VBARMS: A variable block algebraic recursive multilevel solver for sparse linear systems

Liao, Jia

university of
groningen

VBARMS: A variable block algebraic recursive multilevel solver for sparse linear systems

PhD thesis

to obtain the degree of PhD at the
University of Groningen
on the authority of the
Rector Magnificus Prof. E. Sterken
and in accordance with
the decision by the College of Deans.

This thesis will be defended in public on

Friday 13 November 2015 at 16.15 hours

by

Jia Liao

born on 16 December 1985
in Sichuan, China

Supervisor
Prof. A.E.P. Veldman

Co-supervisor
Dr. B. Carpentieri

Assessment Committee
Prof. R.W.C.P. Verstappen
Prof. R.H. Bisseling
Prof. M. Bollhoefer

VBARMS: A variable block algebraic recursive multilevel solver for sparse linear systems

Jia Liao

ISBN: 978-90-367-8407-8 (printed version)
ISBN: 978-90-367-8406-1 (electronic version)

Contents

1 Introduction **1**
1.1 Motivation and background 1

2 Graph compression techniques **5**
2.1 Matrix partitioning methods 5
2.2 Experiments with angle-based compression method 11
2.3 A new graph-based compression method 13

3 Krylov-subspace methods and preconditioning **19**
3.1 Solving linear systems with Krylov-subspace methods 19
3.1.1 GMRES: The Generalized Minimum Residual Method 21
3.2 Preconditioning techniques 23
3.2.1 Incomplete LU factorization preconditioner 24
3.3 Multilevel Incomplete LU (ILU) decomposition solvers 27
3.3.1 ILUM: A Multi-elimination ILU preconditioner 27
3.3.2 BILUM: A block versions of the Multi-elimination and Multilevel ILU Preconditioner 31
3.3.3 ARMS: The Algebraic Recursive Multilevel Solver . . 33

4 The VBARMS solver **37**
4.1 Building blocks for VBARMS 37
4.2 The variable block ARMS factorization 41
4.2.1 The mathematical process of VBARMS 41
4.2.2 Computational aspects of the method 49
4.3 Numerical experiments on a sequential computer 52
4.3.1 The performance of VBARMS on block structured matrices 52
4.3.2 Experiments on unstructured matrices 58
4.3.3 Performance of the graph-based compression method . 60
4.3.4 Performance comparison of the two implementations . 63

5 Parallelization strategy and experiments **67**
5.1 Introduction to parallel computing 67
5.2 Parallel implementation of VBARMS 73
5.3 Parallel results with PVBARMS 79
5.3.1 Performance of VBARMS in a parallel package 79
5.3.2 A Zoltan-based graph partitioning strategy and experiments 88

6 Case study in large-scale turbulent flows simulation **95**
6.1 Definition of the problem 95
6.2 The results of the experiments 98

7 Concluding remarks **103**
7.1 Conclusions 103
7.2 Perspectives 106

Bibliography **107**

Summary **119**

Samenvatting **123**

Acknowledgements **127**

1 Introduction

1.1 Motivation and background

Sparse matrices arising from the numerical solution of systems of partial differential equations often exhibit a perfect block structure, meaning that the nonzero blocks in the sparsity pattern are fully dense (and typically small), e.g., when several unknown quantities are associated with the same grid point. Finite element and finite difference matrices have some degree of compression if there is more than one solution component at a 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, three components of the scaled velocity, and the turbulence transport variable) assigned to each node of the computational mesh; a bidomain system in cardiac electrical dynamics couples the intra-and extra-cellular electric potential at each ventricular cell of the heart; and so on. After numbering consecutively the ℓ distinct variables associated with the same grid point, the permuted matrix has a sparse block structure with nonzero blocks of size $\ell \times \ell$. The blocks are usually fully dense, as variables at the same node are mutually coupled. Blocking can be sometimes unravelled on general sparse unstructured matrices by numbering consecutively rows and columns having a resembling pattern, and treating some zero entries in the reordered matrix as nonzero elements, with a little sacrifice of memory.

Meanwhile, on today's emerging computer platforms, the costs of moving the data between fast and slow memory in the sequential case, or different processors in the parallel case, are decreasing at a much slower rate relatively to the costs of arithmetics. Minimizing the volume of these communications is the key to obtain good performance of numerical algorithms on modern cache-based architectures. Conventional linear algebra kernels for dense matrices can achieve computational rates near the theoretical peak by partitioning the matrix into small sub-blocks that fit the cache size, and rethinking the computation blockwise [33, 1]. Sparse codes are more diffi-

cult to optimize, as they typically perform only a few operations per datum. Significant algorithmic modifications and highly-tuned data structures may be required to exploit efficiently the sparsity of the matrix, and take full advantage of the hardware of current top-class computer systems.

By finding symbiotic relationships between dense and sparse computation, e.g. using dense matrix kernels in both assembly and elimination operations, abreast sparse direct codes can achieve high computational rates without incurring a significant increase in arithmetic operations for large scale realistic factorizations [34, p. 136]. Similar strategies are advocated for the iterative solutions of large linear systems arising from the discretization of three dimensional (3D) partial differential equations (PDEs). In this case, efficient direct solvers may be applied to solve either a *nearby* problem or a *local* problem defined on a sub-block of the matrix, or a sub-domain of the underlying physical mesh, sometimes increasing the size of the solvable system by an order of magnitude or two [35]. Computational experience indicates that block methods often show better performance than their pointwise analogues in the solution of many classes of 2D/3D PDEs (see e.g. [27, 29, 63, 12, 61, 4, 13]). As a rule of thumb, if the underlying physical problem has a natural block structure, it is often convenient to exploit this structure in the design of numerical algorithms.

In this thesis, we present a variable block algebraic recursive multilevel solver (called VBARMS) that takes advantage of these frequently occurring structures in the iterative solution. The VBARMS method detects automatically existing exact or approximate dense structures in the coefficient matrix without any users prior knowledge of the underlying problem, achieving improved reliability and increased throughput during the computation on realistic applications. We review and compare different block ordering techniques and we introduce a novel graph compression algorithm to find approximate dense blocks structures, which requires only one simple to use parameter. As implementation details are always critical aspects to consider in the design of sparse matrix algorithms, we present two implementation strategies of the partial (block) factorization step and compared their performance.

Moreover, we also develop a novel parallel MPI-based implementation of VBARMS (called pVBARMS) for distributed memory computers based on the block Jacobi, the additive Schwarz and the Schur-complement methods. We propose a study of the numerical and parallel scalability of the

pVBARMS method on a set of general linear systems arising from different application fields. A new graph partitioning strategy is also proposed to optimize the distribution of the matrix in a parallel setting.

Finally, we assess the performance of the pVBARMS method for solving the Navier-Stokes equations on a suite of two- and three-dimensional test cases, among which the calculation of the turbulent flow past the DPW3-W1 wing configuration of the third AIAA Drag Prediction Workshop, which is the application that motivated this study. The choice of linear solver and preconditioner has a substantial effect on efficiency when the mean flow and turbulence transport equations are solved in fully coupled form, like we do. Our analyses in this thesis are carried out with coarse to medium-sized grids featuring up to 2.5 million nodes at Reynolds number equal to $5 \cdot 10^6$.

The thesis is organized as follows[1]. In Chapter 2 we recall the classic angle-based compression algorithm and present a novel graph-based compression method for computing a suitable block ordering for a general sparse matrix. In Chapter 3 we review some necessary background on Krylov subspace methods and preconditioning technique for solving sparse linear systems with a special focus on ILU and ILU based multi-level preconditioners. In Chapter 4 we outline the main computational steps of the VBARMS method and we illustrate the performance of VBARMS for solving a large set of matrix problems arising from various applications. In the last section, we also present comparative experiments on the performance between two different implementations of VBARMS and also the comparison of the performances of angle-based compression algorithm and our novel graph-based compression method. In Chapter 5 we move to parallel computing; first, we introduce the basics which provides the framework that we use, and then we discuss the parallel MPI-based implementation of the VBARMS code. Later, we illustrate the parallel performance of VBARMS and numerical and parallel scalability results. Finally, in Chapter 6 we test the performance of the new solver on large block structured linear systems arising from Computational Fluid Dynamics applications.

[1]Parts of the material presented have been published in journal papers [22, 24] and conference proceedings [23, 20, 55, 56, 21].

2 Graph compression techniques

2.1 Matrix partitioning methods

Block iterative methods are attractive to use since they often show better convergence rates and faster timings than their pointwise analogues in the numerical solution of many classes of two- and three-dimensional partial differential equations (PDEs). For PDEs discretized on regular cartesian grids, a regular partition of the domain may provide an effective partitioning for the matrix. E.g., for Poisson's equation defined on a rectangle $(0, \ell_1) \times (0, \ell_2)$ with Dirichlet boundary conditions, discretized uniformly by taking $n_1 + 2$ points in the interval $(0, \ell_1)$ and $n_2 + 2$ points in the interval $(0, \ell_2)$, after numbering the interior points in the *natural ordering* from the bottom up, one horizontal line at a time, one obtains a $n_2 \times n_2$ block tridiagonal structure with square blocks having size $n_1 \times n_1$. The diagonal blocks are tridiagonal and the off-diagonal blocks are diagonal matrices. For large finite element models, an obvious way to block the matrix is to use substructuring, since each substructure of the underlying physical mesh corresponds to one sparse block of the system. If the domain is highly irregular, or if the matrix does not correspond to a differential equation, finding the best block partitioning strategy is much less obvious. Several recent studies have shown the importance of exposing dense blocks during the factorization for achieving better performance, see e.g. [22, 85, 43, 69].

In cases where no good partitioning of the matrix is known to the user, graph reordering techniques are worth considering. The PArameterized BLock Ordering (PABLO) algorithm proposed by O'Neil and Szyld is one of the first graph partitioning algorithm especially designed for solving general linear systems by block iterative methods [64]. The algorithm traverses the adjacency graph of the matrix and selects groups of nodes so that the corresponding diagonal blocks are either full or very dense, CuthillMcKee algorithm also improves the density of the diagonal blocks [30]. Classical block stationary iterative methods such as block Gauss-Seidel and SOR methods combined with the ordering provided by PABLO require fewer operations

than the point analogues for the finite element discretization of a Dirichlet problem on a graded L-shaped region, and on the 9-point discretization of the Laplacian operator on a square grid compared to the natural partitions of the grid. The complexity of the PABLO algorithm is proportional to the number of nodes and edges, *i.e.* the number of nonzeros in the matrix, in both time and space.

One of the first compression methods especially designed to discover dense blocks in a matrix was proposed by Ashcraft with the achieved objective of reducing the ordering time of the minimum degree algorithm [3]. Ashcraft's method searches for sets of rows or columns of a matrix A having the exact same pattern. In graph terminology, it looks for vertices u and v of the adjacency graph (V, E) of A having the same adjacency list, that is $adj(u) = adj(v)$. Such nodes are also called *indistinguishable nodes*. The algorithm assigns a *checksum* quantity to each vertex, e.g., using the function

$$chk(u) = \sum_{(u,w)\in E} w, \tag{2.1}$$

and then sorts the vertices by their checksums. This operation takes $|E| + |V| \log |V|$ time. If u and v are indistinguishable, then

$$chk(u) \equiv \sum_{(u,w)\in E} w = chk(v) \equiv \sum_{(v,w)\in E} w.$$

Therefore, after sorting the vertices by their checksums, nodes having the same checksum are examined. If $|adj(u)| = |adj(v)|$, then $adj(u)$ and $adj(v)$ are explicitly compared to see if u and v are indeed indistinguishable. The ideal checksum function would assign a different value for each different row pattern that occurs, so that there is no need to compare patterns. Such a perfect checksum function is not practical, though, because it leads 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 solving the system, simple checksum functions such as equation 2.1 are used in practice [3]. We recall the steps of the checksum algorithm in Algorithm 2.1. We use the following notations: $K(i)$ is the checksum key for row i, $K(u).key$ is the key value and $K(i).row$ is the row number. $Group(i) = k$ means row i belongs to the group of row k, $Group(i) = -1$ means the row i is not selected yet.

The first step of Algorithm 2.1 calculates the checksum key for all rows, and then it targets at one row and loops over the subsequent rows. If the

checksum key are equal, then the pattern of the rows will be compared exactly; if they are the same, add the row to the group.

Algorithm 2.1 Checksum algorithm.

Input: pattern matrix C
Output: set data structure for blocks
1: Initialize $Group(i) = -1$ for $i, \ldots, n$
2: Compute all the keys of $K(u)$ according to Eq. (2.1)
3: Sort the array $K(u)$ in increasing $K(u).key$
4: **for** $i = 1, 2, \ldots, n$ **do**
5: $row\_target = K(i).row; key\_target = K(i).key$
6: **for** $j = i + 1, \ldots, n$ **do**
7: $row\_new = K(j).row; key\_new = K(j).key.$
8: **if** $key\_new \neq key\_target$ **then**
9: break.
10: **else**
11: **if** $Group(i) = -1$ and $pattern(row\_new) == pattern(row\_target)$ **then**
12: $Group(row\_new) = row\_target$

Suppose now that the structurally symmetric matrix has an imperfect block structure (imperfect block structure means there are some zero entires in the nonzero blocks that are treated as nonzero entries). A simple example of this situation is represented in Figure 2.1. As we described before, the checksum algorithm only detects the rows having exactly the same pattern; in this case, it will discover three nontrivial blocks on the left matrix (B_1) and only one nontrivial diagonal block on the right matrix (B_2), which is obtained by zeroing out the entry $B_1(7, 1)$ and, for symmetry, also $B_1(1, 7)$. Clearly, we would prefer to apply the block structure of B_1 to B_2, by treating $B_2(1, 7)$ and $B_2(7, 1)$ as nonzeros.

Sparse matrices with a relatively large number of nonzero elements per row often show *approximate dense structures*, consisting mostly of nonzero entries and only a few zeros. In this case the zeros in the blocks can be treated as nonzero elements, with a little sacrifice of memory, and a more efficient ordering may be generated for an iterative solver. These approximate dense blocks can be discovered by numbering consecutively matrix rows and columns having a similar sparsity pattern. However, extending the checksum-based algorithm to handle this case would require to define

$$B_1 = \left[\begin{array}{cc|ccc|cc} * & * & 0 & 0 & 0 & * & * \\ * & * & 0 & 0 & 0 & * & * \\ \hline 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 \\ \hline * & * & 0 & 0 & 0 & * & * \\ * & * & 0 & 0 & 0 & * & * \end{array}\right] \qquad B_2 = \left[\begin{array}{cc|ccc|c|c} * & * & 0 & 0 & 0 & * & 0 \\ \hline * & * & 0 & 0 & 0 & * & * \\ \hline 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 \\ \hline * & * & 0 & 0 & 0 & * & * \\ \hline 0 & * & 0 & 0 & 0 & * & * \end{array}\right]$$

Figure 2.1: Two examples of structurally symmetric matrices with a perfect (on the left) and imperfect (on the right) block structure. We denote by the symbol "*" a nonzero entry and by solid lines the block partitioning found by the checksum algorithm in both cases.

a new checksum function that preserves the proximity of patterns, in the sense that close patterns will result in close checksum values. This usually does not hold for Ashcraft's algorithm, at least in its original form.

Alternatively, Saad proposed in [69] to compare angles of rows (or columns) of a matrix A to find approximate dense structures in the pattern of A. Let C be the pattern matrix of A, having the same pattern as A and whose nonzero values are equal to one. The idea of the method proposed by Saad is to compute the upper triangular part of each row i of CC^T. Entry (i, j) in this row is the inner product, or cosine value, between row i and row j only for $j > i$. If the cosine value is big enough, hence the corresponding angle, is small enough, row j will be added to the same group of row i. The operation is repeated for $i = 1, \ldots, n$. Although it may appear expensive to compare all the rows of a matrix with each other, as the algorithm progresses many rows may already have been assigned so that the comparison can be skipped leading to substantial savings. A first pass with the checksum-based algorithm to detect any "exact" block structure may facilitate the search, as the angle algorithm can be performed on the quotient graph which is typically smaller. Then in the second pass the algorithm scans each non-assigned row again to determine whether it can be added to an existing group. That is the so-called *Hybrid algorithm*, we also recall it in Algorithm 2.2, further details are found in [69]. 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 still remains lower than the cost of the least expensive block LU factorization [31, 59], i.e., block ILU(0).

Algorithm 2.2 Hybrid algorithm.

Input: pattern matrix C and tolerance τ
Output: set data structure for blocks

```
1: Run algorithm 2.1 to get an initial blocking Group_0. Set Group_0 = Group
2: for i = 1, 2, ..., n do
3:   if Group(i) == −1 then
4:     for {j|c_ij ≠ 0} do
5:       row = jth_row; s = |Group_0(j)|.
6:       for k = nnz_j, nnz_j − 1, ..., 1 do
7:         col = row(k)
8:         if col < i then
9:           break.
10:        else
11:          if Group(col) == −1 then
12:            Count(col) = Count(col) + s.
13:    for {col|Count(col) ≠ 0} do
14:      if Count(col)^2 > τ * nnz_i * nnz_col then
15:        Group(col) == i; update the size of Group(i)
16:      Count(col) = 0.
```

A few notations used Algorithm 2.2 have to be explained here: $Group_0$ is the initial blocking information computed by the checksum algorithm, $Group$ is the current blocking information, being updated during the procedure. Their entries' value equal -1 means the current row is the reference row of a group, $|Group_0(j)|$ is the number of rows in this group, nnz_j denotes the number of nonzero elements in the jth row.

There are two main steps of Algorithm 2.2: one is the for loop at line 4, it targets one reference row, and from bottom to top loops over all the subsequent rows; during the loop, it counts the number of same column indexes for each row, and uses an array $Count$ to store the number, $Count(col)$ is the number of same column indexes between row i and row col. The loop at line 13 just simply traverses $Count$ and calculates the angle between the

current row and the reference row, then adds the current row to the reference row group if the angle is small enough. τ is a parameter applied by the user, defining the maximum allowed angle in the calculation used for merging two rows. $\tau \in [0, 1]$. τ's value closer to 1 means the two rows' pattern are closer.

2.2 Experiments with angle-based compression method

Next we want to show how to use the block ordering described in Algorithm 2.2 to determine the block structure of general matrices.

We collected 13 matrix problems arising from different applications, they are from University of Florida sparse matrix collection. In Table 2.1 we report the size, application field, number of nonzero entries and percentage of row/column diagonal dominance of the coefficient matrix.

Table 2.1: Set and characteristics of test matrix problems.

Name	Size	Application	nnz(A)	row/colum diag. dom.
RAE	52995	Turbulence analysis	1748266	2.95/6.02
STACOM	8415	Compressible flow	271936	3.01/8.75
BCSSTK35	30237	Automobile seat frame	1450163	1.29/1.29
BMW7ST	141347	Car body	7318399	0.32/0.32
CT20STIF	52329	Engine block	2600295	2.05/2.05
K3PLATES	11107	acoustics problem	378927	0.00/0.00
NASASRB	54870	Shuttle rocket booster	2677324	0.94/0.94
OILPAN	73752	Structural problem	2148558	23.67/23.67
OLAFU	16146	structural problem	1015156	0.12/0.12
PWTK	217918	Pressurized wind tunnel	11524432	0.42/0.42
RAEFSKY3	21200	Fluid structure interaction	1488768	27.64/27.62
S3DKQ4M2	90449	Finite element analysis	4427725	0.01/0.01
VENKAT01	62424	2D Euler solver	1717792	0.00/0.00

In Table 2.2 we report on the characteristics of the block ordering computed by Algorithm 2.2. Before we start, some notations from Table 2.2 have to be explained. We already introduced the parameter τ. The column *b-size* shows the average block size of A after the compression, and the column *b-density* shows the ratio of the number of nonzero entries in A before and after the compression. It is *b-density* $= 1$ if the graph compression algorithm finds a perfect block structure in A with fully dense nonzero blocks, whereas *b-density* < 1 means that some zero entries in the blocks are treated as nonzeros, regardless of their actual numerical value.

In our experiments, we initially set $\tau = 1$ to find sets of rows and columns having the same pattern and discover the presence of fully dense blocks in

the matrix. The results are columns 2-4 of Table 2.2. For these results, we do not include any extra zero entries into blocks, so the *b-density* is 100%, The inherent block structure of the matrix can be detected. This is what we call perfect blocking. However, we can also try to use a smaller value of τ to enlarge the blocks by padding some zero entries.

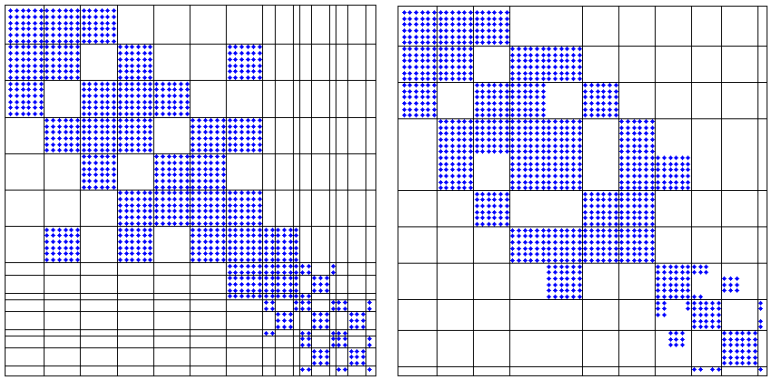
Table 2.2: Block structure of test matrix problems, the highlighted matrices are the ones that gained most from the τ value tuning.

Name	τ	b-size	b-density (%)	τ	b-size	b-density (%)
RAE	1.00	4.00	96.89	0.80	4.67	95.83
STACOM	1.00	4.11	97.10	0.80	4.36	95.97
BCSSTK35	1.00	4.57	100.00	0.90	5.07	99.29
BMW7ST	1.00	4.63	100.00	0.90	5.28	99.24
CT20STIF	1.00	2.61	100.00	0.90	3.47	96.61
K3PLATES	1.00	5.02	100.00	1.00	5.02	100.00
NASASRB	1.00	2.20	100.00	0.90	3.31	92.31
OILPAN	1.00	2.45	100.00	0.80	2.63	99.73
OLAFU	1.00	1.54	100.00	0.90	5.10	89.50
PWTK	1.00	4.67	100.00	0.90	5.48	99.04
RAEFSKY3	1.00	8.00	100.00	1.00	8.00	100.00
S3DKQ4M2	1.00	1.25	100.00	0.70	5.93	90.34
VENKAT01	1.00	4.00	100.00	1.00	4.00	100.00

Fig. 2.2 shows the difference between perfect and imperfect blocking on one small matrix sample.

The compression algorithm exposed any existing (exact) block structure fast and efficiently, without requiring any prior knowledge of the problem. Some matrices were not detected as block matrices, e.g., the *b-size* parameter was approximately one for the S3DKQ4M2 and the OLAFU problems. Choosing a different value for τ leaves the freedom to relax the similarity pattern requirement in the rows/columns comparison, and enlarge the nonzero blocks by treating some zero entries as nonzeros.

We tested different values for τ, ranging from 0.7 to 1 on these two problems; with very little sacrifice in memory, it was possible to obtain larger blocks with still high density around 90%. By slightly decreasing the value



Perfect blocking. Imperfect blocking.

Figure 2.2: An example of perfect and imperfect blocking computed by the angle algorithm on a small sparse matrix using two different values of τ. We can see in the right figure that large blocks may be found by treating some zero entries as nonzeros.

of the dropping tolerance τ, we could increase the block size also for other problems, like CT20STIF, NASASRB, OLAFU, S3DKQ4M2, as it is shown in Table 2.2.

2.3 A new graph-based compression method [1]

The angle-based compression depends on a parameter τ which determines the proximity of row (or column) patterns. If the cosine of the angle between rows i and j is larger than τ, row j will be added to the group of row i. For $\tau = 1$, the method computes perfect dense blocks. Values of τ smaller than one may produce larger blocks with some zeros entries padded in the pattern. The use of approximate dense structures in the iterative solution may clearly speed up BLAS3 operations, but also increases memory costs and the probability to encounter singular blocks during the factorization [22]. Therefore, tuning τ may be critical for performance. Finding the best quality ordering, which minimizes the total solution time on a given problem, may require performing several runs. For example, the *b-density*

[1] This method is proposed in [24]

value may be sensitive to τ and much dependent on the matrix structure. In the experiments reported in Table 2.3, we can see that a value of $\tau = 0.58$ returns a *b-density* of 86.37% for the VENKAT01 matrix and a *b-density* of 45.06% for the STACOM matrix. In our experiments we found that values of $\tau \approx 0.6$ are particularly critical.

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 2.3: *b-density* (%) from the angle compression algorithm for different values of τ.

Due to these problems, we have revisited Saad's angle-based blocking method and we have developed a new compression algorithm that computes an ordering having an average *b-density* not smaller than a user-specified value μ. The method works with the quotient graph $G/\mathcal{B} = (V_{\mathcal{B}}, E_{\mathcal{B}})$. After doing a first pass with the checksum-based Algorithm 2.1 to detect any "exact" block structure in the matrix, it proceeds by merging nodes of $V_{\mathcal{B}}$, also called supernodes or supervertices, provided that the *b-density* after this operation does not drop below μ. Candidate supernodes for merging are those having similar adjacent sets in V, that is supernodes Y and Z such that $adj(Y) \cap adj(Z)$ is largest, where we define the adjacency set of a supernode Y as

$$adj(Y) = \bigcup_{y \in Y} adj(y).$$

The rationale is to minimize the number of extra zeros padded after merging the two blocks. Therefore the method calculates the *b-density* of $Y \cup Z$ before actually merging Y and Z. If this quantity is larger or equal than μ, the

operation is performed and a new merging is attempted. Otherwise, the algorithm will stop.

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 supernode minus the square block on the diagonal which we count twice since we count both columns and rows. The number of 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 supernode minus the amount of nodes inside the diagonal block, which is again counted twice. Algorithm 2.3 shows more details.

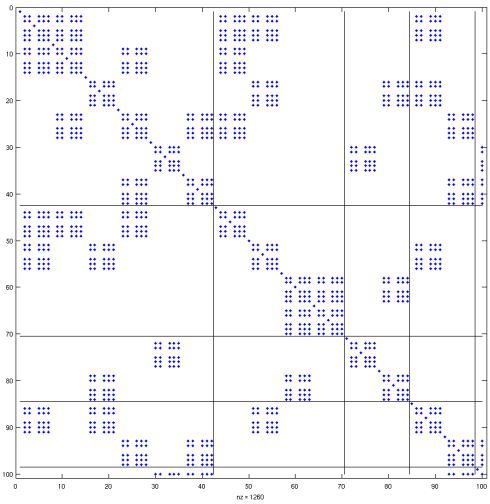
The for loop at line 5 also does the first pass of Hybrid algorithm, it generates an initial block structure. The loop at line 15 performs the attempt to merge the two blocks, it loops over the blocks (supernodes) generated by the first, for each block, it traverses his neighbors and calculates the block density after the merging, and then decides merging the two blocks or not.

Algorithm 2.3 *Graph based compression algorithm.*

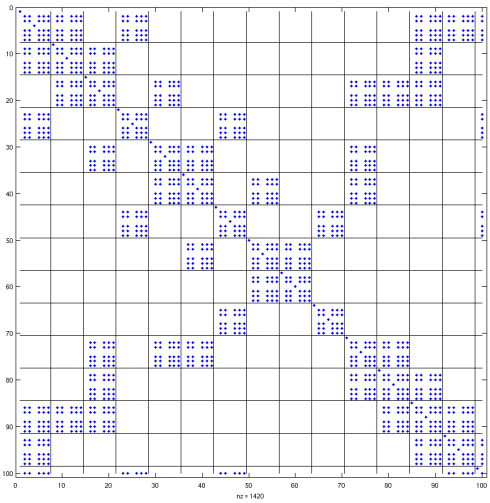
```
1: Compute the keys k_i = chk(i) for all vertices i ∈ V = {1, ..., n}
2: Set processed nodes p_i = 0 ∀i = 1, ..., n
3: Make a set of supernodes V = ∅
4: Set s to the indices V sorted by the corresponding value in k
5: for i = s_1, ..., s_n do
6:    if p_i ≠ 1 then
7:       Add a new supernode Y_i to V
8:       for j = s_{i+1}, ..., s_n do
9:          if k_i ≠ 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 M : i ↦ {Z ∈ V| i ∈ adj(Z)}
15: for X ∈ V do
16:    for Z ∈ ⋃_{i∈X} M(i) do
17:       if b-density ≥ μ then
18:          X = X ∪ Z
19:          V = V\Z
```

The graph based algorithm depends on a parameter μ that is simple to use. The output is an ordering with blocks having a minimum density value of μ. For example, if we desire a *b-density* of around 60%, we simply set $\mu = 0.6$ for every problem. In contrast, the *b-density* calculated by the angle-based compression is an averaged value. This means that on highly irregularly structured matrices, for some combinations of τ the computed orderings may return some very sparse large blocks in addition to the dense blocks. On the OILPAN matrix, using $\tau = 0.6$, we obtained a block ordering having an average density of 70% but some large blocks were only around 20% dense (a region of this pattern is illustrated in Figure 2.3(a)). Therefore a correct tuning of τ may require to run the full solver to see if a singular block is encountered during the factorization. This problem is much less likely to occur with the proposed graph-based compression algorithm. Section 4.3.3 shows detailed results of the performance of graph-based compression and the comparison with angle-based compression algorithm.

In this chapter, we recalled the angle-based compression algorithm and introduced our new graph-based compression algorithm. These two methods enable us to build a variable block structured matrix from the original pointwise matrix, so using block solvers becomes possible and developing new and more powerful block solver becomes necessary.



(a) $\tau = 0.6$



(b) $\tau = 1.0$

Figure 2.3: Block pattern of the OILPAN matrix computed by the angle-based compression method [24].

3 Krylov-subspace methods and preconditioning

Before we introduce our new VBARMS solver, we would like to recall the famous Krylov-subspace methods and preconditioning techniques for linear systems which are the prerequisites for understanding VBARMS.

3.1 Solving linear systems with Krylov-subspace methods

In many numerical simulations in science and engineering, solving the linear systems:

$$Ax = b, \tag{3.1}$$

where A is an large and sparse matrix and b is a given right-hand side vector, is often the most time-consuming phase.

There are two popular types of methods for solving system (3.1), direct methods and iterative methods. Direct methods [36] are based on the Gaussian Elimination (GE) algorithm applied to the coefficient matrix A. Direct solvers are very robust, but their computational cost is high. Moreover, they also have very poor scalability in terms of operations and memory cost, especially on matrices derived from 3D PDEs (see Chapter 6 of [34]).

The idea behind the basic iterative methods is to split the matrix A into the sum of two matrices, one of which is easy to invert. For example, the well-known Richardson iteration is based on the matrix splitting $A = I - (I - A)$:

$$x^i = b + (I - A)x^{i-1} = x^{i-1} + r^{i-1}, \quad r^{i-1} = b - Ax^{i-1} \tag{3.2}$$

In contrast to direct solvers, normally, iterative methods require less memory cost and operation, especially when a high accuracy or absolute accurate solution is not required. However, they are less robust than direct methods.

Normally, iterative methods involve improving the approximate solution from one iterate to the next and updating a few components to achieve a

better approximation like Eq. (3.2). The popular iterative methods are basic iterative methods like the classical Jacobi, Gauss-Seidel, and Successive Over-Relaxation (SOR) method, see Chapter 4 of [70]. However, starting from the mid-1970s, Krylov subspace methods got more and more popular. The focus of this section is on Krylov subspace methods and basic preconditioners.

The idea of the Krylov subspace methods is to search a solution in the Krylov subspace Eq. (3.4), because the solution to a nonsingular linear system lies in a Krylov space, see [48].

To introduce Krylov subspace methods, first we recall some basic concepts; see Chapter 6 of [70]. A general projection method seeks an approximate solution x in the affine subspace $x_0 + \mathcal{K}_m$, so that the residual satisfies:

$$r_m = b - Ax_m \perp \mathcal{L}_m \tag{3.3}$$

$\mathcal{L}_m$ is a subspace of dimension m, x_0 is the initial guess to the solution. This method is a Krylov subspace methods if the subspace $\mathcal{K}_m$ is the Krylov subspace

$$\mathcal{K}_m(A, r_0) = span\{r_0, Ar_0, A^2r_0, \ldots, A^{m-1}r_0\} \tag{3.4}$$

There are different categories of Krylov subspace methods depending on the choice of the subspace $\mathcal{L}_m$.

1. Orthogonal projection methods

   This corresponds to the choice $\mathcal{L}_m = \mathcal{K}_m(A, r_0)$. Krylov subspace methods belonging to this class are the full Orthogonalization Method (FOM) [70] for general non-symmetric matrices, and the Conjugate Gradient (CG) [47] method for symmetric, positive and definite (SPD) matrices. CG is widely used in scientific computing. A combination of CG and preconditioner (see Section 3.2) maybe considered as the method of choice for solving large SPD sparse linear systems.

2. Orthogonal methods

   This corresponds to the choice $\mathcal{L}_m = A\mathcal{K}_m(A, r_0)$. The Generalized Minimum Residual Method (GMRES) [72, 84] based on the Arnoldi process [2] is the most popular algorithm in this class. Such a technique minimizes the residual norm over all vectors in $x_0 + \mathcal{K}_m$. Moreover, GMRES has very good numerical stability. Several variants of GMRES

have been developed, especially to try to reduce the computational and memory cost of the original algorithm, like Restarted GMRES, Quasi-GMRES and DQGMRES [70].

3. Bi-orthogonalization methods

   This corresponds to the choice $\mathcal{L}_m = \mathcal{K}_m(A^T, r_0)$. The Bi-orthogonal version of CG (BiCG) method belongs to this class. One drawback of BiCG is that each step of the BiCG method requires a matrix-vector product with both A and A^T. Later research developed transpose free variants of BiCG, such as the Conjugate Gradient Squared (CGS) algorithm developed by Sonneveld in 1984 [78]. The CGS method sometimes exhibits faster convergence than BiCG for roughly the same computational and memory cost.

   The CGS algorithm squares the residual polynomial, and this may give rise to highly irregular residual norm convergence, and accumulation of rounding errors in some cases. The Biconjugate Gradient Stabilized (BICGSTAB) [83] method was developed to amend this.

4. Normal equation methods

   The choice $\mathcal{L}_m = \mathcal{K}_m(A^T A, A^T r_0)$, corresponds to apply CG to the normal equations $A^T Ax = A^T b$, methods in this class are CGNR (N for "normal" and R for "Residual") and CGNE (N for "normal" and E for "Error").

3.1.1 GMRES: The Generalized Minimum Residual Method

We will briefly recall the procedure of the GMRES method, since it is the one we use in our numerical experiments.

The GMRES is an algorithm based on the Arnoldi process [2]. It computes the orthonormal basis of the Krylov subspace as follows: start with $v_1 = r_0/\|r_0\|_2$ (step 1 of Algorithm 3.1). Assuming that we already have an orthonormal basis $v_1, v_2, \ldots, v_j$ for subspace $\mathcal{K}_j(A, r_0)$. At step j the algorithm implements a modified Gram-Schmidt procedure [42] to find the next basis vector v_{j+1}. It computes $w_j = Av_j$ (line 4 of Algorithm 3.1), then it orthonormalizes w_j with respect to $v_1, v_2, \ldots, v_j$ (line 5-7 and 11 of Algorithm 3.1).

It is obvious that $v_1, v_2, \ldots, v_m$ forms an orthonormal basis of $\mathcal{K}_m(A, r_0)$. If we denote by V_m the matrix with columns $v_1, v_2, \ldots, v_m$, then we have

$$AV_m = V_{m+1}H_{m+1,m} \tag{3.5}$$

where $H_{m+1,m}$ is a $m+1$ by m Hessenberg matrix and its entries $h_{i,j}$ are defined in Algorithm 3.1. Since any vector x in $x_0 + \mathcal{K}_m$ can be written as

$$x = x_0 + V_m y \tag{3.6}$$

with y the m-vector of coefficients of the linear expansion. Eq. (3.5) leads to

$$\begin{aligned} b - Ax &= b - A(x_0 + V_m y) \\ &= r_0 - AV_m y \\ &= \beta v_1 - V_{m+1}H_{m+1,m}y \\ &= V_{m+1}(\beta e_1 - H_{m+1,m}y) \end{aligned} \tag{3.7}$$

The approximation computed by GMRES is the unique vector of the Krylov space $x_0 + \mathcal{K}_m$ that minimizes Eq. (3.7). The minimizer y_m is cheap to compute because it solves a small $(m+1) \times m$ least-square problem.

Algorithm 3.1 GMRES

1: Choose x_0, $r_0 = b - Ax_0$ and $v_1 = r_0/\|r_0\|_2$. $\beta = \|r_0\|_2$
2: Define the $(m+1) \times m$ matrix $H_{m+1,m} = \{h_{ij}\}$. Set $H_{m+1,m} = 0$
3: **for** $j = 1, 2, \ldots, m$ **do**
4: $\quad w_j = Av_j$
5: $\quad$ **for** $i = 1, 2, \ldots, j$ **do**
6: $\quad\quad h_{ij} = (w_j, v_i)$
7: $\quad\quad w_j = w_j - h_{ij}v_i$
8: $\quad h_{j+1,j} = \|w_j\|_2$,
9: $\quad$ **if** $h_{j+1,j} = 0$ **then**
10: $\quad\quad$ goto step 12
11: $\quad v_{j+1} = w_j/h_{j+1,j}$
12: $x_m = x_0 + V_m y_m$, where y_m minimizes $\|\beta e_1 - H_{m+1,m}y\|_2$

But in the practical implementation, when m increases, each new iteration costs more operation counts and memory than the one before it. To improve this, we can restart the algorithm at every m steps, using the approximate solution x_m as initial guess for a new GMRES process continuing

and repeating this process until convergence. This idea leads to the restarted GMRES method sketched in Algorithm 3.2.

Algorithm 3.2 GMRES(m), also called restarted GMRES

1: Choose x_0, $r_0 = b - Ax_0$ and $v_1 = r_0/\|r_0\|_2$. $\beta = \|r_0\|_2$
2: Define the $(m+1) \times m$ matrix $H_{m+1,m} = \{h_{ij}\}$. Set $H_{m+1,m} = 0$
3: **for** $j = 1, 2, \ldots, m$ **do**
4: $w_j = Av_j$
5: **for** $i = 1, 2, \ldots, j$ **do**
6: $h_{ij} = (w_j, v_i)$
7: $w_j = w_j - h_{ij}v_i$
8: $h_{j+1,j} = \|w_j\|_2$,
9: **if** $h_{j+1,j} = 0$ **then**
10: goto step 12
11: $v_{j+1} = w_j/h_{j+1,j}$
12: $x_m = x_0 + V_m y_m$, where y_m minimizes $\|\beta e_1 - H_{m+1,m}y\|_2$
13: **Restart:**
14: $r_m = b - Ax_m$;
15: **if** satisfied **then**
16: stop
17: **else**
18: $x_0 = x_m$, $v_1 = r_m/\|r_m\|$ and go to line 3

While the original GMRES is guaranteed to converge in at most n (the matrix dimension) steps, the restarted GMRES [38] looses this optimality property and it can stagnate when the matrix is not positive definite. The restarted algorithm destroys the Krylov subspace and starts all over. It is possible that the Krylov subspace may not be large enough to converge fast, and the solution may not be found. In order to overcome this and to reduce the number of iteration steps, an effective preconditioner can be used.

3.2 Preconditioning techniques

Lack of robustness is a widely recognized weakness of iterative methods with respect to direct methods. Iterative methods may suffer from slow convergence on problems arising from practical applications.

The use of preconditioning techniques is meant to improve the perfor-

mance and reliability of Krylov subspace methods. It is widely recognized that preconditioning plays a very vital role in developing efficient solvers for difficult matrices in scientific computing.

The term preconditioning refers to transforming the original linear system into another system in which the solution process has better properties to converge. When the coefficient matrix A is highly nonsymmetric and/or indefinite, iterative methods need the assistance of preconditioning to transform system Eq. (3.1) into an equivalent system, more amenable to an iterative solver. The transformed *preconditioned system* can be written in the form

$$M^{-1}Ax = M^{-1}b \tag{3.8}$$

when preconditioning is applied from the left, and

$$AM^{-1}y = b, x = M^{-1}y \tag{3.9}$$

when preconditioning is applied from the right.

The matrix M is a nonsingular approximation to A, and is called the *preconditioner matrix*. There are different types of preconditioners, like diagonal preconditioner (Jacobi preconditioner)[5], Symmetric Successive Over-Relaxation (SSOR) preconditioner [4], the Sparse Approximate Inverse preconditioner (SPAI) [57, 26] and Incomplete LU factorization preconditioner (ILU). We will only briefly introduce ILU [28] in this thesis since that is the one related to our research.

3.2.1 Incomplete LU factorization preconditioner

Triangular factors $\bar{L}$ and $\bar{U}$ can be obtained if we factorize the coefficient matrix A via Gaussian Elimination (GE). By discarding part of the fill-in during the factorization process we can get simple and powerful preconditioners $M = \bar{L}\bar{U}$, where $\bar{L}$ and $\bar{U}$ are incomplete (approximate) LU factors.

Various strategies for selecting the sparsity patterns of $\bar{L}$ and $\bar{U}$ lead to different methods, see e.g. [70]. A stable ILU factorization is proved to exist for arbitrary choices of the sparsity pattern of $\bar{L}$ and $\bar{U}$ only for particular classes of matrices, such as M-matrices [60] and H-matrices with positive diagonal entries [86].

Let $\mathbf{n} = \{1, 2, \ldots, n\}$, we fix a subset $\mathcal{S} \subseteq \mathbf{n} \times \mathbf{n}$ which contains a set of positions in the matrix, which usually includes the diagonal line and all

nonzero-element positions. Fill-in in the LU factors is allowed only in the positions achieved by $\mathcal{S}$. Formally, we can describe the key step of ILU decomposition as follows

$$a_{ij} = \begin{cases} a_{ij} - a_{ik}a_{kk}^{-1}a_{kj}, & \text{if}(i,j) \in \mathcal{S} \\ a_{ij}, & \text{otherwise} \end{cases} \tag{3.10}$$

for each k and for $i, j > k$.

If $\mathcal{S}$ represents the set of nonzero-element positions of coefficient matrix A, we obtain ILU(0) which does not allow any fill-in. It is easy to implement and cheap to compute. It also shows good performance on easy problems such as diagonally dominant matrices. However, for more realistic and difficult problems, a more accurate and sophisticated preconditioner which allows more fill-in in the construction is required to improve the accuracy.

A hierarchy of ILU preconditioners can be obtained based on the "levels of fill-in" concept. The definition of initial level of fill-in of a matrix entry a_{ij} is as follow:

$$lev_{ij} = \begin{cases} 0, & \text{if} \quad a_{ij} \neq 0 \quad \text{or} \quad i = j \\ \infty, & \text{otherwise} \end{cases} \tag{3.11}$$

This entry is modified at each step during the ILU process. Its value is updated according to

$$lev_{ij} = \min\{lev_{ij}, lev_{ik} + lev_{kj} + 1\}. \tag{3.12}$$

where the positive integer p denotes the level of fill, the p level ILU preconditioner is referred to as ILU(p). All fill-ins whose level is greater than p are dropped. The case $p = 0$, corresponds to ILU(0) introduced before. As the level increases, the cost and accuracy grow. Normally, ILU(1) is a good option for most problems. It is a considerable improvement over ILU(0), and the computational and memory cost is still acceptable for many practical problems.

However, ILU(p) is blind to the numerical values of the entries because the dropping is only determined by the structure of A. Because of this, ILU(p) may not be that effective on certain type of problems, especially for matrices which are far away from being diagonally dominant. Many small absolute value fill-ins are stored during the ILU(p) process. They only contribute little to the preconditioner performance but increase the storage. To avoid this situation, some other methods where new fill-in entries are accepted or

dropped based on their magnitude rather than their locations are proposed. With these techniques, the zero pattern $\mathcal{S}$ is determined dynamically.

Saad [70] has proposed the dual threshold based ILU preconditioner. The basic idea is to fix a dropping tolerance τ and a number p which is the maximal amount of fill-in in each row of the incomplete LU factors; at each step of the elimination process, only p nonzero entries are computed in each row with magnitude smaller than τ. This dual threshold ILU is denoted as ILUT(τ, p). It allows more flexibility by tuning both parameters τ and p. Lower values of τ and higher values of p increase the accuracy of the computation. We mostly use ILUT in the following chapters.

Many techniques can help improve the quality of the preconditioner on more general problems, such as reordering, scaling, diagonal shifting, pivoting and condition estimators [37, 74, 62, 15]. As a result of this active development, in the past decade several successful computational experiences have been reported using ILU preconditioners in areas that were the exclusive domain of direct solution methods like, e.g., in circuits simulation, power system networks, chemical engineering plants modeling, graphs and other problems not governed by PDEs, or in areas where direct methods have been traditionally preferred, such as structural analysis, and semiconductor device modeling (see e.g. [71, 14, 12, 58, 73]).

Classic ILU preconditioners are designed to be a class of methods for solving general sparse linear systems of equations. Algebraic MultiGrid (AMG) [6, 7, 45] methods are a type of multilevel solvers for linear systems introduced initially in 1970s by Ruge and Stuben [65]. Its performance highly depends on the underling PDE problem. For some classes of PDEs, AMG methods exhibit linear stability, meaning that the number of iterations is linearly grid independent. ILU preconditioners are more general than AMG and can work on many problems where AMG fails, but their weakness is the poor scalability; the convergence rate normally deteriorates as the matrix size grows. Nowadays, the computational problems tend to be larger and larger, so it is attractive to develop methods which combine the generality of the ILU method and the scalability of AMG.

Previous research already proposed some ILU based multilevel methods [8]. We will recall them in the next section.

3.3 Multilevel Incomplete LU (ILU) decomposition solvers

3.3.1 ILUM: A Multi-elimination ILU preconditioner

Graph theory is an ideal tool for representing the structure of sparse matrices and for this reason it plays a major role in sparse matrix computation. Recall that a graph can be defined by two sets: one is the set of vertices

$$V = \{v_1, v_2 \ldots, v_n\}, \tag{3.13}$$

and a set of edges E which consists of pairs (v_i, v_j), where v_i, v_j are elements of V, so we have:

$$E \subseteq V \times V. \tag{3.14}$$

This graph $G = (V, E)$ is a way of representing a binary relation of a set V. In the sparse matrix context, the adjacency graph of sparse matrix A is a graph $G = (V, E)$, whose n vertices in V represent the n unknowns and whose edges represent the binary relations established by the linear system. There is an edge from vertex j to vertex i when $a_{ij} \neq 0$. Here, the graph is directed, unless the matrix A is structurally symmetric ($a_{ij} = 0$ iff $a_{ji} = 0$ for all $1 \leq i, j \leq n$).

A multi-elimination ILU preconditioner (ILUM) is an incomplete factorization technique based on independent set orderings [70, 68]. The idea is to find the independent set, and then eliminate the unknowns associated with it, then to obtain a smaller reduced linear system and solve it recursively.

During the Gaussian elimination process, parallelism can be exploited if the unknowns x_i and x_j are independent from each other ($a_{ij} = 0$ and $a_{ji} = 0$ in the matrix A) during the factorization. So there are two extreme cases; one is all the unknowns are all independent, i.e. the matrix is diagonal, the other one is that the matrix is fully dense. Sparse matrices derived from applications are somewhere in between these two extremes. To design this multilevel Gaussian elimination process, we only need to find a permutation matrix P to permute the input matrix A to a 2×2 block matrix

$$PAP^T = \begin{pmatrix} D & F \\ E & C \end{pmatrix}, \tag{3.15}$$

where D is diagonal and C is arbitrary. Different reordering schemes can be used for this purpose, like independent set orderings, multicolor orderings; see also [68]. Here, we only introduce the simple greedy algorithm.

Definition 3.3.1. *Let $G = (V, E)$ denote the adjacency graph of the matrix A, and let (x, y) denote an edge from node x to node y. We define an independent set S as subset of the vertex set V, such that*

$$\text{if } x \in S \quad \text{and} \quad \{\, (x,y) \in E \;\; or \;\; (y,x) \in E \,\} \rightarrow y \notin S. \tag{3.16}$$

Simply speaking, the elements in S are not allowed to have couplings between each other, either in forward or backward direction.

Throughout the thesis, we use the term *independent set* to always refer to the *maximal independent set*.

The procedure of a greedy algorithm consists of traversing all the nodes in S, starting from an empty set S, and visiting the nodes in increasing index order. For every node, it marks the node itself and its neighbors, finds the unmarked nodes which are not coupled with S, then adds them into S.

Algorithm 3.3 Greedy algorithm for independent set ordering.

```
1: Let S = ∅
2: for j = 1, 2, ..., n do
3:    if node j is not marked then
4:       S = S ∪ {j}.
5:       Mark j and all its nearest neighbors.
```

The idea of greedy algorithm was also used in the paper [32] to create rank-k updates by having independent pivots.

The greedy algorithm compute the reordering matrix P, and the reordering matrix P permute the input matrix into the form in Eq. (3.15) 2×2 block structure. At this stage, the actual multi-elimination factorization can be computed.

The following block LU decomposition is performed to eliminate the unknowns of the independent set by performing this block LU decomposition

$$\begin{pmatrix} D & F \\ E & C \end{pmatrix} = \begin{pmatrix} I & 0 \\ ED^{-1} & I \end{pmatrix} \times \begin{pmatrix} D & F \\ 0 & A_1 \end{pmatrix} \tag{3.17}$$

A_1 is the Schur complement which is calculated via

$$A_1 = C - ED^{-1}F \tag{3.18}$$

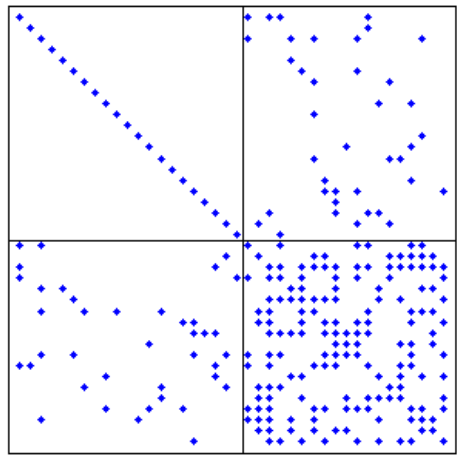


Figure 3.1: Pattern of the one level factorization of ILUM.

where A_1 is the Schur complement with respect to C. The reduction process can be applied recursively to each reduced system, until the last Schur complement is small enough to be solved with a standard method.

Fig 3.1 is an illustration for the permuted 2×2 block matrix except that we replace the lower-right matrix C with the Schur complement A_1 Eq. (3.18). We can see that the Schur complement tends to be denser and denser as the level grows, so the reduced system tends to be more and more expensive to solve in terms of both computational and memory cost. Therefore, a dropping strategy may be adopted during the process.

The solving phase consists of a backward and a forward solution. The last level linear system, does not need to be solved accurately. For instance, a Krylov-subspace method can be used to solve it within a given tolerance.

We describe the solving phase starting from the first level since it is a recursive process. A forward solution process is applied to the right hand side vector b. First use the global permutation array *i.e.*, the product $P_{nlev-1}P_{nlev-2}\dots P_0$ partitions b into

$$b_0 = \begin{pmatrix} f_0 \\ g_0 \end{pmatrix}$$

according to Eq. (3.15). The forward step maps the right hand vector b to the next level. It substitutes the second part of b_0 with

$$g_0 = g_0 - E_0 D_0^{-1} y_0.$$

This holds for the first level. We can continue to apply this to g_0 which is also our b_1 based on the second level partitioning. At step ℓ, we obtain

$$b_\ell = \begin{pmatrix} f_\ell \\ g_\ell \end{pmatrix}$$

We repeat this process until $\ell = nlev - 1$. Then we can solve the last level linear system and obtain the last level solution. The backward step proceeds in a similar way. At the end, we apply the inverse global permutation to x_0 to obtain the solution for the original linear system,

$$x_\ell = \begin{pmatrix} y_\ell \\ z_\ell \end{pmatrix}$$

see also Algorithm 3.4 [68]

Algorithm 3.4 `ILUM_Solve`(A_0, b_0). Forward and backward solutions.

1: Permute right hand vector b via global permutation array
2: **for** $\ell = 0, 1, \ldots, nlev - 1$ **do**
3: $\quad g_\ell = g_\ell - E_\ell D_\ell^{-1} y_\ell$.
4: $\quad b_{\ell+1} = g_\ell$.
5: Solve the last linear system with a relative tolerance ε,
$$A_{nlev} x_{nlev} = b_{nlev},$$
$$z_{nlev-1} = x_{nlev}.$$
6: **for** $\ell = nlev - 1, \ldots, 1, 0$ **do**
7: $\quad y_\ell = D_\ell^{-1}(y_\ell - F_\ell z_\ell)$.
8: $\quad z_{\ell-1} = x_\ell = \begin{pmatrix} y_\ell \\ z_\ell \end{pmatrix}$.
9: Permute the resulting solution vector back to the original ordering to get the solution x_0.

3.3.2 BILUM: A block versions of the Multi-elimination and Multilevel ILU Preconditioner

ILUM is an effective multilevel solver, but is also has some drawbacks. Our problem is that the diagonal entries in D might be small at some steps of the reduction process leading to an unstable factorization. Second, if the matrix A is not sufficiently sparse, the size of D is small, which makes the Schur complement linear system big. This may lead to high computational complexity and poor convergence.

In order to avoid these problems, block versions of ILUM in [75] (BILUM) were developed based on the concept of block independent sets. Recall the definition of independent set in Section 3.3.1. In order to introduce BILUM, we first generalize independent sets to block independent sets.

Greedy algorithm for block independent sets

Consider a group of non-empty subsets of vertex set V which are disjoint, such that

$$Y_j \cap Y_i = \emptyset, \quad \text{if} \quad j \neq i.$$

The quotient graph can be obtained by considering each subset as a supervertex Y_i. There is an edge between supervertex Y_i and Y_j if there exists an edge between one vertex in Y_i and one vertex in Y_j. Formally, we can describe by symbols as

$$Y_i \to Y_j, \quad \text{if} \quad \exists k_i \in Y_i, \quad \exists k_j \in Y_j \quad \text{s.t} \quad a_{k_i,k_j} \neq 0.$$

A block independent set is an independent set [51] on this quotient graph. According to the following definition

Definition 3.3.2. *Let $Y_1, Y_2, \ldots, Y_m$ be a collection of disjoint nonempty subsets of V. The set $S = \{Y_1, Y_2, \ldots, Y_m\}$ is called a block independent set if any two subsets Y_i and Y_j in S are not adjacent in the quotient graph.*

The family of sets $Y_1, Y_2, \ldots, Y_m$ can have variable size, but in this section, we only focus on a block independent set ordering which produces constant block size.

To simplify the description of the block independent set ordering algorithm, we assume a constant block size equals to 2; it can be easily generalized to arbitrary positive integer number k.

If we couple a node with one of its neighbors, a block of size 2 will be found. There are different ways of finding this coupling. One is to check the absolute value of the nodes' neighbors, and pick the one with the largest absolute value; we call this approach as the strongest link. By doing so, we can keep the 2×2 diagonal blocks away from singularity. This will lead to a more stable inversion of the diagonal blocks. In the following algorithms, adj(j) denotes the set of j node neighbors, i.e., all nodes i such that $a_{j,i} \neq 0$.

Algorithm 3.5 Greedy algorithm for independent set ordering by strongest links.

1: Let $m = 0$
2: **for** $j = 1, 2, \ldots, n$ **do**
3: **if** node j is not marked **then**
4: $m = m + 1, B_m = \{j\}$
5: Choose $s \in \text{adj}\{j\}$ such that $|a_{j,i}| = \max\{|a_{j,i}|, i \in \text{adj}\{j\}\}$
6: $B_m = B_m \cup \{s\}$
7: Mark j and all nodes in adj$\{j\}$.

The block-independent set from this algorithm will be $S_2 = \{B_1, B_2, \ldots, B_m\}$. There are some other ways to pick vertex s to couple with j to form a set B_m at line 5 in Algorithm 3.5.

Note that for a multilevel method, at each level it is very advantageous for computational efficiency that the Schur complement size is small, which means the size of the whole independent set should be big. This suggests that we may try to couple the current node with the smallest degree node to form a block of size 2.

Algorithm 3.6 Greedy algorithm for independent set ordering by minimal degree.

1: Let $m = 0$
2: **for** $j = 1, 2, \ldots, n$ **do**
3: **if** node j is not marked **then**
4: $m = m + 1, B_m = \{j\}$
5: Choose $s \in \text{adj}\{j\}$ such that $\deg(s) = \min\{\deg(i), i \in \text{adj}\{j\}\}$
6: $B_m = B_m \cup \{s\}$
7: Mark j and all nodes in adj$\{j\}$.

Block ILUM factorization

Since BILUM is just a variant of ILUM, the main frame is similar with ILUM. Here we recall the main computational steps.

After the greedy algorithm, we will permute A into the form

$$\begin{pmatrix} D & F \\ E & C \end{pmatrix}, \tag{3.19}$$

where D is a block diagonal matrix

$$D = diag(D_1, D_2, \ldots, D_l), \tag{3.20}$$

and D_i is a $k \times k$ matrix. The following procedure is the same as in ILUM, we may eliminate the unknowns of the independent set to obtain a reduced system, analogous to Eq. (3.17) and Eq. (3.18).

But the inverse of matrix D will be calculated differently. Since D is block diagonal instead of diagonal, the inversion can be done by inverting each small block. These small matrices can be factorized using Gaussian Elimination (GE) and the factors can be stored instead of explicit inverses. Another way of inverting small blocks is to compute the pseudo inverse by a Truncated Singular Value Decomposition (SVD). These two strategies are also used to invert diagonal blocks during the block ILU factorization process; see [69].

The forward-backward solution process is similar to ILUM except that the diagonal matrix D is replaced by the block diagonal matrix.

3.3.3 ARMS: The Algebraic Recursive Multilevel Solver

The ARMS solver proposed in [74] is a generalization of BILUM. The multilevel factorization processes are similar, but there are many different implementation aspects.

Recall the greedy algorithms described in the previous sections. In ILUM, it finds an independent set. In BILUM, it generates a block independent set with constant block size k. The greedy algorithm variant used in ARMS is more sophisticated. A criterion for detecting diagonally dominant rows is added and a level set approach is used. Details of the procedure are shown in Algorithm 3.7.

We recall the notations used in Algorithm 3.7: $bsize$ is the upper bound of block size of each independent set, $adj(k)$ is the set of direct neighbors

Algorithm 3.7 Independent set ordering with weights.

```
1: for j = 1, 2, ..., n do
2:   if node j is unmarked then
3:     jcount = 0
4:     if w(j) < tol then
5:       Add_to_F(j)
6:     else
7:       Add_to_C(j); Level_Set = {j}
8:       while jcount < bsize and Level_Set is not empty do
9:         for k ∈ Level_Set do
10:          if w(k) < tol then
11:            Add_to_F(k)
12:          else
13:            Add_to_C(k); jcount++;
14:        for k ∈ Level_Set do
15:          for i ∈ adj(k) do
16:            if i is unmarked then
17:              Add_to_F(i)
```

of node k, and *Level_Set* is updated by each *Add_to_C* operation. The mechanism to compute *Level_Set* is to start from one node to visit all his neighbors, that gives the second *Level_Set*, and then visit all these neighbors' neighbors, giving the third *Level_Set*; w is the weight array, in which each entry is defined as:

$$w(i) = \frac{a_{ii}}{\sum_{j=1}^{n} a_{ij}} \tag{3.21}$$

Obviously, the weight defined by Eq. (3.21) represents the relative diagonal dominance of each row. *tol* is the threshold belongs to $[0, 1]$, defined by user for the row weight. Eq. (3.21) calculates the degree of diagonal dominance of a row and compare it with *tol*. The idea behind this strategy is to move the diagonally dominant rows into the independent set, so it enhances the stability of inverting diagonal block matrix D in Eq. (3.19). *Add_to_C* and *Add_to_F* respectively add the current node into C-block unknowns corresponding to the independent blocks and add the current node into F-block unknowns corresponding to Schur complement nodes. After that, *Add_to_C*

and *Add_to_F* also mark the node.

The for loop line 14 of Algorithm 3.7, is executed in the case where the independent set block reaches *bsize* but the *Level_Set* is not empty. It moves the last *Level_Set*'s neighbors into unknowns corresponding to the block F, to make sure that the unknown candidates of the C block will not be coupled with block F.

The main frame of ARMS is very similar to BILUM. The 2×2 block ILU process can be applied recursively to each consecutively reduced system until the Schur complement is small enough to be solved with a standard method such as a dense LAPACK solver or an ILU solver. The solution process with the above factorization consists of a level-by-level forward elimination, followed by an exact solution on the last reduced system, plus a suitable inverse permutation.

Another major implementation difference is that full recursivity is implemented by ARMS, and the calculation of the Schur complement $A_1 = C - ED^{-1}F$ is optimized.

Since implementation details are very important for numerical algorithms, we would like to introduce the details of the Schur complement calculation in ARMS. First the incomplete triangular factors $\bar{L}$, $\bar{U}$ of D are computed by one sweep of ILUT, and an approximation $\bar{W}$ to $\bar{L}^{-1}F$ is also computed. In a second loop, an approximation $\bar{G}$ to $E\bar{U}^{-1}$ and an approximate Schur complement matrix $\bar{A}_1$ are derived. This holds at each reduction level. At the last level, another sweep of ILUT is applied to the (last) reduced system. The blocks $\bar{W}$ and $\bar{G}$ are stored temporarily, and then discarded from the data structure after the Schur complement matrix is computed. Only the incomplete factors of D at each level, those of the last level Schur matrix, and the permutation arrays are needed for the solving phase. By this implementation, dropping can be performed separately in the matrices $\bar{L}$, $\bar{U}$, $\bar{W}$, $\bar{G}$, $\bar{A}_1$. This in turns allows to factor D accurately without incurring additional costs in $\bar{G}$ and $\bar{W}$, achieving high computational and memory efficiency. Implementation details and careful selection of the parameters are always critical aspects to consider in the design of sparse matrix algorithms. See more details in [74].

Among the three multilevel ILU decomposition methods, ARMS is the most recent and mature one. It also presents a very referable framework for our new VBARMS solver.

4 The VBARMS solver

In this chapter, we will introduce the computational steps of our new solver VBARMS [22] and present the numerical results. But before that, we would like to introduce the block-wise operations which appear during solving linear systems, since they are the building blocks for VBARMS.

4.1 Building blocks for VBARMS

In general, a block-wise matrix can be represented as follows:

$$A = \begin{array}{c} \\ m_1 \\ m_2 \\ \\ m_p \end{array} \begin{array}{c} \begin{array}{cccc} n_1 & n_2 & & n_q \end{array} \\ \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1q} \\ A_{21} & A_{22} & \dots & A_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & \dots & A_{pq} \end{pmatrix} \end{array} \tag{4.1}$$

Here, we have $m_1 + m_2 \cdots + m_p = m$ and $n_1 + n_2 \cdots + n_q = n$, A_{ij} denotes the (i, j)th block of A. So we say that above matrix has block-wise dimension $p \times q$.

Next we will introduce the basic operations for block matrices.

Matrix-matrix addition

Let us assume we have another matrix B which has has the same partition as A.

$$B = \begin{array}{c} \\ m_1 \\ m_2 \\ \\ m_p \end{array} \begin{array}{c} \begin{array}{cccc} n_1 & n_2 & & n_q \end{array} \\ \begin{pmatrix} B_{11} & B_{12} & \dots & B_{1q} \\ B_{21} & B_{22} & \dots & B_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ B_{p1} & B_{p2} & \dots & B_{pq} \end{pmatrix} \end{array}$$

Then the sum $C = A + B$ is $p \times q$ block matrix defined by

$$C = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1q} \\ C_{21} & C_{22} & \cdots & C_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ C_{p1} & C_{p2} & \cdots & C_{pq} \end{pmatrix} = \begin{pmatrix} A_{11}+B_{11} & A_{12}+B_{12} & \cdots & A_{1q}+B_{1q} \\ A_{21}+B_{21} & A_{22}+B_{22} & \cdots & A_{2q}+B_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1}+B_{p1} & A_{p2}+B_{p2} & \cdots & A_{pq}+B_{pq} \end{pmatrix}$$

The addition of each couple of elements $C_{ij} = A_{ij} + B_{ij}$ can be implemented by directly calling the BLAS level 1 routine (the blocks are assumed to be dense) like DAXPY to execute, which is more efficient.

Matrix-matrix multiplication

Let us assume that:

$$A \in R^{m \times n}, B \in R^{n \times k}$$

$$A = \begin{array}{c} \\ m_1 \\ m_2 \\ \\ m_p \end{array} \begin{array}{c} \begin{array}{cccc} n_1 & n_2 & & n_q \end{array} \\ \begin{pmatrix} A_{11} & A_{12} & \ldots & A_{1q} \\ A_{21} & A_{22} & \ldots & A_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & \ldots & A_{pq} \end{pmatrix} \end{array}, \quad B = \begin{array}{c} \\ n_1 \\ n_2 \\ \\ n_q \end{array} \begin{array}{c} \begin{array}{cccc} k_1 & k_2 & & k_r \end{array} \\ \begin{pmatrix} B_{11} & B_{12} & \ldots & B_{1r} \\ B_{21} & B_{22} & \ldots & B_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ B_{q1} & B_{q2} & \ldots & B_{qr} \end{pmatrix} \end{array}$$

We have $k_1 + k_2 + \cdots + k_r = k$ here, since B's block structure is compatible with A's. The matrix product can be formed block-wise, yielding C as an $(m \times k)$ matrix with $(p \times r)$ blocks.

$$C = A \times B = \begin{array}{c} \\ m_1 \\ m_2 \\ \\ m_p \end{array} \begin{array}{c} \begin{array}{cccc} k_1 & k_2 & & k_r \end{array} \\ \begin{pmatrix} C_{11} & C_{12} & \ldots & C_{1r} \\ C_{21} & C_{22} & \ldots & C_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ C_{p1} & C_{p2} & \ldots & C_{pr} \end{pmatrix} \end{array}, \tag{4.2}$$

The blocks in matrix C are calculated by

$$C_{ij} = \sum_{k=1}^{q} A_{ik} B_{kj}, \qquad i = 1:p, \;\; j = 1:r.$$

For each product, in the implementation, we can call BLAS 3 routine DGEMM to compute $A_{ik}B_{kj}$.

Matrix-vector product

One of the most important steps of solving linear systems is the matrix-vector product Ax. It is also a special case of Eq. (4.2)

$$b = Ax = \begin{array}{c} \\ m_1 \\ m_2 \\ \\ m_p \end{array} \begin{array}{c} \begin{array}{cccc} n_1 & n_2 & & n_q \end{array} \\ \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1q} \\ A_{21} & A_{22} & \dots & A_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & \dots & A_{pq} \end{pmatrix} \end{array} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_q \end{pmatrix} = \begin{pmatrix} \sum_{k=1}^{q} A_{1k}x_k \\ \sum_{k=1}^{q} A_{2k}x_k \\ \vdots \\ \sum_{k=1}^{q} A_{pk}x_k \end{pmatrix}$$

According to the partition of A, x also got split into small subvectors, in this context x_i denotes the i-th subvector of the vector x according to the above partitioning. For each $A_{ik}x_k$, in the implementation, we can call BLAS 2 routine DGEMV to perform this operation,

Block-wise ILU preconditioner

After the introduction of basic operations, we also recall the block-wise ILU preconditioner which will be an important component of our VBARMS method. First is block-wise ILU factorization, $A = LU$, see Eq. (4.3) and Algorithm 4.1

$$
\begin{aligned}
A &= \begin{array}{c} \\ m_1 \\ m_2 \\ \\ m_p \end{array}
\begin{array}{c}
\begin{array}{cccc} m_1 & m_2 & & m_p \end{array} \\
\begin{pmatrix} A_{11} & A_{12} & \dots & A_{1p} \\ A_{21} & A_{22} & \dots & A_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & \dots & A_{pp} \end{pmatrix}
\end{array} \\
&= \begin{array}{c} \\ m_1 \\ m_2 \\ \\ m_p \end{array}
\begin{array}{c}
\begin{array}{cccc} m_1 & m_2 & & m_p \end{array} \\
\begin{pmatrix} L_{11} & 0 & \dots & 0 \\ L_{21} & L_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ L_{p1} & L_{p2} & \dots & L_{pp} \end{pmatrix}
\end{array}
\times
\begin{array}{c} \\ m_1 \\ m_2 \\ \\ m_p \end{array}
\begin{array}{c}
\begin{array}{cccc} m_1 & m_2 & & m_p \end{array} \\
\begin{pmatrix} U_{11} & U_{12} & \dots & U_{1p} \\ 0 & U_{22} & \dots & U_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & U_{pp} \end{pmatrix}
\end{array} \\
&= LU
\end{aligned}
\tag{4.3}
$$

Algorithm 4.1 `General Static Pattern block ILU`

Input: The block-wise square matrix A.
Output: Updated A which contains L, U factors.

1: **for** $k = 1, 2, \dots, p-1$ **do**
2: **for** $i = k+1, \dots, p$ **do**
3: $A_{ik} = A_{ik} * A_{kk}^{-1}$,
4: **for** $j = k+1, \dots, p$ **do**
5: $A_{ij} = A_{ij} - A_{ik} * A_{kj}$.

In Algorithm 4.1, the diagonal blocks in A are square and normally assumed nonsingular. In step 3, the inverse of A_{kk} is calculated via GE or Singular Value Decomposition. Step 5 is performed via block operations. The updated matrix A has lower triangular part L and upper triangular part U. It is obvious that L and U also preserve the block-structure.

Once the L, U factors are computed, we can easily generalize the two steps of LU solving phase to a block LU. The LU method contains the following two triangular subproblems:

1. Solving y from $Ly = b$ by block operations, see Algorithm 4.2

$$Ly = \begin{array}{c} \\ m_1 \\ m_2 \\ \\ m_p \end{array} \begin{array}{c} \begin{array}{cccc} m_1 & m_2 & & m_p \end{array} \\ \begin{pmatrix} L_{11} & 0 & \dots & 0 \\ L_{21} & L_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ L_{p1} & L_{p2} & \dots & L_{pp} \end{pmatrix} \end{array} \times \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix} = b$$

2. Solving x from $Ux = y$ by block operations, see Algorithm 4.3

$$Ux = \begin{array}{c} \\ m_1 \\ m_2 \\ \\ m_p \end{array} \begin{array}{c} \begin{array}{cccc} m_1 & m_2 & & m_p \end{array} \\ \begin{pmatrix} U_{11} & U_{12} & \dots & U_{1p} \\ 0 & U_{22} & \dots & U_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & U_{pp} \end{pmatrix} \end{array} \times \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{pmatrix} = y$$

Algorithm 4.2 `Forward substitution`

Input: Right hand vector b and a lower triangular block-wise matrix $L_{p\times p}$.
Output: Intermediate solution y such that $Ly = b$ holds.

1: $y_1 = b_1 * L_{11}^{-1}$
2: **for** $i = 2, 3 \dots, p$ **do**
3: $\quad sum = \{0, 0, ..., 0\}^T$,
4: $\quad$ **for** $k = 1, \dots, i-1$ **do**
5: $\quad\quad sum = sum + L_{ik} y_k$,
6: $\quad y_i = (b_i - sum) * L_{ii}^{-1}$,

In the Algorithm 4.2, there is no need to compute L_{ii}^{-1} since L_{ii} is the identity matrix. In the Algorithm 4.3, the inverse of U_{ii} is calculated by GE and SVD. All the block operations involved have been introduced already.

4.2 The variable block ARMS factorization

4.2.1 The mathematical process of VBARMS

With the previous two chapters setup, it is easy to design a variable block ARMS factorization process that exploits this block structure and achieves high performance.

Algorithm 4.3 `Backward substitution`

Input: Intermediate solution y and a lower triangular block-wise matrix $U_{p\times p}$.

Output: Final solution x from $Ux = y$.

1: $x_p = y_p * U_{pp}^{-1}$
2: **for** $i = p-1, \ldots, 1$ **do**
3: $\quad sum = \{0, 0, ..., 0\}^T$,
4: $\quad$ **for** $k = i+1, \ldots, p$ **do**
5: $\quad\quad sum = sum + U_{ik}x_k$,
6: $\quad x_i = (y_i - sum) * U_{ii}^{-1}$,

Before introducing the method, there are a few advantages described below which we would like to emphasize when a block-wise multilevel solver is used.

1. *Memory.* The usual Block Sparse Row format in SPARSKIT [77] uses Block wise Compressed Sparse Row format. In ITSOL, there is also a Variable Block Compressed Sparse Row (VBCSR) format.

   A clear advantage is to store the matrix as a collection of blocks using VBCSR format instead of the traditional compressed sparse row (CSR) format. We used the ones in ITSOL [52]. VBCSR saves column indexes and pointers for the block entries. Their C code is as follows

Listing 4.1: CSR

```
typedef struct SpaFmt {
/*-------------------------------------------------------
| C-style CSR format
|-------------------------------------------------------*/
  int n;          /* the dimension of the matrix */
  int *nzcount;   /* length of each row */
  int **ja;       /* pointer-to-pointer to store */
                  /* column indices */
  double **ma;    /* pointer-to-pointer to store */
                  /* nonzero entries */
} SparMat, *csptr;
```

In this format, the *nzcount* and *ja* arrays store the sparsity structure of the blocks. *ma* is a two dimensional pointer, each entry stores the value of the element.

Listing 4.2: VBCSR

```
typedef double *BData;

typedef struct VBSpaFmt {
/*-------------------------------------------------------
| C-style VBCSR format -
|-------------------------------------------------------*/
    int n;          /* the block dimension of */
                    /* the matrix */
    int *bsz;       /* the row/col of the first */
                    /* element of each diagonal */
                    /* block */
    int *nzcount;   /* length of each row */
    int **ja;       /* pointer-to-pointer to store */
                    /* column indices */
    BData **ba;     /* pointer-to-pointer to store */
                    /* nonzero blocks*/
} VBSparMat, *vbsptr;
```

Similar to CSR format, the *nzcount* and *ja* arrays store the sparsity structure of the entries. but *ba* is a two dimensional pointer where

each entry points to one dense block. The entries of array *bsz* point to the beginning of each diagonal block. To perform any operations, as we introduced in Section 4.1, we pass each block as a small dense matrix to a BLAS/LAPACK subprogram.

The best way to understand the VBCSR format is to view a point-wise matrix as a block-wise matrix whose nonzero entries are dense blocks (square or rectangular). Each block is treated as a dense block. If there are entries in some blocks they must be taken as nonzero entries with value zero.

VBCSR format can save a lot of column and row indexes storage. For example, if matrix A is an $n \times n$ matrix with block size $l \times l$, the block-wise matrix dimension will be $(n/l) \times (n/l)$ (to simplify the description here, we assume the matrix has constant block size) so we store much less column indexes. In this matrix, each entry is a block; in the VBCSR format, it is an array of size $l \times l$.

2. *Stability.*

   On indefinite problems, during the ILU factorization, small pivots often lead to unstable and therefore inaccurate factorization. Using blocks instead of single elements enables a better control of pivot breakdowns, near singularities, and other possible sources of numerical instabilities. Because the unstable factors like zeros and small values on the diagonal line will be moved into blocks. The diagonal blocks have less chance to be singular or near-singular, and also approximate inverse techniques can be used to invert diagonal small blocks. Block ILU solvers may be used instead of pointwise ILU methods.

3. *Complexity.*

   Since we generate the independent set ordering based on the quotient graph $\mathcal{G}/\mathcal{B} = \{V_{\mathcal{B}}, E_{\mathcal{B}}\}$, then instead of permuting the point-wise entries of the matrix, we switch the block rows and columns. Therefore, normally the obtained block diagonal matrix D in Eq. (4.6) is bigger than the one in ARMS and consequently, the Schur complement is smaller. This is very advantageous for a multi-level solver.

   Besides that, a better conditioned last reduced system is obtained and it will be easily solved.

4. *Efficiency.*

   Every operation will be done on blocks after Eq. (4.4). Higher level optimized BLAS routines will be used as computational kernels. This design will lead to better flops to memory ratios on modern cache-based computer architectures.

   The Table 4.1 illustrates the advantage of the level 3 BLAS. The ratio of floating-point operations to the amount of data movement of BLAS 1, 2, 3 routines is displayed respectively [34].

Table 4.1: Advantage of the Level 3 BLAS.

BLAS routines	Loads and Stores	Floating-Point Operation	Ratio $n = m = k$
Level 1 DAXPY $y = y + \alpha x$	$3n$	$2n$	3:2
Level 2 DGEMV $y = \beta y + \alpha Ax$	$mn + n + 2m$	$2mn$	1:2
Level 3 DGEMM $C = \beta C + \alpha AB$	$2mn + mk + kn$	$2mnk$	2:n

   In Table 4.1, the dimension of y, x is n, A is a $m \times n$ matrix, B is a $n \times k$ matrix and C is a $m \times k$ matrix. The Loads and Stores column gives the amount of data movement, and the Ratio colum shows that BLAS level 2 routine DGEMV is able to perform three times more operations than DAXPY with the same amount of data movement. And BLAS level 3 routine DGEMM is even more operation-intensive than DAXPY.

5. *Cache effects.* Cache is a form of storage that is automatically filled and emptied according to a fixed scheme defined by the hardware system. Caches work on the assumption that data that is accessed once will usually be accessed soon again. This kind of behavior is known as data locality. High performance can be achieved by using data locality.

   Better cache reuse is possible for block algorithms since the computa-

tional unit is stored in an array.

It is simpler to describe VBARMS from a graph point of view. Suppose to permute A in block form as

$$\widetilde{A} = P_B A P_B^T = \begin{bmatrix} \widetilde{A}_{11} & \widetilde{A}_{12} & \cdots & \widetilde{A}_{1p} \\ \widetilde{A}_{21} & \widetilde{A}_{22} & \cdots & \widetilde{A}_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ \widetilde{A}_{p1} & \widetilde{A}_{p2} & \cdots & \widetilde{A}_{pp} \end{bmatrix}, \tag{4.4}$$

where the diagonal blocks $\widetilde{A}_{ii}$, $i = 1, \ldots, p$ are $n_i \times n_i$ and the off-diagonal blocks $\widetilde{A}_{ij}$ are $n_i \times n_j$. We may represent the adjacency graph of $\widetilde{A}$ by the quotient graph of $A + A^T$ [40]. Calling $\mathcal{B}$ the partition into blocks Eq. (4.4), we denote by $\mathcal{G}/\mathcal{B} = \{V_\mathcal{B}, E_\mathcal{B}\}$ the quotient graph obtained by coalescing all the vertices assigned to the block $\widetilde{A}_{ii}$ ($i = 1, \ldots, p$), into a supervertex Y_i. An edge connects any supervertex Y_i to another supervertex Y_j if there exists an edge from a vertex in A_{ii} to a vertex in A_{jj} in the graph $\{V, E\}$ of A. In other words, we consider that the entry in position (i, j) of $\widetilde{A}$ is a (usually dense) block of dimension $|Y_i| \times |Y_j|$, where $|X|$ is the cardinality of the set X. Formally, we may define the quotient graph $\mathcal{G}/\mathcal{B} = \{V_\mathcal{B}, E_\mathcal{B}\}$ as

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

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

1. **PREORDERING** Find the block ordering P_B of A such that, upon permutation, the matrix $\widetilde{A} = P_B A P_B^T$ has fairly dense nonzero blocks; see Eq. (4.4). We use a graph compression algorithm proposed by Saad, described in [69], for discovering any perfect or imperfect block structure in A.

2. **SCALING** Scale the matrix at step 1 in the form $S_1 \widetilde{A} S_2$, with two diagonal matrices S_1 and S_2, so that the 1-norm of the largest entry in each row and column is smaller or equal than one.

3. **ORDERING** Find the block independent sets ordering P_I of the quotient graph $\mathcal{G}/\mathcal{B} = \{V_\mathcal{B}, E_\mathcal{B}\}$. Apply the permutation to the matrix obtained at step 2 as

$$P_I S_1 \widetilde{A} S_2 P_I^T = \begin{pmatrix} D & F \\ E & C \end{pmatrix}. \tag{4.5}$$

We use a simple form of the weighted greedy algorithm for computing the ordering P_I. The algorithm is the same as the one used in ARMS, and described in [74]. But it traverses the vertices of the quotient graph $\mathcal{G}/\mathcal{B}$ in the natural order $1, 2, \ldots, n$, marking each visited vertex v and all of its nearest neighbors connected to v by an edge and adding v and each visited node that is not already marked to the independent set. We assign the weight $\|Y\|_F$ to each supervertex Y.

In the 2×2 partitioning Eq. (4.5), the matrix D is still block diagonal as in other forms of multi-elimination ILU [75, 76, 74], but now each diagonal block of D is additionally a sparse block, as D is obtained upon permutation of $\widetilde{A}$. For the same reasoning, the matrices F, E, C also have a block representation.

4. FACTORIZATION Factorize the matrix (4.5) in the form

$$\begin{pmatrix} D & F \\ E & C \end{pmatrix} = \begin{pmatrix} L & 0 \\ EU^{-1} & I \end{pmatrix} \times \begin{pmatrix} U & L^{-1}F \\ 0 & A_1 \end{pmatrix}, \tag{4.6}$$

where I is the identity matrix of appropriate size, and form the reduced system with the Schur complement

$$A_1 = C - ED^{-1}F. \tag{4.7}$$

Based on the block-wise operations and ILU decomposition introduced in Section 4.1, A_1 can be easily calculated via block computational units. According to the block structure and the dimension of D, E, F, C, the Schur complement A_1 is also block sparse and has the same block partitioning of C.

Steps 2-4 can be repeated on the reduced system a few times until the Schur complement is small enough. E.g., after one additional level, we obtain

$$P_I^{(1)} S_1^{(1)} A_1 S_2^{(1)} (P_I^{(1)})^T = \left[\begin{array}{cc|c} D & F_1 & F_2 \\ E_1 & C_{11} & C_{12} \\ \hline E_2 & C_{21} & C_{22} \end{array}\right],$$

which can be factored as

$$\left[\begin{array}{cc|c} L_D & 0 & 0 \\ L_{E_1} & I & 0 \\ \hline L_{E_2} & L_{C_{21}} & I \end{array}\right] \left[\begin{array}{cc|c} D & 0 & 0 \\ 0 & D_{C_{11}} & 0 \\ \hline 0 & 0 & A_2 \end{array}\right] \left[\begin{array}{cc|c} U_D & U_{F_1} & U_{F_2} \\ 0 & I & U_{C_{12}} \\ \hline 0 & 0 & I \end{array}\right].$$

In Figure 4.1, we depict the pattern of the recursive factorization after three levels of reduction on a setup matrix. We can see the shape of the matrix diagonal matrix D, and also the Schur complement gets denser and denser. Moreover, each submatrix still preserves the block structure.

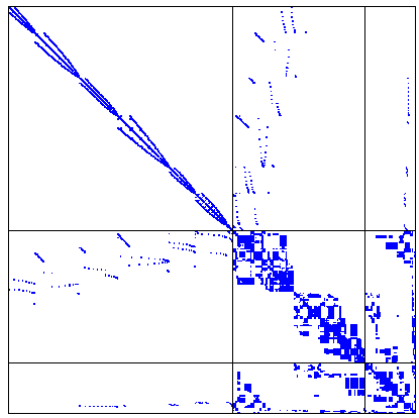


Figure 4.1: Pattern of the recursive factorization of VBARMS after three levels of reduction on a setup matrix.

Denote by A_ℓ the reduced Schur complement matrix at level ℓ, for $\ell > 1$. After scaling and preordering A_ℓ, a system with the matrix

$$P_I^{(\ell)} D_1^{(\ell)} A_\ell D_2^{(\ell)} (P_I^{(\ell)})^T = \begin{pmatrix} D_\ell & F_\ell \\ E_\ell & C_\ell \end{pmatrix} = \begin{pmatrix} L_\ell & 0 \\ E_\ell U_\ell^{-1} & I \end{pmatrix} \times \begin{pmatrix} U_\ell & L_\ell^{-1} F_\ell \\ 0 & A_{\ell+1} \end{pmatrix} \tag{4.8}$$

needs to be solved, with

$$A_{\ell+1} = C_\ell - E_\ell D_\ell^{-1} F_\ell. \tag{4.9}$$

Calling

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

the unknown solution vector and the right-hand side vector of system Eq. (4.8), the solution process with the above multilevel VBARMS factorization consists of level-by-level forward elimination followed by an exact solution on

the last reduced system and suitable inverse permutation. The solving phase is sketched in Algorithm 4.4.

Algorithm 4.4 `VBARMS_Solve`(A_ℓ, b_ℓ). The solving phase of the VBARMS.

Input: $\ell \in \mathbb{N}^*$, $\ell_{max} \in \mathbb{N}^*$, $b_\ell = (f_\ell, g_\ell)^T$, the preconditioning system which contains all the submatrices needed,
Output: the solution y_ℓ.
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 $U_\ell y_\ell = \left[y - L_\ell^{-1} F_\ell z_\ell\right]$

The Algorithm 4.4 is the standard V-cycle process, it aims at solving this linear system $A_\ell x_\ell = b_\ell$. The input is the preconditioning system starting from level ℓ and right hand vector b_ℓ; output is the solution x_ℓ. The algorithm is a recursive process: Step 1 and 2 map the b_ℓ to the next level, and then it continues to go to next level until we reach the last level. Then solve the last level linear system by an approximate LU factorization and get the solution. Following the operators, the last step is to map the solution level by level back to level ℓ. All the matrix-vector products and LU solves are block-wise operations, which are introduced in Section 4.1.

There are two different implementations we have tested. The first implementation we will introduce is based on the explicit calculation of $E_\ell \bar{U}_\ell^{-1}$ and $\bar{L}_\ell^{-1} F_l$ appearing in Eq. (4.8). The second one is based on the implicit calculation of $E_\ell \bar{U}_\ell^{-1}$ and $\bar{L}_\ell^{-1} F_l$, both of them will be introduced in Section 4.2.2.

4.2.2 Computational aspects of the method

Explicit Schur complement calculation

The implementation of the VBARMS method is developed in the C language and is based upon the implementation of ARMS available in the ITSOL package [52]. The compressed sparse storage format (listing 4.1) of ARMS is adapted to store block vectors and matrices as a collection of

contiguous nonzero dense blocks. In our implementation, the approximate transformation matrices $E_\ell \bar{U}_\ell^{-1}$ and $\bar{L}_\ell^{-1} F_l$ appearing in Eq. (4.8), at step ℓ, are temporarily stored in the VBCSR (listing 4.2) format, but they are discarded from the memory after assembling $A_{\ell+1}$. We only store the factors $\bar{L}_\ell, \bar{U}_\ell$ and sub-matrices E_ℓ, F_ℓ at each reduction ℓ, and $\bar{L}_S, \bar{U}_S$, because these are the matrices needed in the solving phase, as it can be seen in Algorithm 4.4. 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.

In the numerical experiments reported in Chapter 4, we use a simple form of weighted greedy algorithm (Algorithm 3.7) on the quotient graph of $\mathcal{G}/\mathcal{B} = \{V_\mathcal{B}, E_\mathcal{B}\}$ for finding block independent sets in step 2. However, other options are possible such as the nested dissection ordering [41] or mesh partitioning[18]. The nested dissection technique repeatedly splits the graph of A into two separate subgraphs, such that there is no coupling between nodes of the two subgraphs. The process is repeated recursively on each of the two subgraphs until a desirable block-size is reached. The last level subgraphs computed represent the block independent sets.

Step 3 of the complete VBARMS process describes an exact factorization process. Due to fill-in caused by the elimination procedure, the reduced systems become denser when the number of levels increases. Therefore, we perform the factorization in VBARMS approximately. After each reduction, we drop small blocks $B_{ij} \in \mathbb{R}^{m_i \times n_j}$ in Schur complement whenever $\frac{\|B\|_F}{m_i \cdot n_j} < t$, for a given user-defined threshold t. We use incomplete LU factorization to invert the matrix D. The same threshold is applied in all these operations.

By exploiting the block structure of D, F, E, C and A_1 in Eq. (4.5-4.6), we use block ILU factorization Algorithm 4.1 to invert both the upper left-most matrix D at each level and the last-level Schur complement, and we use high-level BLAS operations to assemble the Schur complement. This implementation aspect is key to performance and differs from other standard multilevel ILU methods. Clearly, the option to solve the reduced system directly would be too expensive both in terms of computation and memory because the reduced system is likely to be large. Pivoting is not performed during the factorization. The transformation matrices $E_l \bar{U}_l^{-1}$ and $\bar{L}_l^{-1} F_l$, where we denote by $\bar{L}_l$ and $\bar{U}_l$ the lower and upper approximate block triangular factors of D_l, are temporarily stored as a succession of sparse matrices in the VBCSR format, but they are not kept. We need not save the in-

termediate Schur complements, except only the last one that is assembled. Finally, VBARMS explicitly permutes the matrix after step 1 at the first level as well as the matrices involved in the factorization at each new reordering step and stores the global permutation which is the product of all these successive permutations.

Implicit Schur complement calculation

This implementation of the $2\times$ 2 block ILU factorization is different as we describe below. The major advantage of the implicit implementation is that it calculates $E_\ell \bar{U}_\ell^{-1}$ $\bar{L}_\ell^{-1} F_l$ and Schur complement implicitly, which enables the code to avoid to store a lot of temporary matrices.

Algorithm 4.5 *General ILU Factorization, IKJ Version.*

Input: A nonzero pattern set $\mathcal{P}$

1: **for** $i = 2, \ldots, n$ **do**
2: **for** $k = 1, \ldots, i-1$ **do**
3: **if** $(i,k) \notin \mathcal{P}$ **then**
4: $a_{ik} = a_{ik}/a_{kk}$
5: **for** $j = k+1, \ldots, n$ **do**
6: **if** $(i,j) \notin \mathcal{P}$ **then**
7: $a_{ij} = a_{ij} - a_{ik}a_{kj}$

As we mentioned before, the implementation of the VBARMS method is written in the C language and is derived from the ARMS code. The compressed sparse storage format of ARMS is modified to store block vectors and matrices of variable size as a collection of contiguous nonzero dense blocks (the VBCSR data storage format). However, the implementation introduced here differs from the one described in the previous section and is noticeably faster. In our explicit implementation, the approximate transformation matrices $E_\ell \bar{U}_\ell^{-1}$ and $\bar{L}_\ell^{-1} F_l$ appearing in Eq. (4.8) 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 implicit 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 GE algorithm (Algorithm 4.5), where index I runs from 2 to m_ℓ (the size of D_ℓ), index K

from 1 to $(I-1)$ and index J from $(K+1)$ to n_ℓ (the size of A_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, GE 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 the graph compression algorithm at the first level as well as the matrices involved in the factorization at each new reordering step.

4.3 Numerical experiments on a sequential computer

4.3.1 The performance of VBARMS on block structured matrices

Let us use Table 4.2 to recall the matrices and their block structure introduced in Section 2.2. With the help of the angle-based compression algorithm, we are able to build a perfect block structure or an imperfect block structure on these matrices.

Table 4.2: Block structure of test matrix problems.

Name	τ	b-size	b-density (%)	τ	b-size	b-density (%)
RAE	1.00	4.00	96.89	0.80	4.67	95.83
STACOM	1.00	4.11	97.10	0.80	4.36	95.97
BCSSTK35	1.00	4.57	100.00	0.90	5.07	99.29
BMW7ST	1.00	4.63	100.00	0.90	5.28	99.24
CT20STIF	1.00	2.61	100.00	0.90	3.47	96.61
K3PLATES	1.00	5.02	100.00	1.00	5.02	100.00
NASASRB	1.00	2.20	100.00	0.90	3.31	92.31
OILPAN	1.00	2.45	100.00	0.80	2.63	99.73
OLAFU	1.00	1.54	100.00	0.90	5.10	89.50
PWTK	1.00	4.67	100.00	0.90	5.48	99.04
RAEFSKY3	1.00	8.00	100.00	1.00	8.00	100.00
S3DKQ4M2	1.00	1.25	100.00	0.70	5.93	90.34
VENKAT01	1.00	4.00	100.00	1.00	4.00	100.00

After building blocks in the matrix, we can run block solvers upon that. In

our experiments, we compared the VBARMS preconditioner with the original ARMS code [74], the standard ILUT methods [67] and variable block ILUT (Algorithm 4.1 and [69]). For these methods, we used the implementations available in the ITSOL package [52]. We applied the VBARMS and block ILUT preconditioners to the compressed matrix $\widetilde{A}$ Eq. (4.4), while ARMS and ILUT were applied to A, as they do not exploit explicitly any block structure of the matrix. No special ordering was used in the computation of ILUT. We chose not to apply ILUT to $\widetilde{A}$, because earlier studies have clearly illustrated the better performance of block versions over point versions of incomplete factorization preconditioners for solving block structured systems, see, e.g., [25, 69].

To reduce the condition number, we scaled the system by rows and columns prior to the iterative solution so that the modulus of each entry of the scaled coefficient matrix was smaller than one. We solved $S_1 A S_2 y = S_1 b$, $x = S_2 y$ where S_1 and S_2 are the diagonal matrices

$$S_1(i,j) = \begin{cases} \frac{1}{\|A(i,:)\|_1} & , \text{ if i } = \text{ j} \\ 0 & , \text{ if i } \neq \text{ j} \end{cases}, \quad S_2(i,j) = \begin{cases} \frac{1}{\|A(:,j)\|_1} & , \text{ if i } = \text{ j} \\ 0 & , \text{ if i } \neq \text{ j} \end{cases},$$

We used physical right-hand sides b when these were available, otherwise we set $b = Ae$ where $e = [1, \ldots, 1]^T$.

For every run, we recorded the solution time from the start of the solve until either the initial residual was reduced by six orders of magnitude or the process failed. We declared a solver failure when no convergence was achieved after 1,000 iterations of the flexible GMRES (FGMRES) method [66] restarted every 60 inner iterations, or when there was a breakdown during the factorization. We selected the parameters carefully to have a fair comparison in our experiments. In this chapter, the implementation we used for Section 4.3.1 and 4.3.2 is the implementation of explicit Schur complement calculation, for the rest is the implementation of implicit Schur complement calculation.

For each matrix problem, we tested different values for the dropping parameter t in VBARMS, starting from $t = 0.1$ and decrementing it by a factor of 10 in each run; we selected the value of t which gave the best convergence result for the given problem. The maximum number of nonzero entries per row/column of the preconditioner was uniformly set equal to the dimension of the problem. We chose the value of the dropping threshold in the ARMS

and in the ILUT codes which led to roughly the same number of nonzeros as in the VBARMS preconditioner. Finally, the number of levels of recursive factorization in VBARMS and ARMS were calculated automatically by the two codes, which stop when the Schur complement becomes too small to continue reducing the matrix. The maximum allowed size for the last level Schur complement matrix was set to 300. This value also determines the minimum size of the independent sets in the greedy algorithm. The computed block ordering was used in our experiments with the VBARMS and the block ILUT method (denoted shortly as BILUT).

For each experiment shown in Table 4.3, we report the following performance measures:

1. the time cost for computing the factorization (column "P-T") and for solving the linear system (column "I-T"), the unit is second. The construction time for VBARMS and BILUT includes the cost for computing the block ordering ("B-T").

2. the memory burden (column "M-cost"), computed as the ratio of the total number of nonzeros in the preconditioning system to the number of nonzeros in the coefficient matrix A;

3. the number of iterations (column "Its") required by the FGMRES method to reduce the initial residual by six orders of magnitude;

The results highlight the robustness of the VBARMS preconditioner. It is remarkable that fast and stable convergence was obtained using a simple greedy algorithm for finding the independent sets and without the need of monitoring the growth in the factors during the factorization. This is probably due to the better control of near-singularities of block ILU solvers, and to the better conditioning of the Schur complement matrices that are smaller and easier to invert.

We also used nonsymmetric permutations aimed at improving the diagonal dominance of the local matrices, namely the "ddPQ" [71] algorithm available in the ITSOL package [52], but in our tests the greedy algorithm performed decidedly better than the "ddPQ" ordering, see Table 4.4. We selected the matrices where ARMS is able to converge on a reasonable memory cost, and use the same parameter setting to perform greedy algorithm +

Table 4.3
Performance comparison of the three methods. The symbol "†" in the "P-T" column means that a breakdown occurred during the factorization. The symbol "‡" in the "I-T" column means that the iterative solution was not started because the preconditioner was ill-conditioned.

Matrix	Method	P-T(B-T)	I-T	M-cost	Its
RAE	VBARMS	4.250 (0.050)	2.530	2.430	46
	BILUT	2.690 (0.050)	>46.090	3.886	>1000
	ARMS	5.800	>46.010	3.750	>1000
	ILUT	†	‡	–	–
STACOM	VBARMS	0.600 (0.000)	0.250	2.707	28
	BILUT	0.170 (0.000)	1.730	2.930	287
	ARMS	1.010	‡	3.112	–
	ILUT	0.320	‡	2.899	–
BCSSTK35	VBARMS	6.930 (0.020)	8.440	3.160	175
	BILUT	1.710 (0.020)	>32.420	3.721	>1000
	ARMS	7.040	>34.760	3.430	>1000
	ILUT	5.410	>32.950	3.328	>1000
BMW7ST	VBARMS	42.020 (0.130)	0.930	3.079	4
	BILUT	9.480 (0.130)	>146.100	3.076	>1000
	ARMS	49.230	>172.750	3.112	>1000
	ILUT	34.940	‡	3.085	–
CT20STIF	VBARMS	6.920 (0.070)	1.640	1.265	22
	BILUT	0.860 (0.070)	>52.500	1.374	>1000
	ARMS	2.610	>43.250	1.392	>1000
	ILUT	3.300	‡	1.455	–
K3PLATES	VBARMS	0.760 (0.000)	0.450	2.388	38
	BILUT	0.180 (0.000)	>7.170	2.476	>1000
	ARMS	0.540	>11.600	2.528	>1000
	ILUT	0.560	>6.840	2.464	>1000
NASASRB	VBARMS	13.940 (0.080)	11.030	2.412	94
	BILUT	3.080 (0.080)	>76.760	2.827	>1000
	ARMS	18.950	>74.240	3.026	>1000
	ILUT	7.080	>69.250	4.014	>1000
OILPAN	VBARMS	10.270 (0.050)	1.790	2.825	21
	BILUT	2.020 (0.050)	>51.470	3.230	>1000
	ARMS	13.830	>48.250	2.925	>1000
	ILUT	6.200	>47.040	2.954	>1000
OLAFU	VBARMS	2.670 (0.030)	0.900	2.109	34
	BILUT	0.730 (0.030)	3.700	2.381	208
	ARMS	2.110	‡	2.230	–
	ILUT	1.240	>16.280	2.335	>1000
PWTK	VBARMS	50.590 (0.180)	32.790	2.669	93
	BILUT	9.870 (0.180)	37.500	3.013	164
	ARMS	39.370	>260.880	2.963	>1000
	ILUT	44.540	‡	3.038	–
RAEFSKY3	VBARMS	2.930 (0.020)	0.260	1.906	10
	BILUT	1.010 (0.020)	0.230	2.452	13
	ARMS	2.050	25.190	2.529	>1000
	ILUT	1.960	0.210	2.051	10
S3DKQ4M2	VBARMS	14.660 (0.150)	7.160	2.667	55
	BILUT	3.970 (0.150)	8.950	3.350	104
	ARMS	14.850	>100.570	2.781	>1000
	ILUT	5.080	>82.020	2.664	>1000
VENKAT01	VBARMS	0.810 (0.040)	1.040	0.493	40
	BILUT	0.170 (0.040)	1.100	0.577	46
	ARMS	0.340	0.590	0.456	28
	ILUT	0.190	0.510	0.469	32

ARMS and ddPQ + ARMS respectively. We can see from the table, ddPQ can not improve ARMS performance on these matrices.

Table 4.4
Performance comparison of greedy algorithm (GA) and ddPQ.

Matrix	Method	P-T	I-T	M-cost	Its
OILPAN	ARMS+GA	17.24	7.95	5.45	154
	ARMS+ddPQ	13.77	10.63	7.05	179
K3PLATES	ARMS+GA	0.43	0.76	4.69	106
	ARMS+ddPQ	0.59	0.90	8.73	111
OLAFU	ARMS+GA	3.26	5.58	6.93	302
	ARMS+ddPQ	3.90	7.86	7.84	403
RAEFSKY3	ARMS+GA	5.07	0.05	4.01	3
	ARMS+ddPQ	10.38	>29.40	8.99	>1000
VENKAT01	ARMS+GA	2.51	0.15	5.00	4
	ARMS+ddPQ	4.75	0.20	12.31	4

In Table 4.5, we particularly showed the comparison between VBARMS and ARMS on "r-ratio" and "MFlops", their definitions are beneath:

1. the reduction factor (column "r-ratio"), it gives the ratio of the sum of the number of unknowns at all levels of the factorization to the number of unknowns in the original system. Clearly, this performance metric only applies to the multilevel methods VBARMS and ARMS;

2. the megaflop rate (column "MFlops") achieved by the three codes for the construction of the preconditioner, estimated by the PAPI library [82] on a PC with Intel Core i3 processor with a clock speed of 2.53 GHz and 2 GB of main memory.

The VBARMS code achieved higher MFlops than ILUT and ARMS. This is due to the use of the level-3 BLAS operations in the local solvers and in computing the Schur complement. And the advantage of level-3 BLAS routines are displayed in the Table 4.1. On the problem `BCSSTK35`, using PAPI we obtained 236.863 MFlops for VBARMS against 94.597 for ARMS, and 394.92 MFlops for the optimized sparse direct solver SuperLU [53].

A lower "r-ratio" means VBARMS or ARMS generate a smaller Schur complement which shows better performance as a multi-level method. The results from Table 4.5 confirm better *complexity*. That is because we work

Table 4.5
The column "MFlops" refers to the megaflop rate for the construction of the preconditioner.

Matrix	Method	r-ratio	MFlops
RAE	VBARMS	1.270	208.742
	ARMS	3.210	67.087
STACOM	VBARMS	1.220	208.169
	ARMS	2.551	80.755
BCSSTK35	VBARMS	1.424	236.863
	ARMS	2.438	94.597
BMW7ST	VBARMS	1.503	238.262
	ARMS	2.442	59.179
CT20STIF	VBARMS	1.550	129.593
	ARMS	1.774	83.356
K3PLATES	VBARMS	1.221	213.257
	ARMS	1.263	98.711
NASASRB	VBARMS	1.481	156.265
	ARMS	1.633	58.844
OILPAN	VBARMS	1.279	222.714
	ARMS	1.547	96.078
OLAFU	VBARMS	1.351	220.436
	ARMS	1.563	122.646
PWTK	VBARMS	1.399	245.025
	ARMS	2.569	82.735
RAEFSKY3	VBARMS	1.262	281.705
	ARMS	2.867	80.353
S3DKQ4M2	VBARMS	1.286	254.218
	ARMS	1.519	104.323
VENKAT01	VBARMS	1.197	258.346
	ARMS	1.203	18.4576

on the blocks. The block independent set ordering is built upon the quotient graph, which generates bigger B and smaller C in Eq. 4.5.

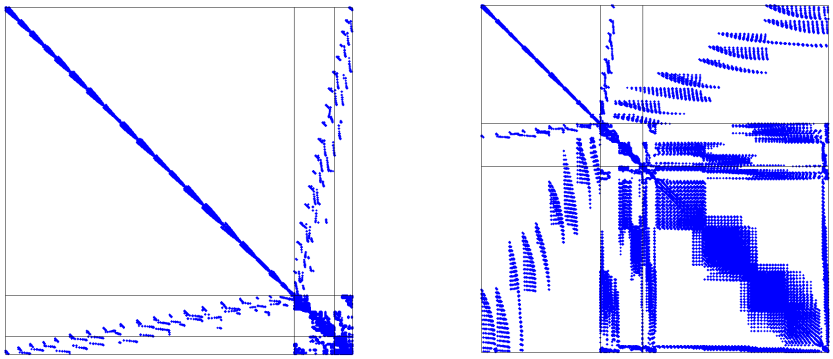


Figure 4.2: Sparsity patterns of the recursive multilevel factorization using the VBARMS method (on the left) and the ARMS method after three levels of reduction on the `STACOM` problem. We observe that the Schur complement is considerably smaller for VBARMS than for ARMS.

In Figure 4.2 we plot the sparsity pattern of the recursive multilevel factorization for the `STACOM` problem after three levels of reduction; we can see that the Schur complement is considerably smaller for VBARMS (on the left) than for ARMS (on the right). In our experiments, we observed that the triangular factors computed by VBARMS were well conditioned; consequently, the triangular solves were numerically stable.

4.3.2 Experiments on unstructured matrices

Up to now, we only computed and applied the block ordering to the original system A. This strategy works on the naturally block structured matrices. For unstructured ones, alternatively, we also tried to block only the Schur complement matrix, which is significantly denser than A and therefore has a higher chance to be compressed satisfactorily. Therefore, after one step of reduction with ARMS we switched to VBARMS. The results show that this hybrid approach may improve the performance of ARMS in terms of convergence and/or the memory/construction costs to some extent. The

first three matrices of Table 4.6 do not have any block structure. The `XENON1` problem revealed fully dense blocks of average size equal to 2.45; by applying the block ordering to A, at roughly equal memory costs VBARMS converged in 26 iterations and 2.310 seconds, but the construction was twice more expensive (21.850 seconds).

However, we encountered systems where the Schur complement matrix after one level of factorization was still quite sparse and/or irregularly structured, and the graph compression algorithm produced inefficient block orderings with low density. It would be natural to compress the Schur complement matrix at further levels, as it tends to fill-in. However, for this approach to work it is necessary that the factors computed by ARMS are stable and accurate, because VBARMS cannot cure the pathology of an ill-conditioned factorization. For instance, this approach may require to run ARMS with non-symmetric permutations and/or condition estimators. This strategy can be investigated in detail in the future.

Table 4.6: Experiments for general unstructured matrices compressing the Schur complement matrix.

Matrix	Method	P-T	I-T	M-cost	Its
KIM1	VBARMS	16.330	1.360	8.909	18
	ARMS	62.450	1.480	18.401	19
CKT11752	VBARMS	12.390	0.370	11.791	9
	ARMS	8.370	85.380	13.731	2000
TORSO1	VBARMS	53.680	6.420	2.086	41
	ARMS	66.450	5.360	2.673	37
XENON1	VBARMS	10.510	12.230	4.415	220
	ARMS	37.210	121.370	9.801	2000

4.3.3 Performance of the graph-based compression method

We introduced the graph-based compression method in Section 2.3. In this section, we combine it with Variable Block ILU method (VBILUT) and show the performance with respect to the angle-based compression method.

Some comparative figures of performance between the angle-based and the graph-based techniques are presented in Tables 4.7-4.9. In Table 4.7, we first tried to find an optimal value of τ that minimizes 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 optimal value of τ was calculated by performing several runs, each using a different value of τ in the range $[0.5, 1.0]$ by increments of 0.1 at every run. Then, for every problem we set μ equal to the *b-density* corresponding to the optimal value of τ that we found. As we can see in Table 4.7, both methods perform similarly, with the angle method generally doing slightly better probably because the parameters were optimally selected for that method. Once again, the results show that the optimal value for τ may be very problem-dependent.

In the experiments reported in Table 4.8 we use optimal values for both τ and μ, which were computed as explained above. The performance of the two methods is again very similar. Finally, in the results reported in Table 4.9 we set $\mu = 0.7$ for the graph compression method, giving a minimum *b-density* of 70% for every problem. A quick comparison with the results obtained by selecting the optimal τ in the angle method reveals that the new compression algorithm still remains very competitive. The set of problems being fairly large, we may conclude that $\mu = 0.7$ may be an overall good choice for most problems in our method.

In Tables 4.7-4.9, we also report on the timings to compute the block ordering by both compression techniques, and for solving the linear system. The compression time is considerably smaller than the total solution time. On the other hand, the graph method is in most cases up to three times slower than the angle method. However, this is not a big downside because computing the optimal value of τ may require several runs as we explained. We also see from the tables that the compression time increases when μ decreases, which is obvious since we merge more supernodes.

Matrix	Method	τ/μ	Block density (%)	Block size	Blocking time (s)	Total (s)	M-cost	Its
OILPAN	Angle	0.70	95.94	7.36	0.03	4.18	0.26	198
	Graph	0.80	99.78	7.02	0.08	4.23	0.24	200
K3PLATES	Angle	0.60	59.16	7.90	0.00	0.7	0.3	239
	Graph	0.55	61.67	8.20	0.01	0.62	0.27	226
VENKAT01	Angle	0.70	99.94	4.00	0.02	0.43	1.33	9
	Graph	1.00	99.38	4.01	0.08	0.43	1.33	9
PWTK	Angle	0.60	56.95	12.17	0.09	26.38	6.85	117
	Graph	0.54	59.94	10.96	0.38	29.84	5.3	157
S3DKQ4M2	Angle	1.00	100.00	5.93	0.03	9.57	1.09	214
	Graph	0.86	100.00	5.93	0.10	9.59	1.09	213
OLAFU	Angle	0.80	81.75	6.47	0.02	1.2	3.14	54
	Graph	0.73	82.84	6.14	0.11	1.5	3.14	65
RAE	Angle	0.80	95.83	4.67	0.03	8.85	9.53	49
	Graph	0.80	95.70	4.65	0.12	10.31	9.77	63
BMW7ST_1	Angle	0.70	77.16	7.28	0.08	0.35	0.18	5
	Graph	0.67	76.80	6.90	0.29	0.47	0.17	9
NASASRB	Angle	0.80	90.87	4.24	0.05	7.51	5.23	30
	Graph	0.77	90.97	4.15	0.17	14.05	6.8	40
CT20STIF	Angle	0.70	66.05	6.55	0.04	0.69	0.18	44
	Graph	0.60	66.16	5.69	0.17	1.15	0.17	59
RAEFSKY3	Angle	0.70	95.23	8.63	0.01	0.08	0.13	13
	Graph	0.80	96.86	8.40	0.02	0.08	0.12	14
BCSSTK35	Angle	0.60	51.95	11.03	0.01	2.1	0.29	209
	Graph	0.48	51.72	10.46	0.05	2.38	0.29	224
STACOM	Angle	0.90	97.00	4.36	0.00	0.97	9.86	1
	Graph	0.82	96.60	4.36	0.01	0.97	9.87	1

Table 4.7: Experiments with the angle-based and the graph-based compression methods [24]. The optimal value of τ is used for the angle-based algorithm. The value of μ in the graph-based algorithm is selected to give a similar *b-density* as in the angle-based method.

Matrix	Method	τ/μ	Block density (%)	Block size	Blocking time (s)	Total (s)	M-cost	Its
OILPAN	Angle	0.70	95.94	7.36	0.03	4.18	0.26	198
	Graph	0.60	65.72	9.95	0.09	4.01	0.38	186
K3PLATES	Angle	0.60	59.16	7.90	0.00	0.7	0.3	239
	Graph	0.70	89.50	5.65	0.01	0.71	0.18	241
VENKAT01	Angle	0.70	99.94	4.00	0.02	0.43	1.33	9
	Graph	0.90	99.38	4.01	0.08	0.44	1.33	9
PWTK	Angle	0.60	56.95	12.17	0.09	26.38	6.85	117
	Graph	0.60	62.58	10.11	0.37	29.91	5.66	144
S3DKQ4M2	Angle	1.00	100.00	5.93	0.03	9.57	1.09	214
	Graph	0.90	100.00	5.93	0.10	9.57	1.09	214
OLAFU	Angle	0.80	81.75	6.47	0.02	1.2	3.14	54
	Graph	0.70	79.66	6.58	0.11	1.62	3.75	57
RAE	Angle	0.80	95.83	4.67	0.03	8.85	9.53	49
	Graph	0.80	95.70	4.65	0.12	10.61	11.2	38
BMW7ST_1	Angle	0.70	77.16	7.28	0.08	0.35	0.18	5
	Graph	0.60	67.04	7.98	0.30	0.44	0.2	8
NASASRB	Angle	0.80	90.87	4.24	0.05	7.51	5.23	30
	Graph	0.60	62.57	5.29	0.23	10.71	7.84	26
CT20STIF	Angle	0.70	66.05	6.55	0.04	0.69	0.18	44
	Graph	0.60	66.16	5.69	0.17	1.15	0.17	59
RAEFSKY3	Angle	0.70	95.23	8.63	0.01	0.08	0.13	13
	Graph	0.90	100.00	8.00	0.02	0.08	0.11	15
BCSSTK35	Angle	0.60	51.95	11.03	0.01	2.1	0.29	209
	Graph	0.60	65.61	8.15	0.05	2.49	0.22	231
STACOM	Angle	0.90	97.00	4.36	0.00	0.97	9.86	1
	Graph	0.90	96.93	4.23	0.01	0.99	9.74	2
Means of ratio			1.01	1.03	0.26	0.86	1.01	0.89

Table 4.8: Experiments with the angle-based and the graph-based compression methods [24]. Optimal value are used for the parameters τ and μ in the angle-based method and in the graph-based method, respectively. The last row reports the geometric means of the ratio between Angle vs Graph of each performance metric.

Matrix	Method	τ/μ	Block density (%)	Block size	Blocking time (s)	Total (s)	M-cost	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
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
STACOM	Angle	0.90	97.00	4.36	0.00	0.97	9.86	1
	Graph	0.70	84.51	4.47	0.01	1.39	11.9	2

Table 4.9: Experiments with the angle-based and the graph-based compression methods [24]. 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.

4.3.4 Performance comparison of the two implementations

The improvement of efficiency obtained with the implicit implementation is remarkable as shown in Table 4.10. Implicit implementation gains a lot on time cost (the highlighted colum "P-T") of preconditioner setup. That is

because it saves the storage of temporary matrices during the calculation of Schur complement, and avoids a lot of memory traffic, especially when the matrix gets bigger like PWTK and BMW7ST.

Matrix	Method	P-T	I-T	Total (s)	M-cost	Its
STACOM	Implicit	0.56	0.06	0.62	4.62	9
	Explicit	0.54	0.09	0.63	4.63	7
NASASRB	Implicit	2.34	4.64	6.98	1.97	85
	Explicit	12.12	5.36	17.48	2.22	92
OILPAN	Implicit	0.72	0.67	1.39	3.28	23
	Explicit	4.59	0.66	5.25	3.43	19
BCSSTK35	Implicit	0.63	3.26	3.89	2.43	156
	Explicit	5.78	1.39	7.18	2.53	72
K3PLATES	Implicit	0.21	0.71	0.92	2.57	133
	Explicit	0.40	0.48	0.89	2.61	75
RAE	Implicit	2.48	0.50	2.99	3.84	14
	Explicit	9.43	0.41	9.84	4.00	10
VENKAT01	Implicit	1.09	0.17	1.25	2.56	5
	Explicit	3.85	0.38	4.24	2.85	4
RAEFSKY3	Implicit	0.32	0.05	0.38	1.95	5
	Explicit	1.28	0.13	1.41	2.02	2
CT20STIF	Implicit	1.46	1.42	2.88	2.08	31
	Explicit	24.24	0.86	25.10	2.33	17
OLAFU	Implicit	0.38	1.09	1.46	1.88	99
	Explicit	1.41	1.16	2.57	2.02	92
BMW7ST_1	Implicit	4.63	0.13	4.75	2.94	1
	Explicit	49.64	1.75	51.39	3.22	1
S3DKQ4M2	Implicit	2.22	10.18	12.39	2.46	178
	Explicit	6.51	7.42	13.93	2.60	128
PWTK	Implicit	5.23	31.75	36.98	2.40	196
	Explicit	36.92	24.50	61.42	2.63	153

Table 4.10: Comparative experiments [24] with implementing a different partial (block) factorization step in VBARMS. Implicit is the implementations of implicit Schur compliment calculation and explicit implementations of explicit Schur compliment calculation

5 Parallelization strategy and experiments

5.1 Introduction to parallel computing

In the past decades, the world went through a very exciting phase on computer hardware, passing from single core Central Processing Unit (CPU) to multiple-core CPU, and from CPU to Graphics Processing Unit (GPU). This transition has led to dramatic performance improvements. Trends indicate that the progress will continue in the coming decades. The boost for these progress is the emergence of microprocessor technology. Microprocessors are smaller, and one chip can contain multiple CPUs. Thus, this microprocessor technology led to the evolution of the larger parallel computer.

Today, even our office desktops start to have a parallel computer architecture. Because of the advent of dual-core and quad-core computers and the expected increase in the number of cores, this may change the way of software design, such as the mathematical software we use in scientific computations. Moreover, the paper [81] points out that concurrency will be even a next revolution in how we write software, just like how Object Oriented Programming influenced the software industry.

Unfortunately, the software development has not kept pace with the hardware advances yet. In order to solve a problem efficiently on a parallel machine, it is usually necessary to design an algorithm that specifies multiple operations on each step, i.e., a parallel algorithm.

Traditionally, computer software was designed for serial computing, where an algorithm is composed by a serial stream of instructions. These instructions are executed on one processing unit on one computer. Moreover, one instruction only can be executed at a time, and one after another, see also [9].

Parallel computing uses multiple processing elements simultaneously to solve a problem. We break the whole process into independent parts, and instructions from each part executes concurrently on different processors. In order to utilize parallel computing techniques, there are some requirements

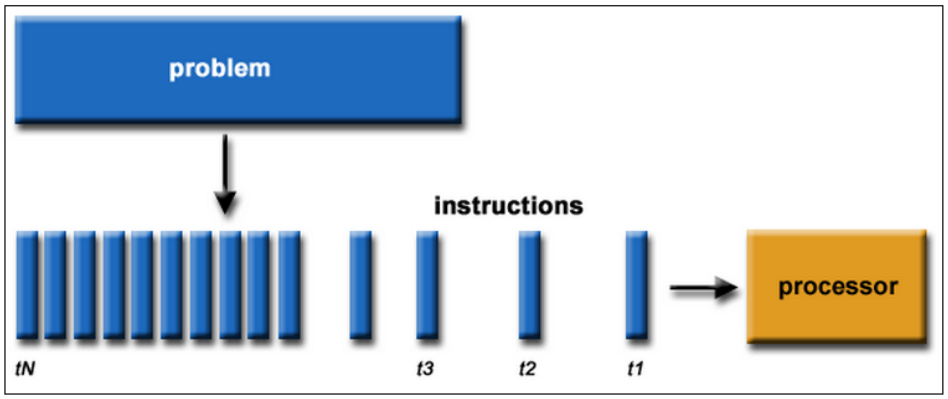


Figure 5.1: A typical serial process [9].

for the problem and compute resource.

The computational problem must be parallelizable, which means it should be amenable to be broken apart into discrete pieces of work that can be computed simultaneously; the compute resources are typically a single computer with multiple processors/cores or a cluster composed by several such computers connected by a network.

It is important to recall Flynn's taxonomy [44] to help understand the parallel workflow of a computer program. The taxonomy of computer systems proposed by M. J. Flynn in 1966 is the most widely used classification for parallel computers. Flynn introduced the concept of instruction and data streams for categorizing of computers. So it can also be considered as a classification of parallel software workflow.

1. **SISD**
   - Single Instruction Stream, Single Data Stream.
   - Conventional sequential computer (von Neumann architecture) i.e. uniprocessor
   - The software that matches this type of computer is the traditional serial software, see Fig 5.3 for a serial workflow.
2. **SIMD**

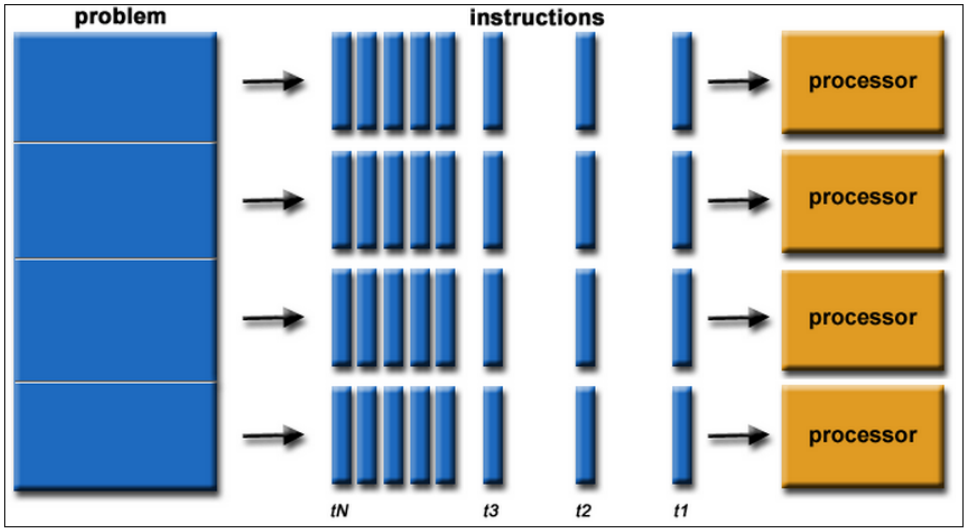


Figure 5.2: A typical parallel process [9].

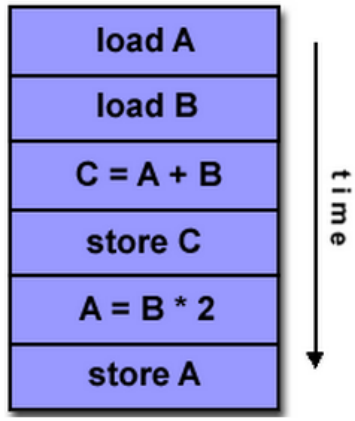


Figure 5.3: A SISD workflow [9].

- Single Instruction Stream, Multiple Data Stream.
- Most modern computers, particularly those with graphics proces-

sor units (GPUs) employ the SIMD architecture.

- All processing elements are concurrently executing the same instruction but on a different data stream at the same time; hence the name SIMD.
- Synchronous (lockstep) execution.
- Analogy in the life: multiple students in a dance class carry out the instructor's instruction one by one simultaneously.

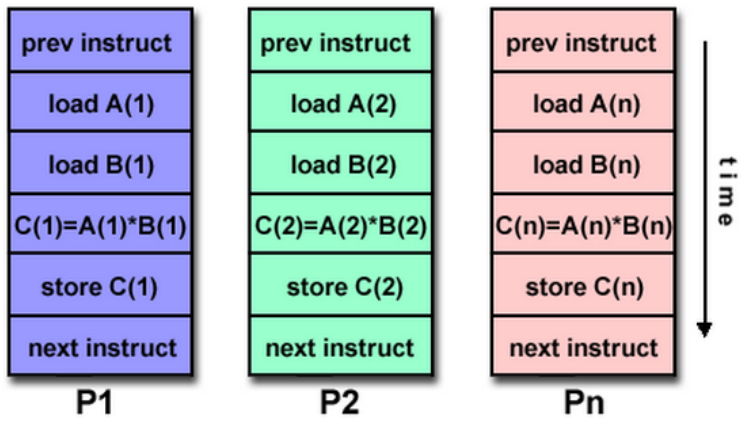


Figure 5.4: A SIMD workflow [9].

3. **MIMD**
   - Multiple Instruction Stream, Multiple Data Stream.
   - Execution can be synchronous or asynchronous.
   - A MIMD is a true multiprocessor.
   - Currently, the most common type of parallel computer - most modern supercomputers belong to this category.
   - Many MIMD architectures also include SIMD execution subcomponents.
4. **MISD**
   - Multiple Instruction Stream, Single Data Stream.

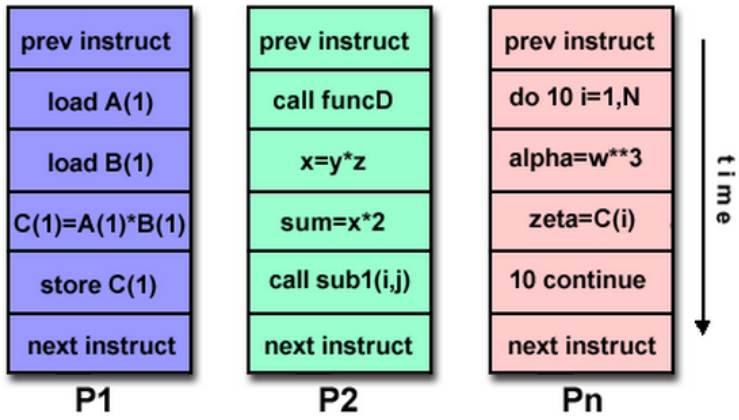


Figure 5.5: A MIMD workflow [9].

- Few actual examples of this category of parallel computer have ever existed.
- Some examples could be pipeline-wise computation, like multiple frequency filters operating on a single signal stream, multiple cryptography algorithms attempting to crack a single coded message..

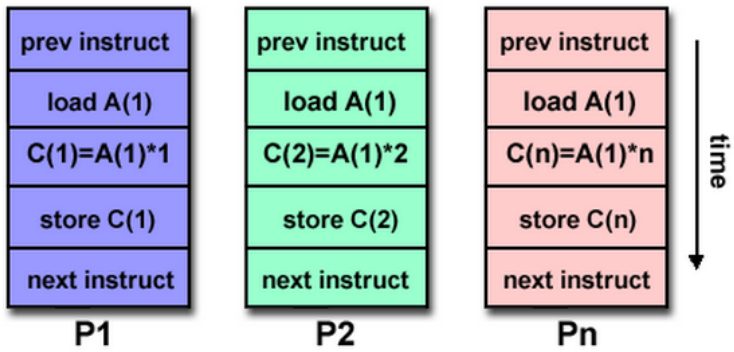


Figure 5.6: A MISD workflow [9].

Corresponding to the parallel computer architecture used, there are also different types of programming model for parallel software. In this thesis, we only introduce the most widely used models; the thread model and the distributed memory programming model.

1. **Thread model**
   - Thread model is a type of shared memory programming.
   - A thread is the smallest processing unit that can be performed in an OS. In most modern operating systems, a thread exists within a process - that is, a single process may contain multiple threads.
   - Threads communicate with each other via shared global memory. This requires synchronization constructs to ensure that more than one thread is not updating the same global address at any time.
   - From a programming perspective, there are two popular implementations of threads: POSIX Threads (Pthreads) and OpenMP.
   - POSIX Threads is a standardized C/C++ language threads programming interface. It is a library based module that can only be called in parallel source code. See more details in [11].
   - OpenMP is a compiler directive based Application Program Interface (API). The API supports C/C++ and Fortran on a wide variety of architectures. It can be easy and simple to use from programmers perspective, see also [10].
   - Normally, POSIX Threads and OpenMP parallel programs have both SIMD and MIMD workflow.
2. **Distributed memory model**
   - A set of computational tasks can have their own local memory during operations. The computer should be either a multicore machine or a group of computers connected by a network.
   - Tasks exchange data through communications by sending and receiving messages.
   - From a programming perspective, message passing implementations usually comprise a library of subroutines. It is due to the programmer to use these subroutines and to determine the parallelism in the source code.

- **Message Passing Interface (MPI)** is an interface specification for the developers and users of message passing libraries.
- It supports several different computer programming languages such as Fortran, C/C++ and Java.

For the distributed memory computing model, the matrix computation software package we have been working on is a MPI-based library, Parallel Algebraic Recursive Multilevel Solver (pARMS) [54]. MPI-based program is also a type of SIMD model, that is single program performs multi-tasks on multi-cores. We will introduce this software in the next section.

5.2 Parallel implementation of VBARMS

pARMS is a MPI-based library of parallel solvers for distributed sparse systems of linear equations. It adopts a distributed sparse parallel framework. This viewpoint generalizes domain decomposition approach to partition an irregularly structured sparse matrix into submatrices. It is common to partition a physical mesh by a graph partitioner like METIS [49] and assign a group of elements which represent a physical subdomain to each processor. Every processor only assembles the equations attached to the local elements, it will eventually end up with a set of equations (rows of the linear system) and a vector of the variables associated with these rows. This is the classic way of distributing a sparse linear system. If the graph form is a matrix, then it will be sending rows of the matrix and the corresponding right hand side variables to each processor.

Following the parallel framework described in [54], in this study we distinguish the local nodes of the quotient graph into *interior nodes*; those coupled only with local variables by the equations, and *interface nodes*, those that may be coupled with local variables stored on processor i and those with remote variables stored on other processors (see Figure 5.7).

In our case, the integration of VBARMS within this parallel framework generate a block wise distributed sparse linear system by distributing block rows and corresponding variables in parallel. So from graph viewpoint, it is natural to split the quotient graph $\mathcal{G}/\mathcal{B}$ into separate subdomains, each assigned to a different processing unit.

The vector of the local unknowns x_i and the right-hand side b_i are also split in two separate components: the subvector corresponding to the internal

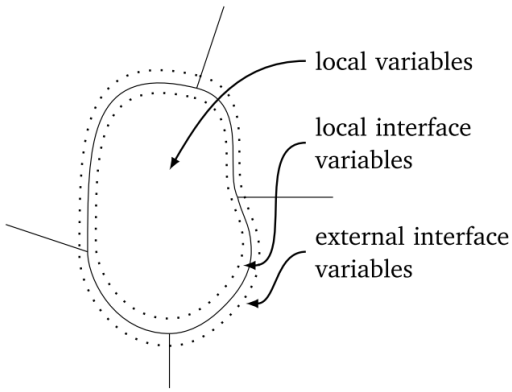


Figure 5.7: Local domain from a physical viewpoint.

nodes followed by the subvector of the local interface variables

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

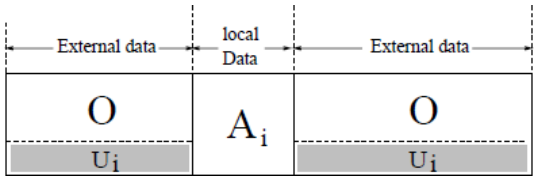


Figure 5.8: A partitioned local sparse matrix.

As can be seen in Figure 5.8, the rows of the matrix assigned to the ith processor are 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 the local equations on processor i write as

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

or, in expanded form, as

$$A_i = \left(\begin{array}{c|c} B_i & F_i \\ \hline E_i & C_i \end{array} \right) \tag{5.1}$$

$$\begin{pmatrix} B_i & F_i \\ E_i & C_i \end{pmatrix} \begin{pmatrix} u_i \\ y_i \end{pmatrix} + \begin{pmatrix} 0 \\ \sum_{j \in N_i} E_{ij} y_j \end{pmatrix} = \begin{pmatrix} f_i \\ g_i \end{pmatrix}, \tag{5.2}$$

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 continue to possess the finest block structure imposed by the block ordering P_B.

In its simplest parallel implementation, we apply the sequential VBARMS method to invert approximately each matrix A_i without subdomain overlapping (Block-Jacobi preconditioner) or with overlapping (Restricted Additive Schwarz preconditioner). The 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 case we have a graph partitioned matrix, we may define a block-Jacobi iteration in a similar way with D block diagonal. This is the case of our parallel implementation of the Block-Jacobi preconditioner that writes as Algorithm 5.1. Clearly, there is high degree of parallelism in this approach since the solves with the matrices A_i are performed independently on all processors.

Algorithm 5.1 *Block-Jacobi preconditioning.*

1: Obtain the remote variables $y_{i,ext}$
2: Compute $r_i = (b - Ax)_i$
3: Solve $A_i\delta_i = r_i$ approximately using the sequential VBARMS method
4: Update $x_i = x_i + \delta_i$

If the subdomains are allowed to overlap, the resulting preconditioner is called Schwarz preconditioner. Again consider we have a graph partitioned matrix, and say the graph partitioning resulted in N nonoverlapping sets W_i^0 with $i = 1, \ldots, N$ and $W_0 = \bigcup_{i=1}^{N} W_0^i$. We define a k-overlap partition

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

where $W_i^k = adj\left(W_i^{k-1}\right)$ with $k > 0$ the levels of overlap with neighboring domains. Associated with each subdomain we define a restriction operator

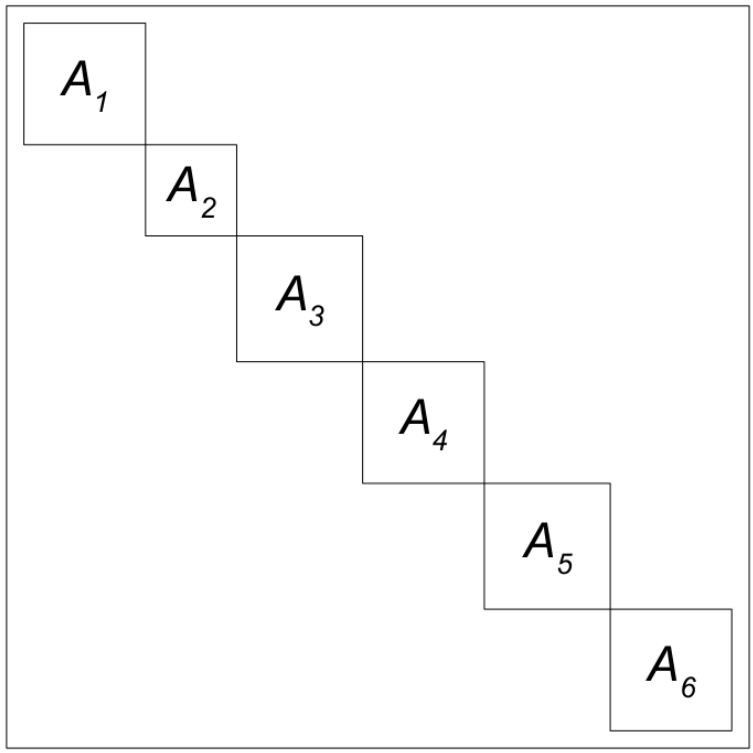


Figure 5.9: The block-Jacobi matrix without overlapping blocks.

R_i^k, which is an $n \times n$ matrix with the (j,j)th element equal to 1 if $j \in W_i^k$ and zero elsewhere. We now denote

$$A_i = R_i^k A R_i^{Tk}.$$

The preconditioning matrix M_{RAS} is defined as

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

and the Restricted Additive Schwarz preconditioner (RAS) [19, 70, 39] writes as Algorithm 5.2.

Algorithm 5.2 *Restricted Additive Schwarz preconditioning.*

1: Obtain the remote variables $y_{i,ext}$
2: Compute $r_i = (b - Ax)_i$
3: Solve $R_i^T A_i^{-1} R_i \delta_i = r_i$ using the sequential VBARMS method to invert approximately A_i
4: Update $x_i = x_i + \delta_i$

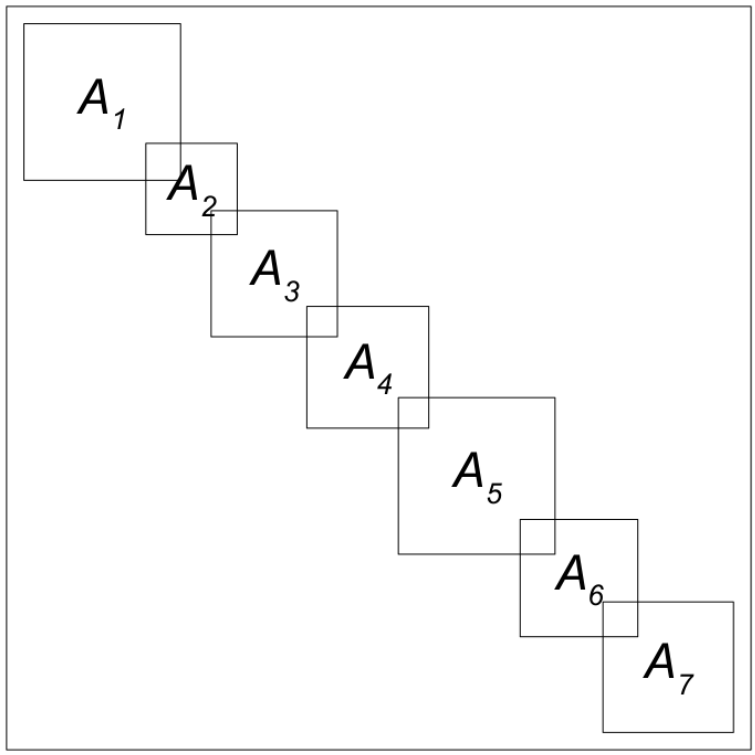


Figure 5.10: The block-Jacobi matrix with overlapping blocks.

Note that the preconditioning step is still parallel and consists of forming the different components of the error update. However, in this case of the

overlapping, the components are added up from the different results obtained in each subdomain. Therefore some communication is required.

We also consider a parallel implementation of VBARMS based on the Schur complement approach [89]. In Eq. (5.2), we can eliminate the vector of interior unknowns u_i from the first equations to obtain 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 is the local Schur complement matrix

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

Writing all the local Schur complement equations together results in the global Schur complement system:

$$\begin{pmatrix} S_1 & E_{12} & \dots & E_{1p} \\ E_{21} & S_2 & \dots & E_{2p} \\ \vdots & & \ddots & \vdots \\ E_{p1} & E_{p-1,2} & \dots & S_p \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{pmatrix} = \begin{pmatrix} g_1' \\ g_2' \\ \vdots \\ g_p' \end{pmatrix}, \tag{5.3}$$

where the off-diagonal matrices E_{ij} are already available from the data structure of the distributed sparse linear system. One preconditioning step consists in solving the global system Eq. (5.3) approximately, and computing the u_i variables from the local equations as

$$u_i = B_i^{-1}[f_i - F_i y_i]. \tag{5.4}$$

This operation requires only a local solve. In our study we solve the global system Eq. (5.3) approximately by performing a few steps of GMRES preconditioned by block Jacobi, 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 (5.4).

In the next section, we report on numerical experiments to illustrate the performance of the VBARMS code for solving realistic applications.

5.3 Parallel results with PVBARMS

5.3.1 Performance of VBARMS in a parallel package

We applied the VBARMS method on a set of sparse linear systems $Ax = b$ arising from different application areas. For each linear system, we give in Table 5.1 the size, application field, number of nonzero entries and the characteristics of the block ordering computed by the compression algorithm. We recall again that the column *b-size* shows the average block size of A after the compression, and the column *b-density* shows the ratio of the number of nonzero entries in A before and after the compression. It is *b-density*=1 if the graph compression algorithm finds a perfect block structure in A with fully dense nonzero blocks, whereas *b-density*<1 means that some zero entries in the blocks are treated as nonzeros in VBARMS.

Table 5.1: Set and characteristics of test matrix problems.

Name	Size	Application	nnz(A)	b-size	b-density (%)
RAE	52995	Turbulence analysis	1748266	4.00	97
CT20STIF	52329	Engine block	2600295	2.61	100
RAEFSKY3	21200	Fluid structure interaction	1488768	8.00	100
VENKAT01	62424	2D Euler solver	1717792	4.00	100
BMW7ST	141347	Car body	7318399	4.63	100

We use the graph compression algorithm described in Chapter 2 that discovers perfect or imperfect block structures in the system by comparing the sparsity patterns of consecutive rows. In our experiments, we initially set the parameter $\tau = 1$ to find sets of rows and columns having the same pattern, and discover the presence of fully dense blocks in the matrix. We tested different values for τ, ranging from 0.7 to 1 on these two problems; with very little sacrifice in memory, it was possible to obtain larger blocks with still high density around 90%. The computed block ordering was used in our experiments with the VBARMS.

We compared the sequential and parallel VBARMS preconditioners with the original ARMS code [74] and the standard ILUT methods [67]. For the sequential version of ARMS and ILUT, we used the implementations available in the ITSOL package [52]; for the parallel version we used the pARMS package [52]. Prior to the iterative solution, we scaled the system by rows and columns so that the modulus of each entry of the scaled coefficient

matrix was smaller than one. By an abuse of notation we continue denoting by A the compressed matrix in the experiments with VBARMS. We used physical right-hand sides b when these were available, otherwise we set $b = Ae$ with $e = [1, \ldots, 1]^T$. For every run, we recorded the solution time from the start of the solve until either the initial residual was reduced by six orders of magnitude or no convergence was achieved after the prescribed maximum number of iterations of the flexible GMRES (FGMRES) method [66]. In this set of experiments, we restarted FGMRES every 20 inner iterations on the small problems (Table 5.2- 5.3), and every 100 inner iterations on the larger problems (Table 5.5). One important parameter to tune in VBARMS is the dropping threshold t. Just like we described in Chapter 4 small blocks $B_{ij} \in \mathbb{R}^{m_i \times n_j}$ are dropped in the incomplete factors $\bar{L}_\ell$, $\bar{U}_\ell$, $\bar{L}_S$, $\bar{U}_S$, $E_\ell \bar{U}_\ell^{-1}$, $\bar{L}_\ell^{-1} F_l$ and in the last level Scur complement matrix $A_{\ell_{max}}$ whenever $\frac{\|B\|_F}{m_i \cdot n_j} < t$.

The parameter settings are similar to the one in Chapter 4. For each matrix problem, we tested different values for the dropping parameter t in VBARMS, starting from $t = 0.1$ and decrementing it by a factor of 10 in each run; we selected the value of t which gave the best convergence result for the given problem. Finally, the number of levels of recursive factorization in VBARMS and ARMS were calculated automatically by the two codes, which stop when the Schur complement becomes too small to continue reducing the matrix. The maximum allowed size for the last level Schur complement matrix was set to 300. This value also determines the minimum size of the independent sets in the greedy algorithm.

The notations are the same, the time cost for computing the factorization (column "P-T") and for solving the linear system (column "I-T"), the ratio of the total number of nonzeros in the factors to the number of nonzeros in the coefficient matrix A (column "M-cost"), and the number of FGMRES iterations (column "Its").

First we conducted the experiments on small matrices (Table 5.2- 5.3) on a desktop computer. In Table 5.2 we report comparative results from our sequential runs. The results highlight the robustness of the VBARMS preconditioner. This is probably due to the better control of near-singularities of block ILU solvers, and to the better conditioning of the Schur complement matrices that are smaller and easier to invert. In our experiments on the small problems, we observed that the triangular factors computed by VBARMS were well conditioned; consequently, the triangular solves were

numerically stable.

Table 5.2: Performance comparison of VBARMS versus ARMS on one processor.

Matrix	Method	P-T	I-T	Total	Its	M-cost
RAE	VBARMS	4.51	0.62	5.13	15	4.62
	ARMS	68.95	>73.36	142.30	>1000	29.26
	ILUT	132.06	>106.12	238.18	>1000	49.99
CT20STIF	VBARMS	1.51	1.68	3.19	39	2.42
	ARMS	18.63	>40.81	59.44	>1000	8.27
	ILUT	90.60	>49.13	139.73	>1000	11.86
RAEFSKY3	VBARMS	0.77	0.04	0.81	3	2.00
	ARMS	5.07	0.05	5.12	3	4.01
	ILUT	1.81	0.06	1.87	6	2.39
VENKAT01	VBARMS	1.74	0.21	1.96	5	2.56
	ARMS	0.72	0.16	0.88	6	2.32
	ILUT	1.18	0.09	1.27	4	4.18
BMW7ST	VBARMS	6.54	0.23	6.77	2	3.67
	ARMS	22.65	>73.44	96.10	>1000	3.73
	ILUT	48.13	>103.97	152.10	>1000	8.37

In Table 5.3 we show the parallel performance of VBARMS, also against parallel ARMS and ILUT on the same problems. In these experiments, we compare the block Jacobi preconditioner (denoted as BJ) with VBARMS, ARMS and ILUT as local solvers. If we focus on the three subrows, we can see that VBARMS can be more efficient and numerically more stable than ARMS and ILUT in the parallel preconditioner on the same matrices. The conclusion would be the same; combined with the parallel preconditioner, VBARMS normally outperforms ARMS on solving the block structured matrices. The results also show good scalability on a modern desktop with multicore CPU.

In the one-level Schur complement method (denoted as Schur), we use VBARMS as a local solver and a fews steps of inner GMRES iterations for solving the global Schur complement system; precisely, the inner iterations are stopped after 100 steps or when the norm of the relative residual is decreased by two orders of magnitude. We refer the reader to Section 5.2 for a description of these preconditioners.

Table 5.3: Performance analysis of parallel VBARMS on 8 processors.

Matrix	Method	P-T	I-T	Total	Its	M-cost
RAE	SCHUR+VBARMS	1.56	98.00	99.56	714	7.19
	RAS+VBARMS	1.38	1.84	3.22	117	3.50
	BJ+VBARMS	1.25	3.79	5.03	251	3.46
	BJ+ARMS	2.43	>22.14	24.57	>1000	9.81
	BJ+ILUT	3.30	>27.85	31.15	>1000	13.36
CT20STIF	SCHUR+VBARMS	0.38	1.82	2.20	40	2.59
	RAS+VBARMS	0.29	1.07	1.36	38	2.04
	BJ+VBARMS	0.20	0.62	0.82	37	1.96
	BJ+ARMS	0.80	>28.59	29.39	>1000	6.93
	BJ+ILUT	0.54	>18.73	19.26	>1000	3.70
RAEFSKY3	SCHUR+VBARMS	0.21	0.03	0.23	4	1.85
	RAS+VBARMS	0.15	0.05	0.20	3	1.64
	BJ+VBARMS	0.09	0.04	0.13	3	1.70
	BJ+ARMS	0.13	0.24	0.37	6	2.45
	BJ+ILUT	0.08	0.32	0.40	8	1.80
VENKAT01	SCHUR+VBARMS	0.40	2.60	3.01	131	2.64
	RAS+VBARMS	0.33	0.21	0.54	8	2.42
	BJ+VBARMS	0.29	0.30	0.59	13	2.43
	BJ+ARMS	0.45	2.70	3.15	14	10.78
	BJ+ILUT	0.20	0.24	0.44	13	3.96
BMW7ST	SCHUR+VBARMS	12.18	14.09	26.27	58	4.00
	RAS+VBARMS	1.04	0.79	1.83	9	2.42
	BJ+VBARMS	0.90	0.51	1.41	5	2.63
	BJ+ARMS	3.95	>57.47	61.41	>1000	4.34
	BJ+ILUT	24.12	>88.36	112.48	>1000	9.75

In our experiments, Table 5.3 and 5.5 also show the comparison between the three parallel preconditioners; block Jacobi, restricted additive Schwarz and Schur complement method. As we can see, block Jacobi and restricted additive Schwarz are more robust than the one-level Schur complement-based preconditioner.

The results reported in Table 5.5 on larger problems confirm this trend. Table 5.4 shows the characteristics of these new larger matrices. For each linear system, we give in Table 5.1 the size, application field, number of nonzero entries and the characteristics of the block ordering computed by the compression algorithm.

Table 5.4: Set and characteristics of test matrix problems.

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

Table 5.5: Performance comparison of BJ + VBARMS and ARMS on larger matices.

Matrix	Method	P-T	I-T	Total	Its	M-cost
AUDIKW_1	BJ+VBARMS	84.23	308.18	392.42	331	3.46
	RAS+VBARMS	77.44	57.45	134.88	32	3.31
	SCHUR+VBARMS	126.39	2545.24	2671.63	63	5.51
	BJ+ARMS	114.43	>1785.02	1899.45	>3000	5.24
LDOOR	BJ+VBARMS	18.43	99.12	117.55	340	3.90
	RAS+VBARMS	23.75	23.64	47.39	47	4.26
	SCHUR+VBARMS	10.64	63.28	73.91	29	3.76
	BJ+ARMS	48.59	>1194.43	1243.01	>3000	7.66
STA004	BJ+VBARMS	19.14	81.14	100.27	92	3.88
	RAS+VBARMS	19.88	42.70	62.58	34	4.12
	SCHUR+VBARMS	8.17	446.88	455.05	256	1.80
	BJ+ARMS	9.36	65.92	75.28	145	2.87
STA008	BJ+VBARMS	44.82	195.89	240.71	256	5.27
	RAS+VBARMS	56.40	108.23	164.63	98	5.52
	SCHUR+VBARMS	824.44	3643.82	4468.26	862	2.06
	BJ+ARMS	151.64	>7740.94	7892.57	>3000	11.83

Scalability study

We also conducted several experiments to study the parallel scalability of our VBARMS method. Table 5.6 shows more details. We fix the matrix AUDIKW_1 and double the processor number repeatedly from 8 to 256, and analyze the time cost and iteration steps. First let us focus on the strong parallel scalability. As we can see in the table, the "P-T" column, the time cost is halved as the processor number doubles, which confirms a strong parallel scalability for both BJ and RAS. Numerical scalability is also very

good, since the "Its" column also shows that the iteration steps increases slowly as the processor number grows.

Compression	Method	P-N	P-T	I-T	Total	Its	M-cost
$\tau = 0.80$, *b-density*=96.40%, *b-size*=3.16.	BJ+VBARMS	8	86.71	169.96	256.66	116	3.55
		16	44.34	85.75	129.91	131	3.50
		32	19.44	98.02	117.46	279	3.29
		64	7.35	32.44	39.79	208	3.08
		128	2.22	18.67	20.89	223	2.88
		256	1.31	49.07	50.37	725	2.72
$\tau = 0.80$, *b-density*=96.40%, *b-size*=3.16.	RAS+VBARMS	8	104.64	69.76	174.41	28	3.39
		16	52.19	39.44	91.64	35	3.38
		32	27.92	19.79	47.71	39	3.16
		64	11.47	21.30	32.76	59	3.08
		128	5.82	13.87	19.69	78	2.93
		256	3.82	8.65	12.47	90	2.77

Table 5.6: Numerical and parallel scalability experiments on the AUDIKW_1 problem.

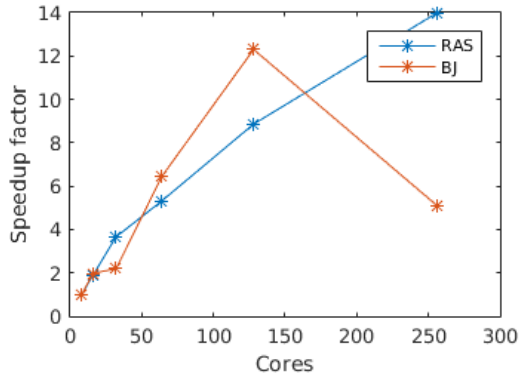


Figure 5.11: Speedup achieved on Millipede from University of Groningen, plot for BJ and RAS methods

Figure 5.11 also shows the plot of the scalability. Restricted Additive Schwarz achieved very good parallel and numerical scalability due to inherent parallelism and good convergence. On the other hand, the scalability of Block Jabobi suffers from the bad convergence on higher processor number.

Global and local graph compression strategy

Let us recall the implementation framework we introduced in Section 5.2, in order to interface it with VBARMS. Recall that the graph partition strategy we adopt consist of distributing the quotient graph and assigning the data to different processors. From a matrix viewpoint, block rows are assigned to different processors instead of point rows. Figure 5.12 and 5.13 illustrate this process. We take the input matrix A and convert it to a block matrix $\widetilde{A}$ and assign the block rows. The advantage of this implementation is that the block structure of the matrix will not be destroyed by the data distribution, so each processor will hold a well block-structured rectangular matrix. Since we perform the block matrix conversion on the global input matrix, we denote this implementation strategy as global graph compression.

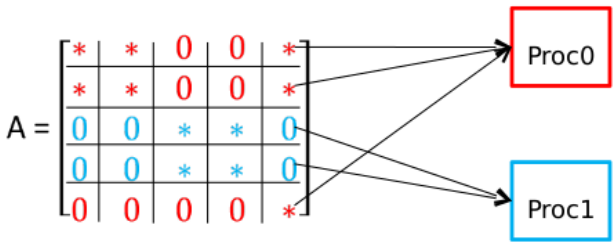


Figure 5.12: point rows distribution.

Besides this one, we also implemented a local graph compression approach. The process is as follows: we distribute the point-wise global matrix to processors, then each processor will hold a point-wise rectangular matrix. After that we generate the local square matrix corresponding to the local solver by calling parallel preconditioner. The last step is to convert the local square matrix into a block matrix and solve it via VBARMS. So the block matrix convertion is done on the local square matrix instead of the global matrix. Table 5.7 and 5.8 show the performance comparison between the two implementation strategies.

As we can see from Table 5.7 and 5.8, there are a few cases where the local graph compression strategy works better in terms of total time cost,

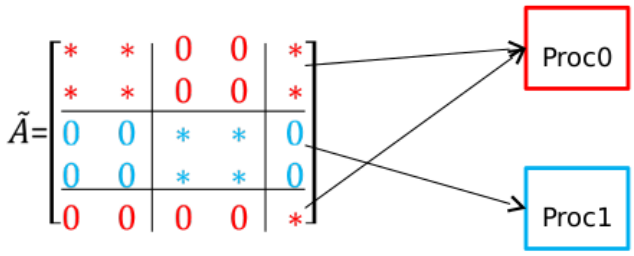


Figure 5.13: block rows distribution.

Table 5.7: Performance comparison of global and local graph compression.

Matrix	Method	C-Type	P-T	I-T	Total	Its	M-cost
RAE	BJ+VBARMS	global	1.25	3.79	5.03	251	3.46
		local	4.09	>33.05	37.14	>1000	3.50
	RAS+VBARMS	global	1.38	1.84	3.22	117	3.50
		local	4.26	18.27	22.53	480	3.55
CT20STIF	BJ+VBARMS	global	0.20	0.62	0.82	37	1.96
		local	0.31	0.56	0.87	37	2.06
	RAS+VBARMS	global	0.29	1.07	1.36	38	2.04
		local	0.46	1.17	1.63	57	2.10
RAEFSKY3	BJ+VBARMS	global	0.09	0.04	0.13	3	1.70
		local	0.06	0.10	0.16	6	1.72
	RAS+VBARMS	global	0.15	0.05	0.20	3	1.64
		local	0.13	0.16	0.29	3	1.59
VENKAT01	BJ+VBARMS	global	0.29	0.30	0.59	13	2.43
		local	0.29	0.26	0.55	14	2.45
	RAS+VBARMS	global	0.33	0.21	0.54	8	2.42
		local	0.32	0.27	0.59	9	2.44
BMW7ST	BJ+VBARMS	global	0.90	0.51	1.41	5	2.63
		local	0.99	0.77	1.76	10	2.28
	RAS+VBARMS	global	1.04	0.79	1.83	9	2.42
		local	1.14	0.90	2.04	11	2.30

memory cost and iteration steps, but the global approach is generally more effective. Besides, the local graph compression has a big drawback, that is its performance is sometimes unpredictable. For instance, on the RAE matrix in Table 5.7, the local graph compression strategy can not even converge.

Table 5.8: Performance comparison of global and local graph compression.

Matrix	Method	C-Type	P-T	I-T	Total	Its	M-cost
AUDIKW_1	BJ+VBARMS	global	84.23	308.18	392.42	331	3.46
		local	70.52	141.07	211.59	203	3.55
	RAS+VBARMS	global	77.44	57.45	134.88	32	3.31
		local	78.95	43.86	122.81	34	3.31
LDOOR	BJ+VBARMS	global	18.43	99.12	117.55	340	3.90
		local	20.31	65.41	85.71	175	3.84
	RAS+VBARMS	global	23.75	23.64	47.39	47	4.26
		local	20.33	14.39	34.72	38	3.97
STA004	BJ+VBARMS	global	19.14	81.14	100.27	92	3.88
		local	10.64	61.59	72.24	90	3.89
	RAS+VBARMS	global	19.88	42.70	62.58	34	4.12
		local	14.17	40.59	54.76	46	4.07
STA008	BJ+VBARMS	global	44.82	195.89	240.71	256	5.27
		local	15.48	176.40	191.88	246	5.35
	RAS+VBARMS	global	56.40	108.23	164.63	98	5.52
		local	19.81	131.79	151.59	125	5.35

The reason is that the local graph compression may destroy the original block structure of the matrix since it distributes a point-wise matrix. When this occurs, the local block solvers will not be able to perform well.

5.3.2 A Zoltan-based graph partitioning strategy and experiments

We report on numerical experiments to illustrate the performance of the parallel VBARMS code on a set of selected matrix problem of larger size. The set and the characteristics of the test matrix problems considered in this section are shown in Table 5.4. The parallel experiments were run on 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). For our runs we used large-memory nodes with 32 cores/node and 1TB of memory.

We investigated two different graph partitioning strategies in a parallel context. The first one applies a sequential graph partitioner on the quo-

tient graph $G/\mathcal{B}$ [49] on one processor and then it assigns each resulting partition to a different processor. Once other processors receive the partition, they load their local data from the matrix file. This strategy is the graph partitioning approach also adopted in Trilinos [46], and is denoted as "serial" in the tables. Its advantage is the low memory cost, since only one processor processes the global data. But it does have its disadvantage, that is the partition distribution is time-consuming because it is a one to all communication process.

load A
compute block ordering and permute A
partition A by contiguous block rows
$A \rightarrow A_i$
$A_i \rightarrow \tilde{A}_i$
Zoltan refines partition
$\tilde{A}_i \rightarrow \tilde{A}_i^{new}$
next instruct

P_i

Zoltan-based graph partition strategy.

Notations: A is the global point-wise matrix.
$i = 1, 2, \cdots, n$.
P_i means the workflow on i-th processor.
A_i is the local point-wise matrix on i-th processor.
$\tilde{A}_i$ is the local block-wise matrix on i-th processor.
$\tilde{A}_i^{new}$ is the updated local block-wise matrix on i-th processor.

The second strategy applies a parallel partitioner available in the Zoltan package [16] to the distributed quotient graph over the available processors. This strategy is denoted as "parallel" in the tables. The computational steps of the process can be summarized as follows:

STEP 1 Every processor loads the global matrix A.

STEP 2 Compute the block ordering via Algorithm 2.2 and permute the pointwise matrix A.

STEP 3 Partitions the pointwise matrix A by contiguous block rows, and create a map based on this partition.

STEP 4 Based on the map, each processor delete the non-local data from memory, after that it holds A_i the rectangular pointwise local matrix.

STEP 5 Each processor performs the data structure conversion, that is converting the pointwise local matrix A_i to blockwise local matrix A_i, after that we obtain a distributed block-wise linear system.

STEP 6 The distributed quotient graph can be extracted from this distributed block-wise linear system, and it can be passed to Zoltan routines to optimize; optimization means the new distributed graph has less couplings between processors.

STEP 7 Based on this new distributed graph, we perform the global data exchange, from each processor's viewpoint, it get a import and export list which tells which unknown should be received from which processor and which unknown on this processor should be sent to which processor. And it sends and receives data according the list. That is $\tilde{A}_i \rightarrow \tilde{A}_i^{new}$ step

At the end we will get a refined distributed system. The advantage of this Zoltan-based graph partition strategy is that it performs all to all communication, so it is supposed to be more time-saving. Our later experiments will show more details on this.

The build-in parallel hyper-graph partitioning in Zoltan is used because of better performance. The results reported in Tables 5.9-5.10 show that the parallel partitioning approach scales well with respect to the number of unknowns of the linear system. The two tables also highlight the performance of the two graph partition approaches, we could notice that the Zoltan-based graph partitioning strategy performs slightly better in terms of the convergence in most cases; also the time cost is much lower than the serial graph partitioning strategy since the step $A_i \rightarrow \tilde{A}_i$ in the diagram requires all to all data communication which is more efficient than one to all communication.

Finally, in Table 5.11 we illustrate the scalability results obtained with our parallel implementation of the Block Jacobi, Restricted Additive Schwarz, and Schur-complement preconditioners. This table can be considered as a extension of Table 5.6 because we also included the time cost of the graph compression strategy and the with the Schur-complement method. We observed that the Schur-complement preconditioner does not scale as well as the Block Jacobi and the Restricted Additive Schwarz methods because its solving phase needs to solve the global communication system. Another observation from this table is that the time cost of graph partitioning does scale very well.

Matrix	Method	G-Type	G-time (s)	Total (s)	Its	M-cost
AUDIKW_1	BJ+VBARMS	serial	54.5	70.23	136	3.13
		parallel	5.2	55.26	117	2.74
	RAS+VBARMS	serial	54.2	46.22	46	2.93
		parallel	5.3	44.99	52	2.87
	SCHUR+VBARMS	serial	54.4	377.83	69	6.21
		parallel	5.3	493.15	59	4.60
LDOOR	BJ+VBARMS	serial	30.0	26.40	345	1.95
		parallel	1.1	19.12	273	1.95
	RAS+VBARMS	serial	29.0	14.95	200	2.00
		parallel	1.1	13.85	196	1.99
	SCHUR+VBARMS	serial	29.0	22.56	54	3.63
		parallel	1.1	10.42	37	3.32
STA004	BJ+VBARMS	serial	79.4	50.08	90	3.61
		parallel	2.5	29.23	72	3.61
	RAS+VBARMS	serial	81.7	43.82	42	3.85
		parallel	2.6	30.99	34	3.31
	SCHUR+VBARMS	serial	81.4	153.04	90	5.29
		parallel	2.5	129.28	88	5.40
STA008	BJ+VBARMS	serial	81.9	97.14	227	4.77
		parallel	2.3	59.62	170	4.78
	RAS+VBARMS	serial	81.8	82.99	101	5.10
		parallel	2.4	59.42	97	5.07
	SCHUR+VBARMS	serial	81.2	620.94	188	8.94
		parallel	2.4	556.67	201	9.83

Table 5.9: Performance comparison of serial and parallel graph partition on 16 processors. Notation: P-N means number of processors, G-Type means graph partitioning strategy, G-time means partitioning timing cost, Total (s) means the time cost of preconditioning construction time cost plus iterative solution time cost, M-cost means memory costs.

Matrix	Method	G-Type	G-time (s)	Total (s)	Its	M-cost
AUDIKW_1	BJ+VBARMS	serial	55.7	67.47	285	2.89
		parallel	4.0	42.55	204	2.63
	RAS+VBARMS	serial	56.9	27.53	64	2.72
		parallel	4.1	22.65	52	2.87
	SCHUR+VBARMS	serial	54.6	219.69	82	5.73
		parallel	4.0	3153.95	>1000	5.84
LDOOR	BJ+VBARMS	serial	31.2	11.03	246	1.94
		parallel	1.1	11.91	260	1.97
	RAS+VBARMS	serial	30.6	8.90	190	1.98
		parallel	1.1	9.20	198	2.00
	SCHUR+VBARMS	serial	30.5	6.43	37	3.53
		parallel	1.2	7.69	54	3.31
STA004	BJ+VBARMS	serial	87.5	23.14	89	3.28
		parallel	2.0	20.53	73	3.25
	RAS+VBARMS	serial	82.6	25.38	39	3.46
		parallel	2.0	24.16	37	3.43
	SCHUR+VBARMS	serial	82.4	86.03	89	5.67
		parallel	2.2	64.68	90	5.10
STA008	BJ+VBARMS	serial	86.2	65.84	213	4.22
		parallel	2.1	49.50	217	4.20
	RAS+VBARMS	serial	82.5	58.06	118	4.46
		parallel	2.0	47.30	117	4.47
	SCHUR+VBARMS	serial	83.3	721.01	362	8.71
		parallel	2.1	360.30	293	8.21

Table 5.10: Performance comparison of serial and parallel graph partition on 32 processors.

Solver	P-N	G-T (s)	P-T (s)	I-T (s)	Total (s)	Its	M-cost
BJ	8	54.3	33.75	91.46	125.21	119	3.19
	16	54.6	18.88	51.35	70.23	136	3.13
	32	55.7	7.80	59.66	67.47	285	2.89
	64	56.7	2.75	19.78	22.52	219	2.69
	128	56.5	1.03	20.95	21.98	426	2.50
RAS	8	54.4	36.12	46.29	82.41	45	2.96
	16	54.2	19.54	26.68	46.22	46	2.93
	32	56.9	10.30	17.23	27.53	64	2.72
	64	55.4	4.49	12.45	16.94	83	2.63
	128	57.8	2.38	7.26	9.64	86	2.47
SCHUR	8	54.6	164.91	526.78	691.69	61	5.35
	16	54.4	82.72	295.11	377.83	69	6.21
	32	54.6	42.94	176.75	219.69	82	5.73
	64	56.0	18.73	959.26	977.99	-	5.42
	128	56.8	7.97	441.73	449.76	-	5.15

Table 5.11: Scalability study of serial graph partition on AUDIKW_1 matrix. The dash symbol in the table means that the FGMRES method did not converge after 1000 iterations. P-T is the time cost of preconditioner construction, I-T is the iterative solution time.

6 Case study in large-scale turbulent flows simulation

We finally get back to the starting point that has motivated this study on parallel block multilevel incomplete LU factorization methods. In this section we illustrate a performance analysis for solving large block structured linear systems arising from an implicit formulation of the Reynolds Averaged Navier Stokes equations (briefly, RANS), using preconditioned Newton-Krylov methods. Although explicit multigrid techniques have dominated the Computational Fluid Dynamics (CFD) area 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. Practical implicit CFD solvers, though, need to be combined with well-suited convergence acceleration techniques in order to be competitive with more conventional solvers in terms of CPU cost [87]. Critical feature is the choice of the preconditioning strategy for inverting the large nonsymmetric linear system at each step of the Newton's algorithm. This can have a strong impact on the computational efficiency especially when the mean flow and turbulence transport equations are solved in fully coupled form, like we do.

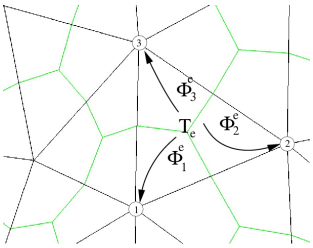
6.1 Definition of the problem

Throughout this section we use standard notation for the kinematic and thermodynamic variables: $\vec{u}$ is the flow velocity, ρ is 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.

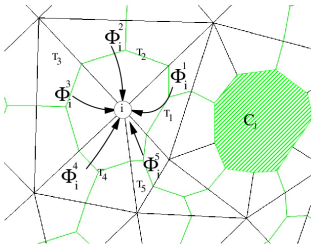
In the case of high Reynolds number flows, we account for turbulence

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 [79].

The mesh 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 6.1.



(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 6.1: 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

$$\int_{C_i} \frac{\partial \vec{q}_i}{\partial t}\, dV = \oint_{\partial C_i} \vec{n} \cdot F\, dS - \oint_{\partial C_i} \vec{n} \cdot G\, dS + \int_{C_i} \vec{s}\, dV, \tag{6.1}$$

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 (6.1), the operators F and G represent the inviscid

and viscous fluxes, respectively. For compressible flows, we have

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

and for incompressible, constant density flows,

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

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

$$c_{b1}\left[1 - f_{t2}\right]\tilde{S}\tilde{\nu} + \frac{1}{\sigma Re}\left[c_{b2}\left(\nabla\tilde{\nu}\right)^2\right] + -\frac{1}{Re}\left[c_{w1}f_w - \frac{c_{b1}}{\kappa^2}f_{t2}\right]\left[\frac{\tilde{\nu}}{d}\right]^2. \quad (6.2)$$

For a description of the various functions and constants involved in (6.2) we refer the reader to [79].

In the fluctuation splitting space discretization approach that we use, the integral form of the governing equations (6.1) is discretized over each control volume C_i evaluating the flux integral over each triangle (or tetrahedron) in the mesh, and then splitting it among its vertices [17] (see Figure 6.1). Therefore, we may write Eq. (6.1) as

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

where

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

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

$$M\frac{d\vec{q}}{dt} = \vec{r}(\vec{q}), \quad (6.3)$$

where t denotes the pseudo time variable, M is the mass matrix and $\vec{r}(\vec{q})$ represents the nodal residual vector of the 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 (6.3) is approximated using a two-point one-sided finite difference (FD) formula we obtain the following implicit scheme:

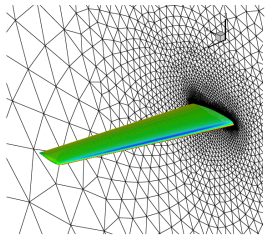
$$\left(\frac{1}{\Delta t^n}V - J\right)\left(\vec{q}^{n+1} - \vec{q}^{n}\right) = \vec{r}(\vec{q}^{n}), \tag{6.4}$$

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. (6.4) 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, *i.e.* entry (i, j) is nonzero if and only if entry (j, i) is nonzero as well; on average, the number of non-zero (block) entries per row in our discretization scheme equals 7 in 2D and 14 in 3D.

6.2 The results of the experiments

We present the turbulent incompressible flow analysis of a three-dimensional wing. The geometry, illustrated in Figure 6.2, was proposed in the 3rd AIAA Drag Prediction Workshop [88]; we refer to this geometry as the "DPW3 Wing-1". Flow conditions are 0.5° angle of attack and Reynolds number based on the reference chord equal to $5 \cdot 10^6$. The freestream turbulent viscosity is set to ten percent of its laminar value.

In Table 6.1 we show experiments with parallel VBARMS on the four meshes of the DPW3 Wing-1 problem, and in Table 6.2 we also report on comparative results against other popular solvers. Finally in Table 6.3-6.4 we perform a scalability study on the Mesh2 case using both parallel and sequential graph partitioning. The results of our experiments confirm the trend of performance shown on general problems, and seem to indicate that



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 : $n = 4918165$ $nnz = 318,370,485$
RANS2 : $n = 4918165$ $nnz = 318,370,485$
RANS3 : $n = 9032110$ $nnz = 670,075,950$
RANS4 : $n = 12085410$ $nnz = 893,964,000$

Figure 6.2: Geometry and mesh characteristics of the DPW3 Wing-1 problem. Problems RANS1 and RANS2 correspond to the same mesh.

the proposed parallel method is efficient and useful for solving large-scale problems in different application areas.

Matrix	Method	G-Type	G-time (s)	Total (s)	Its	M-cost
RANS1 n = 4918165	BJ+VBARMS	serial	501.0	60.70	49	2.89
		parallel	17.3	50.13	34	2.98
	RAS+VBARMS	serial	498.4	51.09	16	2.97
		parallel	17.4	52.37	19	3.06
	SCHUR+VBARMS	serial	501.8	102.68	43	2.68
		parallel	17.6	67.93	35	2.57
RANS2 n = 4918165	BJ+VBARMS	serial	501.2	112.04	51	4.02
		parallel	17.0	86.86	47	4.35
	RAS+VBARMS	serial	499.0	120.2	39	4.23
		parallel	16.8	101.89	39	4.49
	SCHUR+VBARMS	serial	497.6	1294.60	143	7.01
		parallel	17.5	342.39	24	6.47
RANS3 n = 9032110	BJ+VBARMS	serial	2480.3	204.24	180	4.03
		parallel	27.2	287.36	154	4.40
	RAS+VBARMS	serial	2523.2	280.39	119	4.20
		parallel	25.2	209.79	71	4.48
	SCHUR+VBARMS	serial	2440.3	728.63	131	4.11
		parallel	22.0	774.31	140	4.39
RANS4 n = 12085410	BJ+VBARMS	serial	632.6	145.27	335	3.75
		parallel	51.5	117.94	223	3.91
	RAS+VBARMS	serial	637.4	124.46	200	3.99
		parallel	43.9	105.58	143	4.12
	SCHUR+VBARMS	serial	610.2	342.39	161	3.79
		parallel	39.3	305.03	179	3.76

Table 6.1: Experiments on the DPW3 Wing-1 problem. The RANS1, RANS2 and RANS3 test cases are solved on 32 processors, whereas the RANS4 problem on 128 processors.

Matrix	G-Type	Method	Total (s)	Its	M-cost
		BJ+VBARMS	204.24	180	4.03
RANS3	serial	BJ+VBILUT	1575.32	729	7.34
		BJ+ARMS	-	-	6.63
n = 9032110	parallel	BJ+VBARMS	287.36	154	4.40
		BJ+VBILUT	9018.27	979	13.81
		BJ+VBARMS	145.27	335	3.75
RANS4	serial	BJ+VBILUT	261.16	494	4.56
		BJ+ARMS	-	-	5.38
n = 12085410	parallel	BJ+VBARMS	117.94	223	3.91
		BJ+VBILUT	296.35	472	5.26

Table 6.2: 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.

Solver	P-N	G-T (s)	Total-T (s)	Its	M-cost
	8	39.3	421.18	39	5.46
	16	28.0	202.10	47	4.95
BJ	32	16.6	86.89	47	4.35
	64	14.2	44.17	58	3.65
	128	17.3	21.55	69	3.21
	8	38.9	388.37	27	5.70
	16	28.0	219.48	35	5.22
RAS	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 6.3: Scalability study on the RANS2 problem using parallel graph partitioning.

Solver	P-N	G-T (s)	Total-T (s)	Its	M-cost
BJ	8	494.0	337.34	53	4.74
	16	493.0	211.88	48	4.63
	32	501.7	111.78	51	4.02
	64	500.2	48.63	68	3.49
	128	506.6	26.86	86	3.11
RAS	8	495.4	310.06	29	4.71
	16	495.1	230.03	29	5.02
	32	500.2	120.58	39	4.23
	64	507.7	57.75	45	3.78
	128	502.5	30.58	54	3.35

Table 6.4: Scalability study on the RANS2 problem using sequential graph partitioning.

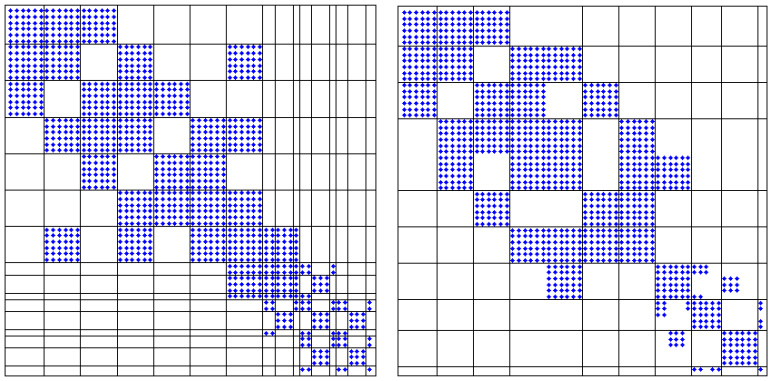
7 Concluding remarks

7.1 Conclusions

Sparse matrices arising from the discretization of partial differential equations often preserve a perfect block structure when several variables are associated with the same grid point. Meanwhile, on today's modern computer platform, the data movement inside the memory hierarchy is a crucial factor which determines the performance of numerical algorithms. For sparse matrix codes, highly-tuned data structures may be useful to exploit the sparsity of the matrix and minimize the data movement and achieve high performance on the hardware of modern computer systems. Given the above reasons, developing an efficient block solver to take advantage of the special structure is an interesting research topic.

In Chapter 2, we recalled the angle-based graph compression method [69] which is able to build an imperfect block structure upon the matrix. An imperfect block structure is also called an approximate dense structure; for each nonzero block, it contains a lot of nonzero entries and only a few zeros. Since we store the whole nonzero block, the zero elements in this block are also stored, they are treated as nonzero elements. This approach sacrifices a little memory, but it can enlarge the block size and improve the performance of the block solver. Figure 7.1 shows an example of imperfect blocking.

However, there is an issue of this angle-based graph compression method. For some matrices, the *b-density* (the ratio of the number of nonzero entries in the matrix before and after the compression) may be sensitive to τ (τ is the input parameter, also the angle value to measure the pattern similarity of two rows) as we showed in Table 2.3. For example, for the matrix VENKAT01, the τ value shifts from 0.57 to 0.58 results in the matrix *b-density* changes from 29.71% to 86.37%. In order to avoid this issue, we proposed a new graph-based compression method, which is based on merging two small blocks into one bigger block and it provides a more user-friendly parameter μ (μ is the lower bound of *b-density* you can accept). We presented a comparison between the two methods in Section 4.3.3, the



Perfect blocking. Imperfect blocking.

Figure 7.1: An example of perfect and imperfect blocking.

results showed similar performance, but the graph-based method is more user-friendly.

Chapter 3 recalls Krylov subspace methods and preconditioning techniques [70] which are the preliminary of our new solver. A few classic multilevel ILU-based preconditioners were also recalled. ARMS (Algebraic Recursive Multilevel Solver) [74] is one of them. In Chapter 4, we combine ARMS with the graph compression methods introduced in Chapter 2 and designed the new solver VBARMS (Variable Block Algebraic Recursive Multilevel Solver). We also presented the exact steps of VBARMS in this chapter, here we use the graph compression method to build a block structure (either perfect or imperfect block structure). Then we perform the multilevel LU factorization on blocks. From the implementation perspective, we convert the matrix data structure CSR (listing 4.1) format to VBCSR (listing 4.2) format, such that higher level BLAS routines can be used. Due to the design of VBARMS and the results we presented, it has a few advantages with respect to ARMS.

1. The VBCSR format uses much less column index storage with respect to CSR format, which results in less memory cost [69].

2. Performing calculations on blocks, decreases the chance of breakdown in the ILU process greatly, the diagonal blocks have much less chance

to be singular than diagonal entries to be zero. This means better stability [28].

3. For multilevel solvers, a smaller Schur complement always improves the performance. As VBARMS generates an independent set ordering based on a quotient graph, it is able to produce a smaller Schur complement than ARMS.

4. BLAS 3 and 2 routines are used for most of the operations. Table 4.1 showed that higher level BLAS routines are able to execute more operations with the same amount of data movement (Chapter 5 of [34]), which means better efficiency.

5. A computational unit is an array instead of a single entry. The array can be reused in the cache, which leads to less data movement inside the memory hierarchy. Hence the better cache reuse is better.

Furthermore, two implementations of calculating the Schur complement were presented. During the calculation, all the submatrices are stored in VBCSR format. One implementation is called explicit Schur complement calculation. It calculates LU factors and applies them to columns of the upper right matrix, which results in more intermediate data storage. The other one is called implicit Schur complement calculation. The implicit implementation contains two major loops which work on the whole row of the matrix. The first one generates LU by performing a variant of GE (Algorithm 4.5), during the loop it also applies L factors to the upper right matrix. The second loop uses the multipliers computed in the first loop to calculate an approximation of the Schur complement.

Since implicit Schur complement calculation has a well-designed and efficient loop, it is supposed to be more time-saving. Table 4.10 compares the performance of two implementations on a set of matrices. The results confirm this point, the time cost of the preconditioner setup for the implicit Schur complement calculation is much less. So it is a more mature implementation than explicit Schur complement calculation.

In Chapter 5, a parallelization of VBARMS based on a block version of Block Jacobi (BJ), the Restricted Additive Schwarz and (RAS) Schur-complement methods [54] for distributed memory computers is proposed. The results of numerical experiments were also presented in this chapter.

The trend continues also in the parallel implementation, VBARMS outperforms ARMS when solving block structured matrices. A scalability study also showed very good results of the BJ and RAS methods. Moreover, in Section 5.3.1, we proposed a new local graph compression strategy which converts the local diagonal matrix (in the distributed matrix context, each processor holds a local rectangular submatrix, the diagonal square matrix is the local diagonal matrix) into a block matrix and then calls VBARMS. It was shown that the local one works well for some cases but may diverge because of possible damage of the block structure during the point-wise matrix distribution.

Furthermore, in the context of distributed parallel computing, the graph partitioning strategy plays a very important role. So two strategies were proposed: one uses Zoltan library [16] to refine the distributed graph and we call it parallel graph partitioning, the other one partitions the serial graph on one processor and broadcasts to other processors. We call this one serial graph partitioning. The results of the experiments were reported in Tables 5.9-5.10, which highlight the performances of the two graph partition approaches. We could notice that the parallel graph partitioning strategy performs slightly better in most cases in terms of the time cost of the graph partitioning process. Table 5.11 exhibited good scalability for the two approaches in combination with BJ and RAS.

7.2 Perspectives

There are still a few topics for future research that should be addressed:

- The graph-based compression method introduced in Chapter 2 provides a more user friendly parameter, but is still not able to totally outperform the angle-based graph compression method. Therefore, refining the graph-based compression method to obtain better block structure and achieve better performance of the solver can be a promising research topic.

- Both VBARMS and pVBARMS are restricted to block structured matrices. So generalizing to unstructured matrices is a very attractive research direction. In Section 4.3.2, we tried to block only the Schur complement matrix which is much denser than the original matrix. However, this strategy works only when Schur complements and the

factors computed by ARMS are stable and accurate. According to the results, for the cases this strategy worked, the improvement is still small. That is because the time and memory cost of the first level factorization are a big portion of the time and memory cost of the whole process. VBARMS only improved on a small portion of the whole process. Improvement of this strategy can be investigated in the future.

- In Chapter 5, we proposed a block version of the three popular global preconditioners (block Jacobi, restricted additive Schwarz and Schur complement method). Block Jacobi and restricted additive Schwarz showed robust and stable performance and good scalability. But the performance of the Schur complement method is not as stable as we expected; it is also sensitive to scaling parameters. Refining the design of the Schur complement method and improving its performance can be an interesting research topic.

- pVBARMS is the parallel implementation of VBARMS in a distributed computing framework and it showed very impressive performance. Today GPU (graphics processing unit)-accelerated computing technology is getting more and more popular and mature in scientific computing. GPU-accelerated computing pioneered in 2007 by NVIDIA, is the use of a GPU together with a CPU to accelerate the calculations performed on data. Based on the pVBARMS framework, a GPU-implementation of VBARMS also is a promising research topic for future.

- VBARMS combines multilevel method with a block-wise solver. This combination reduces the multilevel method complexity effectively (see Table 4.5). So it is natural to apply this strategy to a multigrid method [80], and in this way improve the multigrid performance for block structured matrices.

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J.J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[2] W.E. Arnoldi. The principle of minimized iteration in the solution of the matrix eigen-problem. *Quart. Appl. Math*, 9:17–29, 1951.

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

[4] O. Axelsson. A generalized SSOR method. *BIT*, 12:443–467, 1972.

[5] O. Axelsson. *Iterative solution methods*. Cambridge University Press, Cambridge, 1994.

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

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

[8] Randolph E. Bank and Christian Wagner. Multilevel ILU decomposition. *Numerische Mathematik*, 82(4):543–576, 1999.

[9] Blaise Barney. Introduction to parallel computing. URL:`https://computing.llnl.gov/tutorials/parallel_comp/`.

[10] Blaise Barney. Openmp. URL:`https://computing.llnl.gov/tutorials/openMP/`.

[11] Blaise Barney. Posix threads programming. URL:`https://computing.llnl.gov/tutorials/pthreads/`.

[12] M. Benzi. Preconditioning techniques for large linear systems: A survey. *J. Comp. Phys.*, 182:418–477, 2002.

[13] M. Benzi, R. Kouhia, and M. Tůma. Stabilized and block approximate inverse preconditioners for problems in solid and structural mechanics. *Computer Methods in Applied Mechanics and Engineering*, 190(4950):6533 – 6554, 2001.

[14] M. Bollhöfer. A robust and efficient ILU that incorporates the growth of the inverse triangular factors. *SIAM J. Sci. Comput.*, 25(1):86–103, 2003.

[15] M. Bollhöfer and Y. Saad. Multilevel preconditioners constructed from inverse–based ILUs. *SIAM J. Scientific Computing*, 27(5):1627–1650, 2006.

[16] 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.

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

[18] 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.

[19] Xiao-Chuan Cai and Marcus Sarkis. A restricted additive schwarz preconditioner for general sparse linear systems. *SIAM J. Sci. Comput*, 21:792–797, 1999.

[20] B. Carpentieri, J. Liao, and M. Sosonkina. Variable block multilevel iterative solution of general sparse linear systems. In *Proceedings of the 10th International Conference on Parallel Processing and Applied Mathematics PPAM, Warsaw, Poland*, 2013.

[21] B. Carpentieri, J. Liao, and M. Sosonkina. *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, chapter Variable block algebraic

recursive multilevel solver (VBARMS) for sparse linear systems, pages 163 – 172. M. Bader, A. Bode, H.-J. Bungartz, M. Gerndt, G.R. Joubert, F. Peters (eds.), IOS Press, 2014.

[22] 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.

[23] B. Carpentieri, J. Liao, M. Sosonkina, and A. Bonfiglioli. A parallel multilevel incomplete lu factorization preconditioner that exploits block matrix structures. In *Proceedings of the 27th International Conference on Parallel Computational Fluid Dynamics ParCFD, Montreal, Canada*, 2015.

[24] Bruno Carpentieri, Sven Baars, Jia Liao, Masha Sosonkina, and A. Bonfiglioli. A parallel multilevel ILU factorization preconditioner that exploits block matrix structures. *Parallel Computing*, 2015. under review. See also: Sven Baars. Block and Conquer: Exploiting block structures to improve the performance of multilevel incomplete factorisation preconditioning. M.Sc. thesis, University of Groningen, 2014.

[25] A. Chapman, Y. Saad, and L. Wigton. High-order ILU preconditioners for CFD problems. *Int. J. Numer. Methods Fluids*, 33(6):767–788, 2000.

[26] E. Chow. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM J. Scientific Computing*, 21(5):1804–1822, 2000.

[27] E. Chow and Y. Saad. Approximate inverse techniques for block-partitioned matrices. *SIAM J. Scientific Computing*, 18:1657–1675, 1997.

[28] E. Chow and Y. Saad. Experimental study of ILU preconditioners for indefinite matrices. *Journal of Computational and Applied Mathematics*, 86:387–414, 1997.

[29] P. Concus, G.H. Golub, and G. Meurant. Block preconditioning for the conjugate gradient method. *SIAM J. Scientific and Statistical Computing*, 6:220–252, 1985.

[30] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. 24th National Conference of the Association for Computing Machinery, Brandon Press, New Jersey*, pages 157–172. Brandon Press, New Jersey, 1969.

[31] J.W. Demmel. Block *lu* factorization. *NLAA Numerical Linear Algebra with Applications*, 2:173–190, 1995.

[32] A. Frank Van der stappen, Rob H. Bisseling, and Johannes G.G. Van de vorst. Pararllel sparse LU decomposition on a mesh network of transputers*. *SIAM J. Matrix Analysis and Applications*, 14:853–879, 1993.

[33] 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.

[34] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst. *Numerical linear algebra for high-performance computers*, volume 7 of *Software, Environments, and Tools*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1998.

[35] I.S. Duff. Combining direct and iterative methods for the solution of large systems in different application areas. Technical Report TR/-PA/04/128, CERFACS, Toulouse, France, 2004. Also RAL Technical Reports, RAL-TR-2004-033.

[36] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, 1986.

[37] I.S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Analysis and Applications*, 20(4):889–901, 1999.

[38] Mark Embree. The tortoise and the hare restart GMRES. *SIAM Review*, 45:259–266, 2003.

[39] Andreas Frommer and Daniel B. Szyld. An algebraic convergence theory for restricted additive schwarz methods using weighted max norms. *SIAM J. Num. Anal*, 7:856–869, 1986.

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

[41] J. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973.

[42] G.H. Golub and C.F. Van Loan. *Matrix computations*. Johns Hopkins Studies in the Mathematical Sciences. The Johns Hopkins University Press, Baltimore, MD, USA, third edition, 1996.

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

[44] John L. Hennesy and David A. Patterson. *Computer Architecture*. EIservier, fifth edition, 2012.

[45] V. Henson and U. Yang. BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Appl. Numer. Math.*, 41:155–177, 2002.

[46] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. Trilinos home page. `http://trilinos.org/`, 2014.

[47] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand.*, 49:409–436, 1952.

[48] I.C.F. Ipsen and C.D. Meyer. The idea behind Krylov methods. *The American Mathematical Monthly*, 105(10):889–899, 1998.

[49] 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.

[50] M. Knepley and J. Brown. PETSc home page. `http://www.mcs.anl.gov/petsc/`, 2014.

[51] R. Leuze. Independent set orderings for parallel matrix factorizations by gaussian elimination. *Parallel Computing*, 10:177–191, 1989.

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

[53] X.S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.

[54] 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.

[55] J. Liao and B. Carpentieri. A variable block variant of the algebraic recursive multilevel solver (vbarms) for general linear systems. In *Proceedings of the 3rd IMA Conference on Numerical Linear Algebra and Optimisation, University of Birmingham, UK*, 2012.

[56] Jia Liao, Bruno Carpentieri, and Masha Sosonkina. A varible block variant of the algebraic recursive multilevel solver (vbarms), 2012. The 38th Woudschoten Conference.

[57] C.D. Meyer M. Benzi and M. Tuma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM J. Sci. Comput*, 17:11351149, 1996.

[58] M. Manguoglu. A domain-decomposing parallel sparse linear system solver. *Journal of Computational and Applied Mathematics*, 236(3):319 – 325, 2011.

[59] R.M.M. Mattheij. Stability of block lu-decomposition of matrices arising from bvp. *SIAM Industrial and Applied Mathematics*, 5:314–331, 1984.

[60] J. A. Meijerink and H.A. van der Vorst. An iterative solution method for linear systems of which the coeffcient matrix is a symmetric M-matrix. *Mathematics of Computation*, 31:148–162, 1977.

[61] G.A. Meurant. *Computer Solution of Large Linear Systems*, volume 28 of *Studies in Mathematics and Its Applications*. North-Holland, Amsterdam, 1999.

[62] M. Magolu monga Made, R. Beauwens, and G. Warzee. Preconditioning of discrete Helmholtz operators perturbed by a diagonal complex matrix. *Communications in Numerical Methods in Engineering*, 11:801–817, 2000.

[63] M. Magolu monga Made and Ben Polman. Experimental comparison of three-dimensional point and line modified incomplete factorizations. *Numerical Algorithms*, 23(1):51–70, 2000.

[64] 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.

[65] J.W. Ruge and K. Stüben. *Multigrid Methods, Frontiers in Applied Mathematics 3*, chapter Algebraic multigrid (AMG), pages 73–130. S.F. McCormick ed., SIAM, Philadelphia, PA, 1987.

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

[67] Y. Saad. ILUT: A dual threshold incomplete LU factorization. *Numerical Linear Algebra with Applications*, 1:387–402, 1994.

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

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

[70] Y. Saad. *Iterative Methods for Sparse Linear Systems.* SIAM Publications, second edition, 2003.

[71] Y. Saad. Multilevel ILU with reorderings for diagonal dominance. *SIAM J. Scientific Computing*, 27(3):1032–1057, 2005.

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

[73] Y. Saad, A. Soulaimani, and R. Touihri. Variations on algebraic recursive multilevel solvers (ARMS) for the solution of CFD problems. *Applied Numerical Mathematics*, 51:305–327, 2004.

[74] 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.

[75] Y. Saad and J. Zhang. BILUM: block versions of multielimination and multilevel ILU preconditioner for general sparse linear systems. *SIAM J. Sci. Comput.*, 20:2103–2121, 1999.

[76] Y. Saad and J. Zhang. BILUTM: A domain-based multilevel block ILUT preconditioner for general sparse matrices. *SIAM J. Matrix Analysis and Applications*, 21(1):279–299, 1999.

[77] Youcef Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2, 1994.

[78] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Scientific and Statistical Computing*, 10:36–52, 1989.

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

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

[81] Herb Sutter. The free lunch is over. a fundamental turn toward concurrency in software. *C/C++ Users Journal*, 23, 2005.

[82] D. Terpstra, H. Jagode, H. You, and J.J. Dongarra. *Tools for High Performance Computing 2009*, chapter Collecting Performance Data with PAPI-C, pages 157–173. Springer Berlin / Heidelberg, 3rd Parallel Tools Workshop, Dresden, Germany, 2009.

[83] H.A. van der Vorst. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Scientific and Statistical Computing*, 13:631–644, 1992.

[84] H.A. van der Vorst and C. Vuik. The superlinear convergence behaviour of GMRES. *J. Comput. Appl. Math.*, 48:327–341, 1993.

[85] 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.

[86] R.S. Varga, E.B. Saff, and V. Mehrmann. Incomplete factorizations of matrices and connections with H-matrices. *SIAM J. Numerical Analysis*, 17:787–793, 1980.

[87] P. Wong and D.W. Zingg. Three-dimensional aerodynamic computations on unstructured grids using a Newton-Krylov approach. *Computers and fluids*, 37:107–120, 2008.

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

[89] M. Sosonkina Y. Saad. Distributed schur complement techniques for general sparse linear systems. *SIAM J. Scientific Computing*, 21(6):1337–1356, 1999.

Summary

In the first chapter, we address the objective for this thesis: Solving linear system is the most time-consuming part of the whole simulation. The linear systems we obtained often possess a block structure. Therefore, taking advantage of the block structure of these matrices, developing a novel powerful multi-level solver to solve these matrices effectively becomes necessary.

Let us focus on how to exploit the matrix block structure first. Note that there are two types of block structured matrices.

1. the constant block size matrix

2. the variable block size matrix

For the first type, there is already a mature implementation in Trilinos [46] and PETSc [50]; but the second type is still a subject for ongoing research. In Chapter 2, we recalled the classic checksum method which can detect the inherent variable block structure, also recalled the angle-based graph compression method which enables us to build an imperfect block structure. The experiments showed the user how to tune parameters to get the desired block structure. Furthermore, we introduced a new graph-based compression method which attempts to build an imperfect block structure based on merging two initial blocks into a bigger block.

So after Chapter 2, the first part of the challenge is resolved. Then we move to the second part: developing a block-wise multi-level solver. Before that, it is essential to recall the basics of solving large sparse linear system. So in Chapter 3, we recalled the popular Krylov subspace methods and preconditioning techniques which play a very important role in developing powerful solvers. Particularly, we also reviewed several ILU-based multilevel solvers which exhibit a very beneficial framework for the design of our new solver.

With all the preliminary introduction, the design of our block-wise multilevel solver comes naturally. We follow the ARMS (Algebraic Recursive

Multilevel Solver) framework, the novelty of our work is using graph compression algorithms to build the block structure, then we perform all the operations on blocks during the whole process of ARMS. In Chapter 4, we started with basic block operations for block structured matrices, and then move to detailed steps of VBARMS (Variable Block Algebraic Recursive Multilevel Solver). In our experiments, we use VABRMS as a preconditioner in a Krylov subspace method. Results of numerical experiments showed the effectiveness and stability of VBARMS on block structured matrices. Moreover, we also displayed the results of the comparison between the classic angle-based graph compression method and our new graph-based compression. The results confirm that the graph-based compression method has more user friendly parameter tuning with similar performance. The results of the two VBARMS implementation strategies (implementation of explicit Schur complement calculation and implementation of implicit Schur complement calculation) were also exhibited, Implementation of implicit Schur complement calculation showed better performance in terms of time cost.

Up to now, VBARMS showed very good performance on small test problems. This motivates us to reach the end of the challenge we set: solve the real application problems. That requires incorporating VBARMS into a parallel implementation. Therefore in Chapter 5, we recalled the basics of parallel computing, and two parallel block solvers were presented: one based on the additive Schwarz method and the other one based on the Schur complement method. The distributed parallel implementation of VBARMS (pVBARMS) were also introduced; pVBARMS also showed robust performance and good scalability. Furthermore, a new Zoltan-based graph partitioning strategy was introduced and its performance was presented.

At the end, for a specific application, large-scale turbulent Navier-Stokes equations, the derived matrices also possess a block structure. Here our experiments confirm the trend of performance shown on general problems.

Sparse matrices arising from the solution of systems of partial differential equations often exhibit a perfect block structure, meaning that the nonzero blocks in the sparsity pattern are fully dense (and typically small), e.g., when several unknown quantities are associated with the same grid point. However, similar block orderings can be sometimes unravelled also on general unstructured matrices, by ordering consecutively rows and columns with a similar sparsity pattern, and treating some zero entries of the reordered matrix as nonzero elements, and the nonzero blocks as dense. The reordering

results in linear systems with blocks of variable size in general.

Our recently developed parallel package pVBARMS (parallel variable block algebraic recursive multilevel solver) for distributed memory computers takes advantage of these frequently occurring structures in the design of the multilevel incomplete LU factorization preconditioner, and maximizes computational efficiency achieving increased throughput during the computation and improved reliability on realistic applications. The method detects automatically any existing block structure in the matrix without any users prior knowledge of the underlying problem, and exploits it to maximize computational efficiency.

Furthermore, in the context of distributed parallel computing, two graph partitioning strategies were proposed: one uses Zoltan library to refine the distributed graph, the other one partitions the serial graph on one processor and broadcasts to other processors. The two strategies combine with pVBARMS show very good performance on solving the unsteady, turbulent, Reynolds-averaged, Navier-Stokes equations.

Samenvatting

In het eerste hoofdstuk richten we ons op de doelstelling van dit proefschrift: het oplossen van lineaire systemen is het meest tijdrovende deel van de gehele simulatie. De lineaire systemen die wij verkregen hebben hebben vaak een blok structuur. Derhalve is het uitbuiten van de blokstructuur van deze matrices, het ontwikkelen van een nieuwe krachtige multilevel oplossers voor deze matrices daadwerkelijk nodig.

Laten we ons concentreren op hoe de matrix blokstructuur te exploiteren. Merk op dat er twee soorten blokgestructureerde matrices zijn.

1. De constante blokgrootte matrix

2. De variabele blokgrootte matrix

Voor het eerste type is er al een volwassen implementatie in Trilinos [46] en PETSc [50]; maar de tweede soort nog een onderwerp wat onderhevig is aan onderzoek. In Hoofdstuk 2, kijken we terug op de klassieke checksum methode die de inherente variabele blokstructuur kan detecteren, kijken we terug op de hoek gebaseerde graafcompressiemethode die ons in staat stelt om een onvolmaakte blok structuur op te bouwen. De experimenten toonden aan de gebruiker hoe de parameters afgestemd moeten worden om de gewenste blokstructuur te krijgen. Verder bekijken we een nieuwe compressiemethode gebaseerd op grafen die probeert een onvolmaakte blok structuur op te bouwen gebaseerd op het samenvoegen van twee blokken in een groter blok.

Dus na hoofdstuk 2, wordt het eerste deel van de uitdaging opgelost. Dan gaan we naar het tweede deel: het ontwikkelen van een bloksgewijze multilevel oplosser. Daarvoor is het noodzakelijk om de basis van het oplossen van grote ijle lineair systeem te herhalen. Dus in hoofdstuk 3, bekijken we de populaire Krylov deelruimte methoden en preconditionering technieken die een zeer belangrijke rol in de ontwikkeling van krachtige oplossers spelen. Vooral ook kijken we naar verschillende ILU-gebaseerde multilevel oplossers die een zeer gunstig kader voor het ontwerp van nieuwe solver vertonen.

Met alle voorbereidende introductie, komt het ontwerp van onze bloksgewijszen multilevel solver als vanzelf. We volgen het ARMS (Algebraic Recursive Multilevel Solver) kader, de nieuwheid van ons werk is het gebruik van het graaf gebaseerd compressie algoritme om de blokstructuur op te bouwen, en verder doen we exact hetzelfde als ARMS maar dan voor een blokstructuur. In Hoofdstuk 4 beginnen we met basis blokhandelingen voor blok gestructureerde matrices, en dan gaan we verder met gedetailleerde stappen van VBARMS (Variable Block Algebraic Recursive Multilevel Solver). In onze experimenten gebruiken we VABRMS als preconditioner in een Krylov deelruimte methode. Resultaten van numerieke experimenten toonden de effectiviteit en stabiliteit van VBARMS op blokgestructureerde matrices. Bovendien hebben we ook de resultaten weergegeven van de vergelijking tussen de klassieke hoek gebaseerde graafcompressiemethode en onze nieuwe graafgebaseerde compressie. De resultaten bevestigen dat de graafgebaseerde compressiemethode een gebruiksvriendelijker parameter tuning met vergelijkbare prestaties heeft. De resultaten van twee implementatiestrategieën voor VBARMS (uitvoering van expliciete Schurcomplementberekening en uitvoering van impliciete Schurcomplementberekening) werden ook tentoongesteld. Implementatie van impliciete Schurcomplementberekening toonde betere prestaties in termen van tijd.

Tot nu vertoonde VBARMS zeer goede prestaties op kleine testproblemen. Dit motiveert ons om het einde van de uitdaging te bereiken: de echte realistische problemen op te lossen. Dat vereist de integratie van VBARMS in een parallelle implementatie. Daarom wordt in hoofdstuk 5 terug geblikt op de basisprincipes van parallel rekenen, en twee parallelle blok solvers werden gepresenteerd: een op basis van de additieve Schwarz methode en de andere op basis van de Schur complement methode. De gedistribueerde parallelle uitvoering van VBARMS (pVBARMS) werden geïntroduceerd; pVBARMS toonde ook sterke prestaties en goede schaalbaarheid. Bovendien werd een nieuw Zoltan-gebaseerde graafpartitioneringsstrategie geïntroduceerd samen met zijn prestaties.

Tenslotte, voor een specifieke toepassing, grootschalige turbulent Navier-Stokes vergelijkingen, bezitten afgeleide matrices ook een blokstructuur. Hier bevestigen onze experimenten de trend van de prestaties die op algemene problemen.

Ijle matrices die voorkomen in het oplossen van systemen van partiele differentiaalvergelijkingen en hebben vaak een perfecte blokstructuur. Dit

betekent dat de niet-nul blokken in het ijlheidspatroon vol zijn (en meestal klein), dus dat er verschillende onbekenden geassocieerd zijn met dezelfde gridpunten. Echter, soortgelijke blokorderingen kunnen soms gevonden worden in een matrix zonder enige evidente blokstructuur door rijen en kolommen met een soortgelijke structuur achter elkaar te zetten in de matrix en door vervolgens sommige nul elementen als niet-nul elementen te beschouwen en deze als volle blokken op te slaan zonder dat dit veel geheugen kost. De herordening geeft lineaire systemen met blokken van variabele grootte.

Ons recent ontwikkelde parallelle softwarepakket pVBARMS (parallel Variable Block Algebraic Recursive Multilevel Solver) voor computers met gedistribueerd geheugen buit deze blokstructuren uit bij het ontwikkelen onze multilevel incomplete LU factorisatie preconditioner. Het maximaliseert de computationele efficiëntie en heeft een verhoogde doorvoer tijdens het rekenen en een verbeterde betrouwbaarheid op realistische applicaties. De methode detecteert automatisch bestaande blokstructuren in de matrix zonder enige kennis van de gebruiker van het onderliggende probleem en buit het uit voor maximale computationele efficiëntie.

Bovendien, in de context van gedistribueerde parallelle gegevensverwerking, worden twee graafpartitioneringsstrategieën voorgesteld: men gebruikt Zoltan bibliotheek om de gedistribueerde graaf te verfijnen, de andere verdeelt de seriële graaf op één processor en zendt deze uit naar andere processors. Beide strategieën combineren met pVBARMS blijkt zeer goede resultaten te hebben op het oplossen van instabiele, turbulente, Reynolds-averaged, Navier-Stokes vergelijkingen.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr. Bruno Carpentieri for giving me the opportunity of working in his group. Also Prof. dr. Arthur Veldman, I am thankful for your advices regarding to my research direction. I also would like to express my thanks to Prof. dr Masha Sosonkina for pleasant collaborations during my research project.

I want to acknowledge the member of the reading committee Prof. dr. Rob Bisseling, Prof. dr. Matthias Bollhöfer and Prof. dr. Roel Verstappen. They read the manuscript thoroughly and gave helpful comments.

I would like to thank my colleagues at University of Groningen for providing a positive work environment and pleasant interactions. Many interesting and inspiring technical discussions has be done during the 4 years. Special thanks go to Sven Baars for his contribution to Chapter 2 and translating the summary and abstract into Dutch.

I am grateful to everyone who ever helped me during this project.

At the end, I owe my gratitude to my beloved parents and cousin for unconditional support.