.

2.3 Experiments

In this section we give some comparative performance figures to show the viability of the graph algorithm. We summarize in Table 2 the characteristics of the test matrix problems, and we present the results of our experiments in Table 3. In our runs, we attempted to find the optimal value of τ by trial and error. By optimal value we mean the one that minimizes the number of GMRES iterations required to reduce the initial residual by 6 orders of magnitude using a standard block incomplete LU factorization as a preconditioner for GMRES. The optimal value for the parameter τ was calculated by running the angle algorithm with different $\tau \in [0.5, 1.0]$, by increments of 0.1 at every run. The results evidence the difficulty to compute a unique value which is nearly optimal for every problem. On the other hand, for the graph method we set $\mu = 0.7$ which gave us a minimum block density of 70% for every matrix. We see that the new compression algorithm is very competitive and additionally may be simple to use. In Table 3 we also report on the timing to compute the block ordering by both compression techniques, and for solving the linear system. The new graph algorithm is in most cases up to three times slower than the angle algorithm. However, this is not a big downside because the compression time is considerably smaller than the total solution time, and computing the optimal value of τ may require several runs as we explained. Clearly, the compression time increases when μ decreases since we merge more supervertices in this circumstance. By the way, both compression methods helped reduce iterations. Without blocking, no convergence was achieved in 1000 iterations using pointwise ILUT on the OILPAN, K3PLATES, S3DKQ4M2, OLAFU, RAE, NASASRB, CT20STIF, RAEFSKY3, BCSSTK35, STACOM problems at equal or higher memory usage. On the other hand, no evident gain was observed from using level-2 BLAS routines in the sparse matrix-vector product operation, probably due to the small block size.

Name	Size	Application	nnz(A)	symmetry
OILPAN	73752	Structural problem	2148558	symmetric value
K3PLATES	11107	FE stiffness matrix	378927	symmetric value

Name	Size	Application	nnz(A)	symmetry
VENKAT01	62424	Unstructured 2D Euler solver	1717792	symmetric structure
PWTK	217918	Pressurized wind tunnel	11524432	symmetric value
S3DKQ4M2	90449	Structural mechanics	2455670	symmetric value
OLAFU	16146	Structural problem	1015156	symmetric value
RAE	52995	Turbulence analysis	1748266	symmetric structure
BMW7ST_1	141347	Stiffness matrix	7318399	symmetric value
NASASRB	54870	Shuttle rocket booster	2677324	symmetric value
CT20STIF	52329	Stiffness matrix engine block	2600295	symmetric value
RAEFSKY3	21200	Fluid structure interaction turbulence problem	1488768	symmetric structure
HEART1	3557	Quasi-static FEM of a heart	1385317	symmetric structure
BCSSTK35	30237	Automobile seat frame	1450163	symmetric value
STACOM	8415	Compressible flow	271936	symmetric structure

Table 2: Set and characteristics of test matrix problems.

Matrix	Method	τ/μ	*av_bd* (%)	*av_bs*	Blocking time (s)	Solving time (s)	Mem	Its
OILPAN	Angle	0.70	95.94	7.36	0.03	4.18	0.26	198
	Graph	0.70	95.02	7.42	0.08	4.17	0.27	198
K3PLATES	Angle	0.60	59.16	7.90	0.00	0.7	0.3	239
	Graph	0.70	89.50	5.65	0.01	0.7	0.18	241
VENKAT01	Angle	0.70	99.94	4.00	0.02	0.43	1.33	9
	Graph	0.70	94.05	4.28	0.08	0.48	1.58	9
PWTK	Angle	0.60	56.95	12.17	0.09	26.38	6.85	117
	Graph	0.70	78.16	7.31	0.35	32.64	4.5	137
S3DKQ4M2	Angle	1.00	100.00	5.93	0.03	9.57	1.09	214
	Graph	0.70	77.92	7.81	0.12	15.1	1.42	309
OLAFU	Angle	0.80	81.75	6.47	0.02	1.2	3.14	54
	Graph	0.70	79.66	6.58	0.11	1.63	3.75	57
RAE	Angle	0.80	95.83	4.67	0.03	8.85	9.53	49
	Graph	0.70	86.21	4.64	0.13	15.74	13.8	42
BMW7ST_1	Angle	0.70	77.16	7.28	0.08	0.35	0.18	5
	Graph	0.70	79.54	6.65	0.29	0.48	0.17	9
NASASRB	Angle	0.80	90.87	4.24	0.05	7.51	5.23	30
	Graph	0.70	77.62	4.20	0.20	12.39	7.46	16
CT20STIF	Angle	0.70	66.05	6.55	0.04	0.69	0.18	44
	Graph	0.70	78.42	4.76	0.16	1.18	0.14	56
RAEFSKY3	Angle	0.70	95.23	8.63	0.01	0.08	0.13	13
	Graph	0.70	77.67	10.56	0.02	0.09	0.17	15
HEART1	Angle	0.90	98.81	18.62	0.00	0.5	0.78	151
	Graph	0.70	0.00	0.00	0.00	-	-	-
BCSSTK35	Angle	0.60	51.95	11.03	0.01	2.1	0.29	209
	Graph	0.70	78.72	6.57	0.05	2.66	0.18	235

Matrix	Method	τ/μ	*av_bd* (%)	*av_bs*	Blocking time (s)	Solving time (s)	Mem	Its
STACOM	Angle	0.90	97.00	4.36	0.00	0.25	5.19	31
	Graph	0.70	84.51	4.47	0.01	0.29	5.65	33

Table 3: Experiments with the angle-based and the graph-based compression methods. The optimal value of τ is used for the angle-based algorithm. The value $\mu = 0.7$ is used for the graph-based algorithm in all our runs. The number of iterations refer to the VBILUT preconditioner.

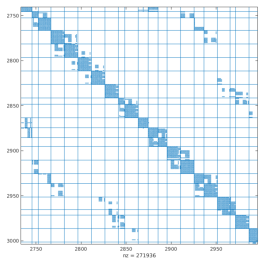
Algorithm 1 *The graph based compression algorithm.*

```
1: Compute the keys $k_i = chk(i)$ for all vertices $i \in V = \{1, \dots, n\}$
2: Set processed nodes $p_i = 0 \ \forall i = 1, \dots, n$
3: Make a set of supervertices $\mathcal{V} = \emptyset$
4: Set $s$ to the indices $V$ sorted by the corresponding value in $k$
5: for $i = s_1, \dots, s_n$ do
6:   if $p_i \neq 1$ then
7:     Add a new supervertex $Y_i$ to $\mathcal{V}$
8:     for $j = s_{i+1}, \dots, s_n$ do
9:       if $k_i \neq k_j$ then
10:        break
11:      if $adj(i) = adj(j)$ then
12:        Add node $j$ to $Y_i$
13:        Set $p_j = 1$
14: Make a map $\mathcal{M} : i \mapsto \{Z \in \mathcal{V} | \ i \in adj(Z)\}$
15: for $X \in \mathcal{V}$ do
16:   for $Z \in \bigcup_{i \in X} \mathcal{M}(i)$ do
17:     Compute the block density value $bd$ of the rows that are involved after merging $X$ and $Z$.
18:     if $bd \geq \mu$ then
19:       $X = X \cup Z$
20:       $\mathcal{V} = \mathcal{V} \backslash Z$
```

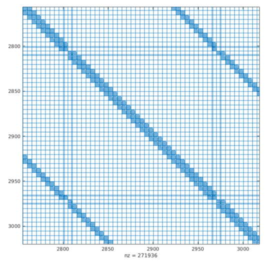
For the sake of comparison, we also ran some experiments using the PABLO algorithm introduced by O'Neil and Szyld in [15], in combination with block incomplete LU factorization preconditioning. The convergence results are reported in Table 4, and a comparison of patterns produced by the two compression techniques is shown in Figure 1 for two matrices. We observe that the block ordering computed by PABLO may produce larger blocks compared to the graph and angle methods. However, the average block size can be significantly smaller, probably due to the design philosophy of PABLO that attempts to maximize the density of the diagonal blocks of a matrix. The convergence results show that overall the resulting block ordering may be less suitable for block factorization.

Matrix	*av_bd*	*av_bs*	Total time (s)	Mem	Its
STACOM	66.54	2.38	6.22	11.02	152
K3PLATES	83.51	2.00	8.94	5.54	329
OLAFU	89.60	2.00	7.66	3.89	84
RAE	68.28	2.34	412.89	26.75	1000

Table 4: Performance of the PABLO ordering with VBILUT. The quantity *av_bd* refers to the average block density of the block ordering, *av_bs* is the average block size, *Total time* includes the preconditioning construction and the solving time, *Mem* is the ratio between the number of nonzeros in the preconditioner and in the matrix.



(a) Using the PABLO algorithm



(b) Using the graph algorithm

Figure 1: Block patterns computed by different compression methods for the STACOM problem. The figure shows a zoom of one window area of the matrix.

3 The VBARMS method

The VBARMS method discussed in this paper incorporates compression techniques to maximize computational efficiency during the factorization. We recall briefly below the main steps of the algorithm and we point the reader to [8] for further details. After permuting the coefficient matrix A in block form as

$$\widetilde{A} \approx P_B A P_B^T = \begin{bmatrix} \widetilde{A}_{11} & \widetilde{A}_{12} & \cdots & \widetilde{A}_{1p} \\ \widetilde{A}_{21} & \widetilde{A}_{22} & \cdots & \widetilde{A}_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ \widetilde{A}_{p1} & \widetilde{A}_{p2} & \cdots & \widetilde{A}_{pp} \end{bmatrix}, \tag{3}$$

where the diagonal blocks $\widetilde{A}_{ii}$, $i = 1, \ldots, p$ are $n_i \times n_i$, the off-diagonal blocks $\widetilde{A}_{ij}$ are $n_i \times n_j$, and P_B is the permutation matrix of the block ordering computed by the compression algorithm, we can represent the adjacency graph of $\widetilde{A}$ by the quotient graph of $A + A^T$ [10], which is smaller. Let $\mathcal{B}$ the partition into blocks given by (3) and $\mathcal{G}/\mathcal{B} = (V_{\mathcal{B}}, E_{\mathcal{B}})$ the quotient graph constructed by coalescing the vertices of each block $\widetilde{A}_{ii}$, for $i = 1, \ldots, p$, into one supervertex (or supernode) Y_i. An edge of $E_{\mathcal{B}}$ connects two supervertices Y_i and Y_j of $V_{\mathcal{B}}$ if there exists an edge of (V, E) connecting a vertex of the block A_{ii} to a vertex of the block A_{jj}.

The complete pre-processing and factorization process of VBARMS consists of the following steps.

Step 1 Using the angle-based or the graph-based compression algorithms described in Section 2, compute a block ordering P_B of A such that, after permutation, the matrix $P_B A P_B^T$ has fairly dense nonzero blocks.

Step 2 Scale the matrix permuted at **Step 1** as $S_1 P_B A P_B^T S_2$, where S_1 and S_2 are two diagonal matrices such that the 1-norm of the largest entry in each row and column becomes smaller or equal than one.

Step 3 Apply the block independent sets (or the nested dissection) algorithms to the quotient graph $\mathcal{G}/\mathcal{B}$ and compute an independet sets ordering P_I of $\mathcal{G}/\mathcal{B}$. Upon permutation by P_I, the matrix obtained at **Step 2** will write as

$$P_I S_1 P_B A P_B^T S_2 P_I^T = \begin{pmatrix} D & F \\ E & C \end{pmatrix}. \tag{4}$$

We use a simple weighted greedy algorithm for computing the ordering P_I [21].

In the 2×2 partitioning (4), the upper left-most matrix $D \in \mathbb{R}^{m \times m}$ is block diagonal like in ARMS. However, due to the block permutation (**Step 1**), the diagonal blocks D_i of D are block sparse matrices while in ARMS they are sparse unstructured. The matrices $F \in \mathbb{R}^{m \times (n-m)}$, $E \in \mathbb{R}^{(n-m) \times m}$, $C \in \mathbb{R}^{(n-m) \times (n-m)}$ are also block sparse, because of the same reason.

Step 4 Factorize the matrix in (4) as

$$\begin{pmatrix} D & F \\ E & C \end{pmatrix} = \begin{pmatrix} L & 0 \\ EU^{-1} & I \end{pmatrix} \times \begin{pmatrix} U & L^{-1}F \\ 0 & A_1 \end{pmatrix}, \tag{5}$$

where I is the identity matrix of appropriate size, and

$$A_1 = C - ED^{-1}F. \tag{6}$$

is the Schur complement corresponding to C. Observe that the Schur complement is also block sparse and it has the same block structure as matrix C.

Steps 2-4 can be repeated on the reduced system a few times until the Schur complement is small enough. Denoting by A_ℓ the reduced Schur complement matrix at level ℓ, for $\ell > 1$, after scaling and preordering A_ℓ a system with coefficient matrix

$$P_I^{(\ell)} D_1^{(\ell)} A_\ell D_2^{(\ell)} (P_I^{(\ell)})^T = \begin{pmatrix} D_\ell & F_\ell \\ E_\ell & C_\ell \end{pmatrix} = \begin{pmatrix} L_\ell & 0 \\ E_\ell U_\ell^{-1} & I \end{pmatrix} \times \begin{pmatrix} U_\ell & L_\ell^{-1} F_\ell \\ 0 & A_{\ell+1} \end{pmatrix} \tag{7}$$

needs to be solved, with $D_\ell \in \mathbb{R}^{m_\ell \times m_\ell}$, $F_\ell \in \mathbb{R}^{m_\ell \times (n_\ell - m_\ell)}$, $E_\ell \in \mathbb{R}^{(n_\ell - m_\ell) \times m_\ell}$, $C_\ell \in \mathbb{R}^{(n_\ell - m_\ell) \times (n_\ell - m_\ell)}$, and

$$A_{\ell+1} = C_\ell - E_\ell D_\ell^{-1} F_\ell \in \mathbb{R}^{(n_\ell - m_\ell) \times (n_\ell - m_\ell)}. \tag{8}$$

Calling

$$x_\ell = \begin{pmatrix} y_\ell \\ z_\ell \end{pmatrix}, \quad b_\ell = \begin{pmatrix} f_\ell \\ g_\ell \end{pmatrix}$$

the unknown solution vector and the right-hand side vector of system (7), respectively, the solution process with the above multilevel VBARMS factorization consists of a level-by-level forward elimination step followed by an exact solution on the last reduced subsystem and a suitable inverse permutation. The complete solving phase is sketched in Algorithm 2.

In VBARMS we perform the factorization approximately for memory efficiency. We use block ILU factorization with threshold to invert inexactly both the upper left-most matrix $D_\ell \approx \bar{L}_\ell \bar{U}_\ell$, at each

Algorithm 2 `VBARMS_Solve`$(A_{\ell+1}, b_\ell)$. The solving phase with the VBARMS method.

```
Require: $\ell \in \mathbb{N}^*$, $\ell_{max} \in \mathbb{N}^*$, $b_\ell = (f_\ell, g_\ell)^T$
1: Solve $L_\ell y = f_\ell$
2: Compute $g'_\ell = g_\ell - E_\ell U_\ell^{-1} y$
3: if $\ell = \ell_{max}$ then
4:     Solve $A_{\ell+1} z_\ell = g'_\ell$
5: else
6:     Call VBARMS_Solve($A_{\ell+1}, g'_\ell$)
7: Solve $U_\ell y_\ell = \left[ y - L_\ell^{-1} F_\ell z_\ell \right]$
```

level ℓ, and the last level Schur complement matrix $A_{\ell_{max}} \approx \bar{L}_S \bar{U}_S$. The block ILU method used in VBARMS is a straightforward block variant of the one-level pointwise ILUT algorithm. We drop small blocks $B \in \mathbb{R}^{m_B \times n_B}$ in $\bar{L}_\ell$, $\bar{U}_\ell$, $\bar{L}_S$, $\bar{U}_S$ whenever $\frac{\|B\|_F}{m_B \cdot n_B} < t$, for a given user-defined threshold t. The block pivots in block ILU are inverted exactly by using Gaussian Elimination with partial pivoting. Every operation performed during the factorization calls optimized level-3 BLAS routines [9], taking advantage of the finest block structure appearing in the matrices D_ℓ, F_ℓ, E_ℓ, C_ℓ. Recall that this fine-level block structure results from the block ordering P_B and consists of small, usually dense, blocks in the diagonal blocks of D_ℓ as well as in the matrices E_ℓ, F_ℓ, C_ℓ. We do not drop entries in the construction of the Schur complement except at the last level. The same threshold is applied in all these operations.

Algorithm 3 *General ILU Factorization, IKJ Version.*

```
Require: A nonzero pattern set $\mathcal{P}$
1: for $i = 2, \ldots, n$ do
2:   for $k = 1, \ldots, i-1$ do
3:     if $(i,j) \in \mathcal{P}$ then
4:       $a_{ik} = a_{ik}/a_{kk}$
5:     for $j = k+1, \ldots, n$ do
6:       if $(i,j) \in \mathcal{P}$ then
7:         $a_{ij} = a_{ij} - a_{ik} a_{kj}$
```

3.1 The new implementation of VBARMS

The code for the VBARMS method is developed in the C language and is adapted from the existing ARMS code available in the ITSOL package [13]. The compressed sparse storage format of ARMS is modified to store block vectors and block matrices of variable size as a collection of contiguous nonzero dense blocks (we refer to this data storage format as VBCSR). However, the implementation used in this paper is different and noticeably faster than the one described in [8]. In the old implementation, the approximate transformation matrices $E_\ell \bar{U}_\ell^{-1}$ and $\bar{L}_\ell^{-1} F_l$ appearing in Eqn (7) at step ℓ were explicitly computed and temporarily stored in the VBCSR format. They were discarded from the memory immediately after assembling $A_{\ell+1}$. In the new implementation, we first compute the factors $\bar{L}_\ell$, $\bar{U}_\ell$ and $\bar{L}_\ell^{-1} F_\ell$ by performing a variant of the IKJ version of the Gaussian Elimination algorithm (Algorithm 3), where index I runs from 2 to m_ℓ, index K from 1 to $(I-1)$ and index J from $(K+1)$ to n_ℓ. This loop applies implicitly $\bar{L}_\ell^{-1}$ to the block row $[D_\ell \ , \ F_\ell]$ to produce $\left[U_\ell \ , \ \bar{L}_\ell^{-1} F_\ell\right]$. In the second loop, Gaussian Elimination is performed on the block row $[E_\ell \ , \ C_\ell]$ using the multipliers computed in the first loop to give $E_\ell \bar{U}_\ell^{-1}$ and an approximation of the Schur complement $A_{\ell+1}$. We explicitly permute the matrix after Step 1 at the first level as well as the matrices involved in the factorization at each new reordering step. The improvement of efficiency obtained with the new implementation is noticeable, as appears from the results shown in Table 5. Finally, in Table 6 we assess the performance of the VBARMS method against

Matrix	Implementation	Factorization time (s)	Solving time (s)	Total time (s)	Mem	Its
HEART1	New	0.12	0.43	0.55	0.83	147
	Old	0.36	0.33	0.69	0.86	113
PWTK	New	12.71	25.02	37.73	4.42	144
	Old	90.73	26.08	116.81	4.95	140
RAE	New	1.45	1.28	2.72	2.46	34
	Old	5.12	1.15	6.27	2.71	30
NASASRB	New	2.56	3.68	6.23	3.86	76
	Old	15.54	3.34	18.88	4.06	64
OILPAN	New	0.77	1.63	2.39	2.57	42
	Old	5.64	1.29	6.93	2.62	32
BCSSTK35	New	0.15	3.22	3.36	0.95	242
	Old	-	-	-	-	-

Table 5: Comparative experiments with the old and the new VBARMS codes, implementing a different partial (block) factorization step. The symbol '-' means that no convergence is achieved after 1000 iterations of GMRES.

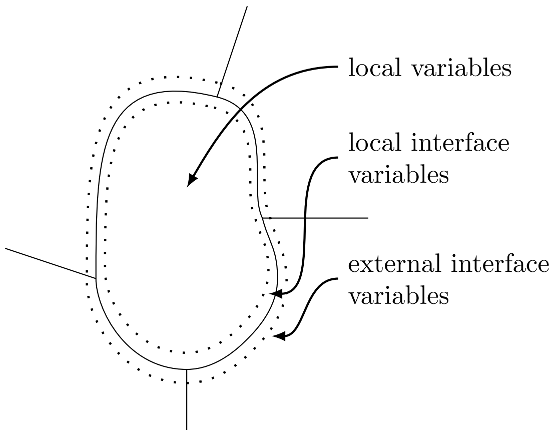
other popular preconditioning techniques on selected linear systems from Table 2 that are representative of the general trend; we report on the number of GMRES iterations required to reduce the initial residual by 6 orders of magnitude using a block incomplete LU factorization as a preconditioner for GMRES. The results show a remarkable robustness for low to moderate memory cost. We point the reader to [8] for more extensive results.

4 Using VBARMS in parallel computing

In the experiments reported in this section the VBARMS method is used for solving large linear systems on distributed memory computers; its overall performance are assessed against the parallel implementation of the ARMS solver provided in the pARMS package [14]. On multicore machines, the quotient graph $\mathcal{G}/\mathcal{B}$ is split into distinct subdomains using a parallel graph partitioner, and each of them is assigned to a different core. We follow the parallel framework described in [14] which separates the nodes assigned to the ith subdomain into *interior nodes*, that are those coupled only with local variables by the equations, and *interface nodes*, those that may be coupled with local variables stored on processor i as well as with remote variables stored on other processors (see Figure below).

Matrix	Bsize	Bdensity	τ	Method	Factorization time (s)	Solving time (s)	Total time (s)	Mem	Its
HEART1	18.62	98.81	0.9	VBARMS	0.12	0.43	0.55	0.83	147
PWTK	56.95	12.17	0.6	VBARMS	12.71	25.02	37.73	4.42	144
RAE	4.67	95.83	0.8	VBARMS	1.45	1.28	2.72	2.46	34
NASASRB	9.18	47.35	0.6	VBARMS	2.56	3.68	6.23	3.86	76
				VBILUT	1.5	23.02	24.52	4.58	464
OILPAN	7.01	99.94	0.8	VBARMS	0.77	1.63	2.39	2.57	42
				ILUT	0.06	32.02	32.08	0.02	952
BCSSTK35	11.03	51.95	0.6	VBILUT	0.09	2.95	3.03	1.08	243
				VBARMS	0.15	3.22	3.36	0.95	242

Table 6: Assessment performance of VBARMS against other popular preconditioning methods. In our experiments, we considered the ILUT, VBILUT and ARMS methods; in the table, only runs with solvers achieving convergence within 1000 iterations of GMRES are reported.



The vector of the local unknowns x_i and the local right-hand side b_i are split accordingly in two separate components: the subvector corresponding to the internal nodes followed by the subvector of the local interface variables

$$x_i = \begin{pmatrix} u_i \\ y_i \end{pmatrix}, \quad b_i = \begin{pmatrix} f_i \\ g_i \end{pmatrix}.$$

The rows of A corresponding to the nodes belonging to the ith subdomain are assigned to the ith processor. They are naturally separated into a local matrix A_i acting on the local variables $x_i = (u_i, y_i)^T$, and an interface matrix U_i acting on the remotely stored subvectors of the external interface variables $y_{i,ext}$. Hence we can write the local equations on processor i as

$$A_i x_i + U_{i,ext} y_{i,ext} = b_i$$

or, in expanded form, as

$$\begin{pmatrix} B_i & F_i \\ E_i & C_i \end{pmatrix} \begin{pmatrix} u_i \\ y_i \end{pmatrix} + \begin{pmatrix} 0 \\ \sum_{j \in N_i} E_{ij} y_j \end{pmatrix} = \begin{pmatrix} f_i \\ g_i \end{pmatrix}, \tag{9}$$

where N_i is the set of subdomains that are neighbors to subdomain i and the submatrix $E_{ij} y_j$ accounts for the contribution to the local equation from the jth neighboring subdomain. Notice that matrices B_i,

C_i, E_i and F_i still preserve the finest block structure imposed by the block ordering P_B. At this stage, the VBARMS method described in Section 3 can be used as a local solver for different types of global preconditioners.

In the simplest parallel implementation, the so-called block-Jacobi preconditioner, the sequential VBARMS method can be applied to invert approximately each local matrix A_i. The standard Jacobi iteration for solving $Ax = b$ is defined as

$$x_{n+1} = x_n + D^{-1}(b - Ax_n) = D^{-1}(Nx_n + b)$$

where D is the diagonal of A, $N = D - A$ and x_0 is some initial approximation. In cases we have a graph partitioned matrix, the matrix D is block diagonal and the diagonal blocks of D are the local matrices A_i. The interest to consider this basic approach is its inherent parallelism, since the solves with the matrices A_i are performed independently on all the processors and no communication is required.

If the diagonal blocks of the matrix D are enlarged in the block-Jacobi method so that they overlap slightly, the resulting preconditioner is called Schwarz preconditioner. Consider again a graph partitioned matrix with N nonoverlapping sets W_i^0, $i = 1, \ldots, N$ and $W_0 = \bigcup_{i=1}^N W_0^i$. We define a δ-overlap partition

$$W^\delta = \bigcup_{i=1}^N W_i^\delta$$

where $W_i^\delta = adj\left(W_i^{\delta-1}\right)$ and $\delta > 0$ is the level of overlap with the neighbouring domains. For each subdomain, we define a restriction operator R_i^δ, which is an $n \times n$ matrix with the (j, j)th element equal to 1 if $j \in W_i^\delta$, and zero elsewhere. We then denote

$$A_i = R_i^\delta A R_i^\delta.$$

The global preconditioning matrix M_{RAS} is defined as

$$M_{RAS}^{-1} = \sum_{i=1}^s R_i^T A_i^{-1} R_i.$$

and named as the Restricted Additive Schwarz preconditioner (RAS) [16, 20]. Note that the preconditioning step is still parallel, as the different components of the error update are formed independently. However, some communication is required in the final update, as the components are added up from each subdomain due to overlapping. In our experiments, the overlap used for RAS was the level 1 neighbours of the local nodes in the quotient graph.

A third global preconditioner that we consider in this study is based on the Schur complement approach. In Eqn (9), we can eliminate the vector of interior unknowns u_i from the first equations to compute the local Schur complement system

$$S_i y_i + \sum_{j \in N_i} E_{ij} y_j = g_i - E_i B_i^{-1} f_i \equiv g_i',$$

where S_i denotes the local Schur complement matrix

$$S_i = C_i - E_i B_i^{-1} F_i.$$

The local Schur complement equations considered altogether write as the global Schur complement system

$$\begin{pmatrix} S_1 & E_{12} & \ldots & E_{1p} \\ E_{21} & S_2 & \ldots & E_{2p} \\ \vdots & & \ddots & \vdots \\ E_{p1} & E_{p-1,2} & \ldots & S_p \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{pmatrix} = \begin{pmatrix} g_1' \\ g_2' \\ \vdots \\ g_p' \end{pmatrix}, \tag{10}$$

where the off-diagonal matrices E_{ij} are available from the parallel distribution of the linear system. One preconditioning step with the Schur complement preconditioner consists in solving approximately the global system (10), and then recovering the u_i variables from the local equations as

$$u_i = B_i^{-1}[f_i - F_i y_i] \tag{11}$$

at the cost of one local solve. We solve the global system (10) by running a few steps of the GMRES method preconditioned by a block diagonal matrix, where the diagonal blocks are the local Schur complements S_i. The factorization

$$S_i = L_{S_i} U_{S_i}$$

is obtained as by-product of the LU factorization of the local matrix A_i,

$$A_i = \begin{pmatrix} L_{B_i} & 0 \\ E_i U_{B_i}^{-1} & L_{S_i} \end{pmatrix} \begin{pmatrix} U_{B_i} & L_{B_i}^{-1} F_i \\ 0 & U_{S_i} \end{pmatrix}.$$

which is by the way required to compute the u_i variables in (11).

4.1 Experiments

Some preliminary results with a parallel MPI-based implementation of VBARMS for distributed memory computers, reported in a conference contribution [7], revealed promising performance against the parallel ARMS method and the conventional ILUT method. They showed that exposing dense matrix blocks during the factorization may lead to more efficient and more stable parallel solvers. The parallel implementation of VBARMS considered in this study differs from the one presented in [7] in one important aspect. In the old implementation we used a sequential graph partitioner, namely the recursive dissection partitioner from the METIS package [12], to split the quotient graph $G/\mathcal{B}$ and then assign the computed partitions to different processors. In the new implementation, the quotient graph is initially distributed amongst the available processors; then, the built-in parallel hypergraph partitioner available in the Zoltan package [4] is applied on the distributed data structure to compute an optimal partitioning of the quotient graph that can minimize the amount of communications.
In the experiments reported in Table 8 we notice the significant reduction of CPU time spent for the graph partitioning operation in the new implementation of VBARMS; note that the numerical efficiency of the solvers is generally well preserved. The matrix problems used are listed in Table 7. The parallel experiments were run on the large-memory nodes (32 cores/node and 1TB of memory) of the TACC Stampede system located at the University of Texas at Austin. TACC Stampede is a 10 PFLOPS (PF) Dell Linux Cluster based on 6,400+ Dell PowerEdge server nodes, each outfitted with 2 Intel Xeon E5 (Sandy Bridge) processors and an Intel Xeon Phi Coprocessor (MIC Architecture). We linked the default vendor BLAS library, which is MKL. Although, MKL is multi-threaded by default, we used it in a single-thread mode since our MPI-based parallelisation employed one MPI process per core (communicating via the shared memory for the same-node cores). We used the Flexible GMRES (FGMRES) method [17] as Krylov subspace method, a tolerance of $1.0e-6$ in the stopping criterion and a maximum number of iteration equal to 1000. Memory costs were calculated as the ratio between the sum of the number of nonzeros in the local preconditioners, and the sum of the number of nonzeros in the local matrices A_i. Overall, the Restricted Additive Schwarz solver showed better performance against the Block Jacobi and the Schur-complement methods.

Name	Size	Application	nnz(A)
AUDIKW_1	943695	Structural problem	77651847
LDOOR	952203	Structural problem	42493817
STA004	891815	Fluid Dynamics	55902989
STA008	891815	Fluid Dynamics	55902989

Table 7: Set and characteristics of test matrix problems.

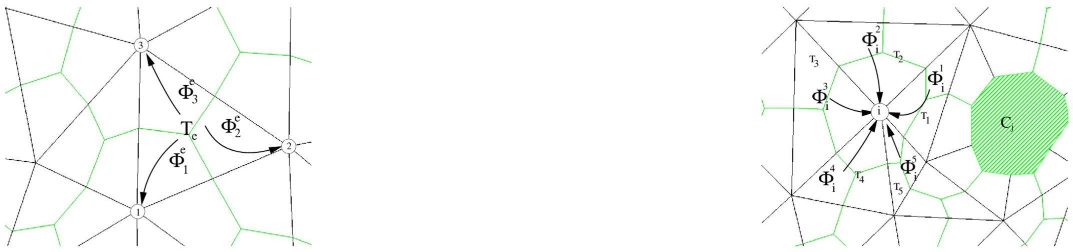
Matrix	Method	Graph type (s)	Graph time (s)	Factorization time (s)	Solving time (s)	Total time (s)	Its	Mem
AUDIKW_1	BJ+VBARMS	METIS (seq.)	54.5	18.88	51.35	70.23	136	3.13
		Zoltan (par.)	5.2	17.28	37.98	55.26	117	2.74
	RAS+VBARMS	METIS (seq.)	54.2	19.54	26.68	46.22	46	2.93
		Zoltan (par.)	5.3	22.75	22.24	44.99	52	2.87
	SCHUR+VBARMS	METIS (seq.)	54.4	82.72	295.11	377.83	69	6.21
		Zoltan (par.)	5.3	166.09	327.06	493.15	59	4.60
LDOOR	BJ+VBARMS	METIS (seq.)	30.0	1.29	25.10	26.40	345	1.95
		Zoltan (par.)	1.1	1.04	18.09	19.12	273	1.95
	RAS+VBARMS	METIS (seq.)	29.0	1.56	13.40	14.95	200	2.00
		Zoltan (par.)	1.1	1.12	12.73	13.85	196	1.99
	SCHUR+VBARMS	METIS (seq.)	29.0	5.81	16.75	22.56	54	3.63
		Zoltan (par.)	1.1	5.64	4.78	10.42	37	3.32
STA004	BJ+VBARMS	METIS (seq.)	79.4	7.53	42.56	50.08	90	3.61
		Zoltan (par.)	2.5	5.11	24.12	29.23	72	3.61
	RAS+VBARMS	METIS (seq.)	81.7	9.55	34.27	43.82	42	3.85
		Zoltan (par.)	2.6	7.90	23.09	30.99	34	3.31
	SCHUR+VBARMS	METIS (seq.)	81.4	17.46	135.58	153.04	90	5.29
		Zoltan (par.)	2.5	16.05	113.24	129.28	88	5.40
STA008	BJ+VBARMS	METIS (seq.)	81.9	11.36	85.77	97.14	227	4.77
		Zoltan (par.)	2.3	9.45	50.17	59.62	170	4.78
	RAS+VBARMS	METIS (seq.)	81.8	15.01	67.98	82.99	101	5.10
		Zoltan (par.)	2.4	12.90	46.52	59.42	97	5.07
	SCHUR+VBARMS	METIS (seq.)	81.2	56.20	564.75	620.94	188	8.94
		Zoltan (par.)	2.4	66.42	490.25	556.67	201	9.83

Table 8: Performance comparison of serial and parallel graph partition on 16 processors. We ran one MPI process per core, so in these experiments we used shared memory on a single node. Notation: P-N means number of processors, G-Type means graph partitioning strategy, G-time means partitioning timing cost, P-T means preconditioning construction time, I-T iterative solution time, Mem means memory costs.

4.2 A case study in large-scale turbulent flows analysis

We finally get back to the starting point that motivated this study. In this section we present a performance analysis with the parallel VBARMS implementation for solving large block structured linear systems arising from an implicit Newton-Krylov formulation of the Reynolds Averaged Navier Stokes (briefly, RANS) equations. Although explicit multigrid techniques have dominated the Computational Fluid Dynamics (CFD) arena for a long time, implicit methods based on Newton's rootfinding algorithm are recently receiving increasing attention because of their potential to converge in a very small number of iterations. One of the most recent outstanding examples on the use of implicit unstructured RANS CFD is provided in the article [25], which reports the turbulent analysis of the flow past three-dimensional wings using a vertex-based unstructured Newton-Krylov solvers. Practical implicit CFD solvers need to be combined with ad-hoc preconditioners to invert efficiently the large nonsymmetric linear system at each step of Newton's algorithm.

Throughout this section we use standard notation for the kinematic and thermodynamic variables: we denote by $\vec{u}$ the flow velocity, by ρ the density, p is the pressure, T is the temperature, e and h are respectively the specific total energy and enthalpy, ν is the laminar kinematic viscosity and $\tilde{\nu}$ is a scalar variable related to the turbulent eddy viscosity via a damping function. The quantity a denotes the sound speed or the square root of the artificial compressibility constant in case of the compressible, respectively incompressible, flow equations. In the case of high Reynolds number flows, we account for turbulence effects by the RANS equations that are obtained from the Navier-Stokes (NS) equations by means of a time averaging procedure. The RANS equations have the same structure as the NS equations with an additional term, the Reynolds' stress tensor, that accounts for the effects of the turbulent scales on the mean field. Using Boussinesq's approximation, the Reynolds' stress tensor is linked to the mean velocity gradient through the turbulent (or eddy) viscosity. In our study, the turbulent viscosity is modeled using the Spalart-Allmaras one-equation model [22]. The physical domain is partitioned into nonoverlapping control volumes drawn around each gridpoint by joining, in two space dimensions, the centroids of gravity of the surrounding cells with the midpoints of all the edges that connect that gridpoint with its nearest neighbors, as shown in Figure 2.



(a) The flux balance of cell T is scattered among its vertices.

(b) Gridpoint i gathers the fractions of cell residuals from the surrounding cells.

Figure 2: Residual distribution concept.

Given a control volume C_i, fixed in space and bounded by the control surface ∂C_i with inward normal $\vec{n}$, we write the governing conservation laws of mass, momentum, energy and turbulence transport equations as

$$\int_{C_i} \frac{\partial \vec{q}_i}{\partial t}\, dV = \oint_{\partial C_i} \vec{n} \cdot \vec{F}\, dS - \oint_{\partial C_i} \vec{n} \cdot \vec{G}\, dS + \int_{C_i} \vec{s}\, dV, \tag{12}$$

where we denote by $\vec{q}$ the vector of conserved variables. For compressible flows, we have $\vec{q} = (\rho, \rho e, \rho\vec{u}, \tilde{\nu})^T$, and for incompressible, constant density flows, $\vec{q} = (p, \vec{u}, \tilde{\nu})^T$. In (12), the vector operators $\vec{F}$ and $\vec{G}$ represent the inviscid and viscous fluxes, respectively. For compressible flows, we have

$$\vec{F} = \begin{pmatrix} \rho\vec{u} \\ \rho\vec{u}h \\ \rho\vec{u}\vec{u} + p\mathbf{I} \\ \tilde{\nu}\vec{u} \end{pmatrix}, \quad \vec{G} = \frac{1}{\mathrm{Re}_\infty} \begin{pmatrix} 0 \\ \vec{u} \cdot \tau + \nabla q \\ \tau \\ \frac{1}{\sigma}\left[(\nu + \tilde{\nu})\, \nabla\tilde{\nu}\right] \end{pmatrix},$$

and for incompressible, constant density flows,

$$\vec{F} = \begin{pmatrix} a^2\vec{u} \\ \vec{u}\vec{u} + p\mathbf{I} \\ \tilde{\nu}\vec{u} \end{pmatrix}, \quad \vec{G} = \frac{1}{\mathrm{Re}_\infty} \begin{pmatrix} 0 \\ \tau \\ \frac{1}{\sigma}\left[(\nu + \tilde{\nu})\, \nabla\tilde{\nu}\right] \end{pmatrix},$$

where τ is the Newtonian stress tensor. The source term vector $\vec{s}$ has a non-zero entry only in the row corresponding to the turbulence transport equation, which takes the form

$$c_{b1}\left[1-f_{t2}\right]\tilde{S}\tilde{\nu}+\frac{1}{\sigma Re}\left[c_{b2}\left(\nabla\tilde{\nu}\right)^2\right]+-\frac{1}{Re}\left[c_{w1}f_w-\frac{c_{b1}}{\kappa^2}f_{t2}\right]\left[\frac{\tilde{\nu}}{d}\right]^2. \tag{13}$$

For a description of the various functions and constants involved in (13) we refer the reader to [22].

We consider a fluctuation splitting approach to discretize in space the integral form of the governing equations (12) over each control volume C_i. The flux integral is evaluated over each triangle (or tetrahedron) in the mesh, and then split among its vertices [5] (see Figure 2), so that we may write from Eq. (12)

$$\int_{C_i}\frac{\partial\vec{q}_i}{\partial t}\,dV=\sum_{T\ni i}\vec{\phi}_i^T$$

where

$$\vec{\phi}^T=\oint_{\partial T}\vec{n}\cdot\vec{F}\,dS-\oint_{\partial T}\vec{n}\cdot\vec{G}\,dS+\int_T\vec{s}\,dV$$

is the flux balance evaluated over cell T and $\vec{\phi}_i^T$ is the fraction of cell residual scattered to vertex i. Upon discretization of the governing equations, we obtain a system of ordinary differential equations of the form

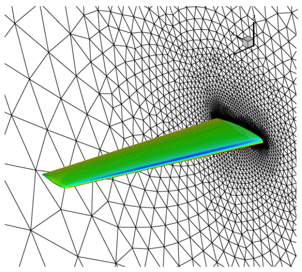
$$M\frac{d\vec{q}}{dt}=\vec{r}(\vec{q}), \tag{14}$$

where t denotes the pseudo time variable, M is the mass matrix and $\vec{r}(\vec{q})$ represents the nodal residual vector of spatial discretization operator, which vanishes at steady state. The residual vector is a (block) array of dimension equal to the number of meshpoints times the number of dependent variables, m; for a one-equation turbulence model, $m=d+3$ for compressible flows and $m=d+2$ for incompressible flows, d being the spatial dimension. If the time derivative in equation (14) is approximated using a two-point one-sided finite difference (FD) formula we obtain the following implicit scheme:

$$\left(\frac{1}{\Delta t^n}V-J\right)\left(\vec{q}^{n+1}-\vec{q}^n\right)=\vec{r}(\vec{q}^n), \tag{15}$$

where we denote by J the Jacobian of the residual $\dfrac{\partial\vec{r}}{\partial\vec{q}}$. We use a finite difference approximation of the Jacobian, where the individual entries of the vector of nodal unknowns are perturbed by a small amount ϵ and the nodal residual is then recomputed for the perturbed state. Eq. (15) represents a large nonsymmetric sparse linear system of equations to be solved at each pseudo-time step for the update of the vector of the conserved variables. The nonzero pattern of the sparse coefficient matrix is symmetric; on average, the number of non-zero (block) entries per row in our discretization scheme equals 7 in 2D and 14 in 3D. Choice of the iterative solver and of the preconditioner can have a strong influence on computational efficiency, especially when the mean flow and turbulence transport equations are solved in fully coupled form like we do.

We consider turbulent incompressible flow analysis past a three-dimensional wing illustrated in Fig. 3. The geometry, called DPW3 Wing-1, was proposed in the 3rd AIAA Drag Prediction Workshop [26]. Flow conditions are 0.5° angle of attack and Reynolds number based on the reference chord equal to $5\cdot10^6$. The freestream turbulent viscosity is set to 10% of its laminar value.
In Tables 9-10 we show experiments with parallel VBARMS on the five meshes of the DPW3 Wing-1 problem. We illustrate only examples with the parallel graph partitioning strategy described in Section 4. In Table 10 we report on only one experiment on the largest mesh, as this is a resource demanding



Ref. Area,	S = 290322 mm^2	= 450 in^2
Ref. Chord,	c = 197.556 mm	= 7.778 in
Ref. Span,	b = 1524 mm	= 60 in

RANS1 :	$n =$ 4918165	$nnz =$ 318370485
RANS2 :	$n =$ 4918165	$nnz =$ 318370485
RANS3 :	$n =$ 9032110	$nnz =$ 670075950
RANS4 :	$n =$ 12085410	$nnz =$ 893964000
RANS5 :	$n =$ 22384845	$nnz =$ 1659721325

Figure 3: Geometry and mesh characteristics of the DPW3 Wing-1 problem proposed in the 3rd AIAA Drag Prediction Workshop. Note that problems RANS1 and RANS2 correspond to the same mesh, and are generated at two different Newton steps.

problem. In Table 11 we perform a strong scalability study on the problem denoted as RANS2 by increasing the number of processors. Finally, in Table 12 we report on comparative results with parallel VBARMS against other popular solvers. The method denoted as `pARMS` is the solver described in [14], using default parameters. The results of our experiments confirm the same trend of performance shown on general problems. The proposed VBARMS method is remarkably efficient for solving block structured linear systems arising in applications in combination with conventional parallel global solvers such as in particular the Restricted Additive Schwarz preconditioner. A truly parallel implementation of the VBARMS method that may offer better numerical scalability will be considered as the next step of this research.

Matrix	Method	Graph time (s)	Factorization time (s)	Solving time (s)	Total time (s)	Its	Mem
RANS1	BJ+VBARMS	17.3	8.58	41.54	50.13	34	2.98
	RAS+VBARMS	17.4	10.08	42.28	52.37	19	3.06
	SCHUR+VBARMS	17.6	11.94	55.99	67.93	35	2.57
RANS2	BJ+VBARMS	17.0	16.72	70.14	86.86	47	4.35
	RAS+VBARMS	16.8	21.65	80.24	101.89	39	4.49
	SCHUR+VBARMS	17.5	168.85	173.54	342.39	24	6.47
RANS3	BJ+VBARMS	27.2	99.41	187.95	287.36	154	4.40
	RAS+VBARMS	25.2	119.32	90.47	209.79	71	4.48
	SCHUR+VBARMS	22.0	52.65	721.67	774.31	140	4.39

Table 9: Experiments on the DPW3 Wing-1 problem. The RANS1, RANS2 and RANS3 test cases are solved on 32 processors. We ran one MPI process per core, so in these experiments we used shared memory on a single node.

Matrix	Method	Graph time (s)	Factorization time (s)	Solving time (s)	Total time (s)	Its	Mem
RANS4	BJ+VBARMS	51.5	12.05	105.89	117.94	223	3.91
	RAS+VBARMS	43.9	14.05	91.53	105.58	143	4.12
	SCHUR+VBARMS	39.3	15.14	289.89	305.03	179	3.76
RANS5	RAS+VBARMS	1203.94$^{(1)}$	16.80	274.62	291.42	235	4.05

Table 10: Experiments on the DPW3 Wing-1 problem. The RANS4 and RANS5 test cases are solved on 128 processors. Note $^{(1)}$: due to a persistent problem with the Zoltan library on this run, we report on the result of our experiment with the Metis (sequential) graph partitioner.

Solver	Number of processors	Graph time (s)	Total time (s)	Its	Mem
RAS+VBARMS	8	38.9	388.37	27	5.70
	16	28.0	219.48	35	5.22
	32	17.0	101.49	39	4.49
	64	16.0	54.19	47	3.91
	128	18.2	28.59	55	3.39

Table 11: Strong scalability study on the RANS2 problem using parallel graph partitioning.

Matrix	Method	Factorization time (s)	Solving time (s)	Total time (s)	Its	Mem
RANS3	pARMS	-	-	-	-	6.63
	BJ+VBARMS	99.41	187.95	287.36	154	4.40
	BJ+VBILUT	20.45	8997.82	9018.27	979	13.81
RANS4	pARMS	-	-	-	-	5.38
	BJ+VBARMS	12.05	105.89	117.94	223	3.91
	BJ+VBILUT	1.16	295.20	296.35	472	5.26

Table 12: Experiments on the DPW3 Wing-1 problem. The RANS3 test case is solved on 32 processors and the RANS4 problem on 128 processors. The dash symbol $-$ in the table means that in the GMRES iteration the residual norm is very large and the program is aborted.

5 Conclusions

We have presented a parallel MPI-based implementation of a new variable block multilevel ILU factorization preconditioner for solving general nonsymmetric linear systems. One nice feature of the proposed solver is that it detects automatically exact or approximate dense structures in the coefficient matrix. It exploits this information to maximize computational efficiency. We have also introduced a modified compression algorithm that can find these approximate dense blocks structures, and requires only one simple to use parameter. The results show that the solver has nice parallel performance, also thanks to the use of a parallel graph partitioner, and it may be noticeably more robust than other state-of-the-art methods that do not exploit the fine-level block structure of the underlying matrix. The domain decomposition method used (BJ or RAS) is fundamentally not scalable as can be seen by the increase in iteration counts. A coarse grid correction would likely fix this. A truly parallel implementation of VBARMS without domain decomposition would be very interesting and and will be considered in a separate study. We have also tried to gain further parallelism using Many Integrated Codes (MIC) technology via the “MKL automatic MIC” approach but faced a lack of MIC memory in all our experiments on large problems (of around one million unknowns). Hence, significant algorithm adaptations for MIC technology may desirable, which constitute our future research.

6 Acknowledgements

The work of M. Sosonkina was supported in part by the Air Force Office of Scientific Research under the AFOSR award FA9550-12-1-0476, and by the National Science Foundation grants NSF/OCI0941434, 0904782, 1047772. The authors acknowledge the Texas Advanced Computing Center (TACC) at the University of Texas at Austin for providing HPC resources that have contributed to the research results reported in this paper. URL: `http://www.tacc.utexas.edu`. The authors are grateful to the reviewers for their insightful comments that helped much improve the presentation.

References

[1] C. Ashcraft. Compressed graphs and the minimum degree algorithm. *SIAM J. Scientific Computing*, 16(6):1404–1411, 1995.

[2] O. Axelsson and P. S. Vassilevski. Algebraic multilevel preconditioning methods, I. *Numer. Math.*, 56:157–177, 1989.

[3] O. Axelsson and P. S. Vassilevski. Algebraic multilevel preconditioning methods. II. *SIAM J. Numer. Anal.*, 27:1569–1590, 1990.

[4] Erik Boman, Karen Devine, Lee Ann Fisk, Robert Heaphy, Bruce Hendrickson, Vitus Leung, Courtenay Vaughan, Umit Catalyurek, Doruk Bozdag, and William Mitchell. Zoltan home page. `http://www.cs.sandia.gov/Zoltan`, 1999.

[5] A. Bonfiglioli. Fluctuation splitting schemes for the compressible and incompressible Euler and Navier-Stokes equations. *IJCFD*, 14:21–39, 2000.

[6] E.F.F. Botta, A. van der Ploeg, and F.W. Wubs. Nested grids ILU-decomposition (NGILU). *Journal of Computational and Applied Mathematics*, 66:515–526, 1996.

[7] B. Carpentieri, J. Liao, and M. Sosonkina. *Parallel Processing and Applied Mathematics*, volume 8385 of *Lecture Notes in Computer Science*, chapter Variable block multilevel iterative solution of general sparse linear systems, pages 520–530. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski. Springer-Verlag., 2014.

[8] B. Carpentieri, J. Liao, and M. Sosonkina. VBARMS: A variable block algebraic recursive multilevel solver for sparse linear systems. *Journal of Computational and Applied Mathematics*, 259 (A):164–173, 2014.

[9] J.J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16:1–17, 1990.

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

[11] A. Gupta and T. George. Adaptive techniques for improving the performance of incomplete factorization preconditioning. *SIAM J. Sci. Comput.*, 32(1):84–110, 2010.

[12] G. Karypis and V. Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices version 4.0. `http://glaros.dtc.umn.edu/gkhome/views/metis`. University of Minnesota, Department of Computer Science / Army HPC Research Center Minneapolis, MN 55455.

[13] Na Li, B. Suchomel, D. Osei-Kuffuor, and Y. Saad. ITSOL: iterative solvers package.

[14] Z. Li, Y. Saad, and M. Sosonkina. pARMS: a parallel version of the algebraic recursive multilevel solver. *Numerical Linear Algebra with Applications*, 10:485–509, 2003.

[15] J. O'Neil and D.B. Szyld. A block ordering method for sparse matrices. *SIAM J. Scientific and Statistical Computing*, 11(5):811–823, 1990.

[16] A. Quarteroni and A. Valli. *Domain decomposition methods for partial differential equations.* Clarendon Press Oxford, 1999.

[17] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Scientific and Statistical Computing*, 14:461–469, 1993.

[18] Y. Saad. ILUM: A multi-elimination ILU preconditioner for general sparse matrices. *SIAM J. Scientific Computing*, 17(4):830–847, 1996.

[19] Y. Saad. Finding exact and approximate block structures for ilu preconditioning. *SIAM J. Sci. Comput.*, 24(4):1107–1123, 2002.

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

[21] Y. Saad and B. Suchomel. ARMS: An algebraic recursive multilevel solver for general sparse linear systems. *Numerical Linear Algebra with Applications*, 9(5):359–378, 2002.

[22] P.R. Spalart and S.R. Allmaras. A one-equation turbulence model for aerodynamic flows. *La Recherche-Aerospatiale*, 1:5–21, 1994.

[23] N. Vannieuwenhoven and K. Meerbergen. IMF: An incomplete multifrontal LU-factorization for element-structured sparse linear systems. *SIAM J. Sci. Comput.*, 35(1):A270–A293, 2013.

[24] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, 2007.

[25] P. Wong and D. Zingg. Three-dimensional aerodynamic computations on. unstructured grids using a newton-krylov approach. *Computers & Fluids*, 37:107–120, 2008.

[26] Drag Prediction Workshop. URL:`http://aaac.larc.nasa.gov/tsab/cfdlarc/aiaa-dpw/Workshop3/workshop3.html`.