Efficiency Improvement of Panel Codes

Master Thesis Report

by

Ang Yun Mei Elisa (4420888)

in partial fulfilment of the requirements for the degree of

Master of Science
In Applied Mathematics

At the Delft University of Technology
To be defended publicly on Friday July 10, 2015 at 3.30 pm

Supervisor:	Dr. ir. M.B. van Gijzen,	TU Delft
MARIN Supervisor:	Dr. ir. A. van der Ploeg,	MARIN
Thesis committee:	Prof. dr. ir. C. Vuik,	TU Delft
	Dr. ir. H.X. Lin	TU Delft

PREFACE

Panel codes are used by the Maritime Reseach Institute Natherlands (MARIN) to compute flows around ships and propellers. These codes are based on Boundary Element Methods (BEM). A known drawback of BEM is that it forms dense linear system of equations that have to be solved. By improving the efficiency of the dense linear solver, the computational time required by panel codes can be significantly reduced. Since applications of panel codes in MARIN include automatic optimization, where a large number of hull forms or propeller geometries have to be evaluated, the reduction of computational time is important.

Four strategies were explored to improve the performance of the dense linear solver. First, to replace the current GMRES solver with IDR(s). Second, the updating of a fixed size block Jacobi preconditioner into a variable size block Jacobi preconditioner. Third, to use a hierarchical matrix-vector multiplication in the solver instead of dense matrix-vector multiplication. Lastly, to replace the block Jacobi preconditioner with a hierarchical-LU preconditioner. Out of the four strategies, the use of hierarchical-LU preconditioner was found to speed up the dense linear solver substantially, especially for large systems. The use of IDR(s) instead of GMRES is also recommended as it removes the problems introduced by the need to restart.

This report discusses the theory, implementation and test results obtained from the four strategies aforementioned. As a result of this project, the use of IDR(s) combined with hierarchical-LU preconditioner is recommended to be implemented in the panel codes.

The author would like to thank Dr. ir. Martin van Gijzen from TU Delft for his guidance throughout the project. To Dr. ir. Auke van der Ploeg from MARIN, thanks for all the valuable insights and support. It has been a pleasure to work on this project with both of you for the past nine months.

Ang Yun Mei Elisa

Delft, The Netherlands

June 21, 2015.

CONTENTS

1 INTRODUCTION

1.1 PROBLEM STATEMENT

At MARIN, Boundary Element Method (BEM) is used to compute flows around ships and propellers. Some examples of computer codes based on BEM, commonly known as panel codes, are stated below [1]:

1. FATIMA: Used to compute ship motions and added resistance from incoming waves
2. PROCAL: Used for the analysis of propellers
3. EXCALIBUR: Computes the hull pressure fluctuations induced by the propeller
4. RAPID: computes the wave system generated by the ship

The use of BEM results in a dense linear system of equation to be solved in every time step. This is unlike methods like Finite Element Method (FEM), where the system of equations formed is sparse. Thus, efficient linear solvers developed for FEM cannot be applied to BEM. There is a need to reduce the computational time required to solve this dense linear system of equations in MARIN. Therefore, this project seeks to explore ways to speed up the dense linear solver.

1.2 BACKGROUND

Currently, GMRES combined with incomplete LU-decomposition preconditioner, is used to solve the system of equations formed. M. de Jong had proposed the use of GMRES with Block-Jacobi preconditioner to solve the dense system of equations more efficiently [1]. Parallelization techniques using OpenMP and Graphics Processing Units (GPUs) were also studied to improve the performance of the Block-Jacobi preconditioner. The best solve times are listed in [1, Table 40] in the test environment stated in [1, Section 6].

On Sept 2014, a literature review was conducted to better understand the nature of BEM, and to assess the strategies available to address the problem [2]. The literature review divided the strategies into three main parts: different solver, different preconditioner, and different methods (known as Fast Multipole Method (FMM) and the Hierarchical Method). The results of the literature review suggested that the following strategies have the most potential. These methods are investigated in detail in this project.

1. Use of a different solver: the Induced Dimension Reduction solver (IDR(s))
2. Updating the current block Jacobi preconditioner to work for varying block sizes
3. Use of the Hierarchical method

The full literature review report can be found in [2].

1.3 REPORT PURPOSE AND OVERVIEW

The report summarizes the work done to improve the efficiency of the panel codes according to the strategies laid out at the end of the literature review. The report is arranged in chronological order in which these strategies were explored.

Section 3 discusses the advantages and disadvantages of the IDR(s) solver compared with the current GMRES solver. The theories of both GMRES and IDR(s) are first presented briefly, followed by a discussion on the integration of the IDR(s) solver into the current program. The results of the comparison are then discussed.

In Section 4, updating of the current block Jacobi preconditioner to include the ability to accept variable block Jacobi blocks is discussed. The theory, implementation and results are presented.

Section 5 is devoted to the use of hierarchical matrices to speed up matrix-vector multiplication (matvec) in the solver. In the literature review, focus was placed on the use of Lanzcos Bidiagonalization to perform the hierarchical splitting. In this report, an alternative method, known as the Adative Cross Approximation (ACA), is discussed and compared with the Lanzcos Bidiagonalization. The dense matvec operations in the solver are then replaced with hierarchical matvec. The corresponding implementation details and results are discussed within the section.

In Section 6, a new strategy that was not formulated at the end of the literature review is explored. This strategy, known as the hierarchical-LU decomposition, forms a lower and an upper triangular hierarchical matrix that can be used as a preconditioner. A thorough theoretical review is first presented, followed by details on the implementation in Fortran. Results are presented and discussed next.

The report then ends with a conclusion and recommendations for future work.

2 TEST ENVIRONMENT

Throughout the project, the new strategies are translated into Fortran codes and tested to evaluate their performance. The details of the test matrices and the system on which the tests were ran are given below. In Section 2.3, the best solve times that can be obtained in this test environment using the code described in [1] is presented. This is the baseline results that the new strategies are compared against.

2.1 TEST MATRICES

MARIN provided us with a few test matrices generated from their existing systems. The matrices are dense, and their characteristics are summarized below:

Name	Size	Real/Complex
Steadycav1	4620	Real
Steadycav2	4620	Real
Steadycav3	4620	Real
Steadycav4	4649	Real
Passcal	4400	Real
FATIMA_7894	7894	Complex
FATIMA_20493	20493	Complex

Table 1 Test Matrices

2.2 SYSTEM INFORMATION

Brand/Type Owner/ System no.	DELL TU DELFT/ TUD205717
CPU No. of cores Cache Memory	Intel® Core™ i5-4670 CPU @ 3.40GHz 4 256 KB x 4 L2/ 6 MB Smart L3 8 GB RAM DDR3-1333/1600
Motherboard Operating System System Kernel	Dell 0PC5F7 Windows 7
GPU Memory No of cores	Intel® HD Graphics 4600 1696 MB 20
OpenMP version	2.5

Table 2 System information

2.3 BASELINE TEST RESULTS

The final program resulting from the work of M. de Jong as presented in [1] were reran using the test matrices on the system described in Table 2. These set of results are used as a benchmark for comparison with the new strategies. The best solve times that are obtained are summarized below. The results are all obtained with OpenMP parallelization enabled, and number of physical cores set to 4. The other parameters are left the same as those described in [1, Section 6.2].

Test Matrix	**Number of right hand side**	**Block Jacobi Block size**	**Time(s)**
FATIMA_20493	1	4000	87.62
	7	4000	211.49
FATIMA_7894	1	1000	6.36
	7	1000	25.74
PASSCAL	1	500	0.72
Steadycav1	1	500	0.57
Steadycav2	1	500	0.60
Steadycav3	1	500	0.67
Steadycav4	1	500	0.67

Table 3 Best solve times before improvements

3 PART 1: SOLVER

This project focuses on two Krylov methods to solve the linear system: Generalized Minimized Residual (GMRES) and Induced Dimension Reduction (IDR). GMRES is the method used currently, while IDR(s) is the new solver that is integrated into the current program.

The literature report in [2] details the literature review done on GMRES and IDR(s). This includes the mathematical concept, pseudo-algorithm, and performance analysis. A summary of the theory is given here, but the reader is referred to [2] for details. Following the theory, the implementation details and comparison results are described.

3.1 THEORY

The following subsections briefly summarizes the concepts, advantages and disadvantages of the GMRES and IDR(s) solver.

3.1.1 GMRES

GMRES is the most common iterative method employed to solve $Ax = b$, when A is not hermitian. At every iteration m, it approximates the exact solution, x_*, with a vector x_m that resides in the Krylov space $\mathcal{K}_m$, such that the residual $\|r_m\| = \|b - Ax_m\|$ is minimized. A Krylov space, $\mathcal{K}_m$, is defined as the space $span\{b, Ab, A^2b, \dots, A^{m-1}b\}$. The main advantage of GMRES is that it is optimal, since at every step the residual is minimized. In addition, only one matrix-vector multiplication is required per iteration. The main disadvantage of the GMRES is that it is a long-recurrence method. This means that the work and storage required increase with iteration. Therefore, in most cases, including this application, GMRES is implemented with restart to prevent the work and storage requirement from growing too large. This means that after k number of iterations, the algorithm is restarted with x_k as the initial guess.

3.1.2 IDR(s)

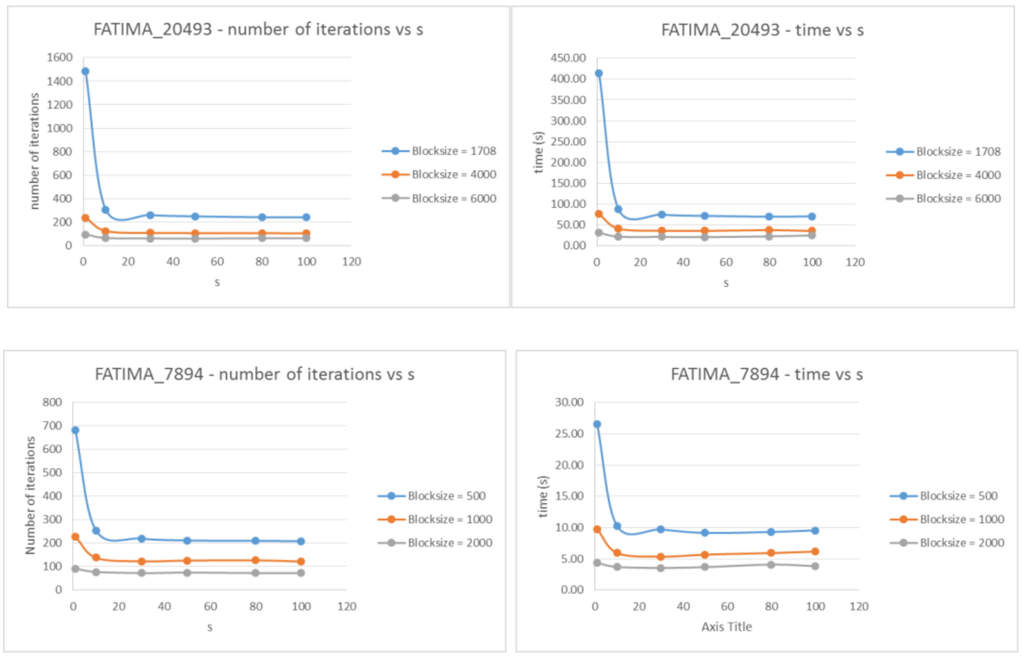
The IDR method was introduced by Sonneveld, P. & van Gijzen, M.B. in 2008 [3]. Unlike GMRES, IDR(s) is a short recurrence method. The depth of recurrence depends on the parameter s. Also, instead of increasing the subspace with every iteration, IDR(s) uses the concept of nested subspaces $\mathcal{G}_j$, where $\mathcal{G}_j \subset \mathcal{G}_{j-1}$ and $\mathcal{G}_0 =$the full Krylov subspace, $\mathcal{K}(A, r_0)$. The main benefit of IDR(s) over GMRES is its short recurrence, therefore ensuring that storage and work remains constant with increasing iterations. It also has the benefit over other short recurrence methods like bi-CG of requiring at most $N + \frac{N}{s}$ matrix-vector product to arrive at the exact solution, where N is the problem size, and s is the codimension of a fixed subspace [2].

The main mathematical concept behind IDR(s) is to search for the residual $r_m \in \mathcal{G}_j$. The IDR theorem states that the space $\mathcal{G}_j$ is shrinking as j increases, and there will be some $j \leq N$ during which the space $\mathcal{G}_j$ reduce to just null space. Thus, this means that the residual will be **0** at some point $j \leq N$ and the exact solution is found.

3.2 IMPLEMENTATION

A Fortran implementation of IDR(s) was adapted from Martin van Gijzen [4], and integrated with the current solver such that the user can choose to use GMRES or IDR(s) to solve the system.

An open question remains as to what is the value of s to use. To investigate this, the IDR(s) solver was ran with the test matrices for different values of s. The results are graphed as shown in Figure 1. The figures on the left represent the trend lines for the number of iterations required to solve the system with IDR(s), while those on the right represent the time required. The time shown here is purely the time taken for IDR(s) and does not include the time required to construct the preconditioner.



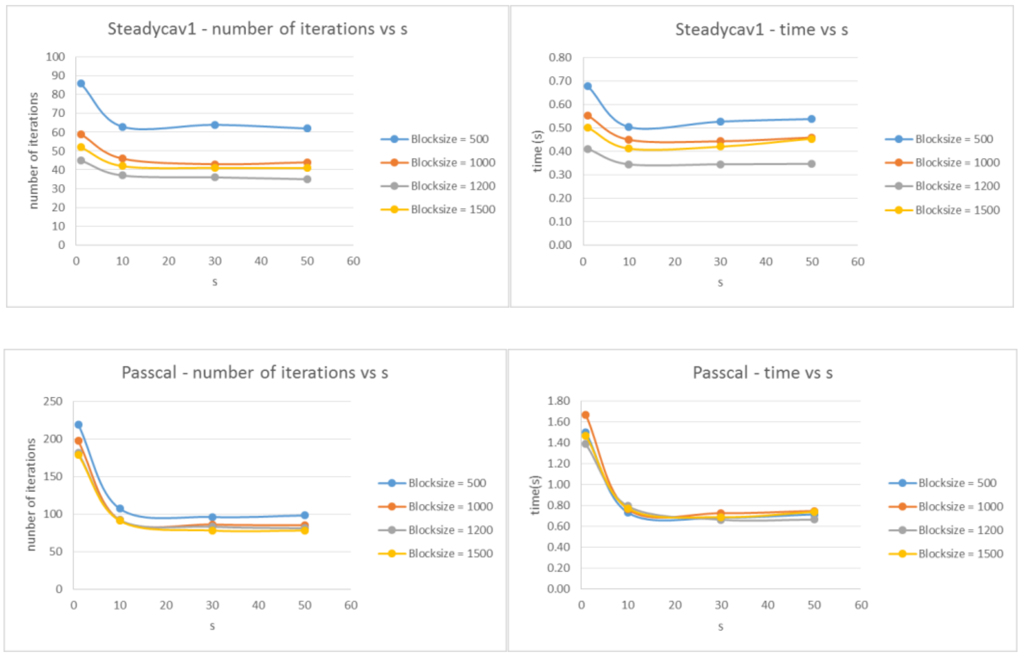


Figure 1 Effect of parameter s on the number of iteration and time to IDR(s) solve time for different test matrices.

The optimal s value for FATIMA_20493 is around 50. When s is small ($s \leq 20$), there is a significant drop in the number of iterations as s increases. This is especially true for small block Jacobi block sizes, where the system is less well-conditioned. For well-conditioned system (in this case when block Jacobi block size is around 6000), the effect of increasing s reduces. The reduction continues until s reaches about 50. After this, number of iteration remains roughly constant even when s increases. This effect can be explained by the fact that IDR(s) can never outperform GMRES (without restart) in terms of the number of iterations required, since GMRES is optimal. Thus, there is a limit as to how much the number of iterations can be reduced by increasing s.

Since s determines the depth of recursion, when s increases, the amount of work (and storage) per iteration increases. Thus, a slight rise in time as s increases beyond 50 can be observed, since the number of iterations remain constant, but each iteration involves more work. The rise in time is slight, as the main bulk of the time is still taken up by the matvec operations due to the dense nature of the system.

The same trend can be observed for the FATIMA_7894 test matrix, but with the optimal value closer to 30 instead. Although for the less well-conditioned system (when block Jacobi block size is 500), s still seems to be optimal at a higher value of 50, the better conditioned system shows a lower optimal value of s.

All the smaller Steadycav matrices exhibit the same trend. Therefore, only the results for Steadycav1 are depicted here. In this case, the optimal value of s is at 10. When s increases beyond 10, the number of iterations remain constant, thus, a slight increase in timing is again observed. Passcal matrix seems to be optimal with s = 30.

With these observations, the values of s used are 50 for FATIMA_20493, 30 for FATIMA_7894 and Passcal, and 10 for the Steadycav matrices. For other systems, initial assessments need to be carried out to decide on an optimal value of s.

3.3 RESULTS AND DISCUSSION

The test matrices were solved both using GMRES and IDR(s), with the following configuration.

- OpenMP turned on for the expensive parts of the operation (matrix-vector or matrix matrix multiplication, construction of block Jacobi preconditioner, application of block Jacobi preconditioner), with number of cores set to 4
- Mixed precision (#define PRECISION_zc or #define PRECISION_ds)
- Tolerance for relative residual (Exit criteria for solvers) set to 1e-09
- GMRES restart after 200 iterations
- Number of right hand side (nrhs) vectors is 1. The results for multiple right hand side (RHS) exhibits the same trend as when the number of RHS is 1, and therefore, are not elaborated on here

The complete results can be found in Table 4. An explanation of each column is provided below. The subsections that follow discuss these results.

- **Prec const:** Time spent on the LU-factorization of the block Jacobi preconditioner
- **Prec apply:** Time spent on applying the block Jacobi preconditioner during solving
- **Matvec:** Time spent on matrix-vector multiplication during solving
- **Solve:** Total time spent on GMRES or IDR(s) routine. The total solve time is approximately the sum of Prec apply and Matvec timings.
- **Total:** Total wall clock time. This is roughly the sum of Solve and Prec const timings.
- **#iter:** Total number of iterations required for solving
- **Rel error:** Final relative error computed using the formula $||x_{solution} - x_{exact}||_1$[1]

[1] The exact solution is known because in the test program, the right hand sides are constructed with specified solution.

Matrix	Blocksize for block Jacobi	GMRES							IDR(s)						
		Wall clock time (s)					#iter	Rel error	Wall clock time (s)					#iter	Rel error
		Prec const	Prec apply	Matvec	Solve	Total			Prec const	Prec apply	Matvec	Solve	Total		
FATIMA_20493	1708	6.58	7.89	104.60	116.08	122.66	393	1.49E-06	6.51	5.08	67.26	73.94	80.45	260	6.20E-07
	4000	53.19	6.74	27.19	34.44	87.62	103	1.17E-07	53.06	7.31	28.65	36.57	89.75	110	2.45E-07
	6000	267.00	4.51	15.92	20.64	287.64	60	2.27E-07	266.28	5.12	17.59	23.07	292.51	66	1.36E-07
FATIMA_7894	500	0.23	0.59	8.55	9.83	10.06	231	7.03E-07	0.23	0.61	8.66	9.54	9.77	237	2.79E-07
	1000	0.90	0.58	4.63	5.47	6.36	121	1.57E-07	0.88	0.64	4.90	5.68	6.55	133	1.14E-07
	2000	12.56	0.67	2.81	3.58	16.14	74	1.50E-07	12.52	0.74	3.05	3.89	16.70	82	5.93E-08
Steadycav1	500	0.08	0.05	0.42	0.48	0.57	61	2.53E-04	0.08	0.06	0.47	0.54	0.62	68	5.63E-05
	1000	0.40	0.10	0.31	0.42	0.82	43	2.23E-04	0.40	0.11	0.33	0.45	0.86	47	2.85E-05
	1200	1.23	0.06	0.24	0.31	1.54	35	6.27E-05	1.23	0.07	0.28	0.35	1.58	39	1.41E-05
	1500	1.99	0.10	0.28	0.39	2.38	41	1.74E-04	1.98	0.11	0.31	0.43	2.41	45	4.10E-04
Steadycav2	500	0.08	0.06	0.45	0.52	0.60	65	2.59E-04	0.08	0.06	0.50	0.57	0.65	72	6.73E-07
	1000	0.40	0.10	0.33	0.44	0.84	47	2.39E-05	0.49	0.12	0.37	0.49	0.90	52	1.79E-04
	1200	1.23	0.06	0.26	0.33	1.57	38	8.84E-06	1.23	0.08	0.31	0.39	1.62	43	1.22E-04
	1500	1.99	0.11	0.30	0.42	2.40	44	7.16E-05	1.98	0.13	0.35	0.48	2.46	50	2.13E-04
Steadycav3	500	0.08	0.06	0.51	0.58	0.67	70	1.60E-02	0.08	0.06	0.51	0.59	0.67	73	1.23E-02
	1000	0.41	0.11	0.34	0.46	0.87	50	2.49E-03	0.40	0.12	0.37	0.49	0.89	53	1.06E-02
	1200	1.23	0.07	0.28	0.35	1.58	40	2.74E-04	1.23	0.08	0.32	0.40	1.63	45	6.79E-04
	1500	1.99	0.11	0.33	0.45	2.44	47	6.92E-03	1.98	0.14	0.40	0.55	2.53	57	3.41E-03
Steadycav4	500	0.08	0.06	0.50	0.58	0.67	72	8.05E-05	0.08	0.07	0.60	0.69	0.77	87	7.80E-06
	1000	0.40	0.11	0.35	0.47	0.88	50	1.09E-05	0.40	0.12	0.38	0.52	0.92	54	3.28E-05
	1200	1.25	0.07	0.28	0.36	1.60	40	1.19E-05	1.24	0.08	0.33	0.42	1.67	47	6.73E-06
	1500	1.99	0.11	0.33	0.45	2.44	47	3.02E-05	1.98	0.13	0.38	0.52	2.50	54	2.71E-05
Passcal	500	0.08	0.08	0.54	0.64	0.72	91	3.54E-07	0.08	0.08	0.57	0.69	0.76	96	5.19E-07
	1000	0.40	0.18	0.47	0.67	1.07	81	5.59E-07	0.40	0.19	0.50	0.73	1.13	86	1.58E-06
	1200	0.99	0.13	0.46	0.60	1.560	77	2.99E-07	0.99	0.14	0.50	0.67	1.65	83	2.06E-07
	1500	1.83	0.17	0.43	0.62	2.45	73	1.28E-06	1.83	0.19	0.50	0.69	2.52	78	3.15E-06

Table 4 Comparison between GMRES and IDR(s). The best solve time, total time, and number of iterations required for each case is highlighted in green.

3.3.1 Number of iterations

First, attention is given to comparing the number of iterations GMRES and IDR(s) take to solve the problem. In most cases, IDR(s) takes a few more iterations as compared to GMRES. This is due to the fact that GMRES is optimal, thus, it is expected that GMRES uses the lowest number of iterations required to solve the system. However, when restart is required for GMRES, it can be seen that IDR(s) could require a significantly lower number of iterations. This is exemplified by the FATIMA_20493 matrix, with a block Jacobi block size of 1708. GMRES requires 393 iterations, while IDR(s) requires only 260 (Refer to Table 4).

The reason for this behavior can be explained using the relative residual plots, shown in Figure 2. The usual cases for GMRES without restart are shown in the first three plots. GMRES always display a faster convergence, but IDR(s) stays close to this convergence behavior of GMRES. The last figure shows the case for the FATIMA_20493 matrix with block Jacobi block size of 1708, where a restart is required. It can be observed that at the onset of a restart, the relative residual behavior of GMRES is to plateau, and then converge steeply again. Since restart is not required in IDR(s), the plateau behavior is not observed and the number of iterations required is therefore lower. Although GMRES without restart gives again the lowest number of iterations required, its use is prohibitive due to the increase in the work and storage requirement with iterations. On the other hand, IDR(s) do not require restart since it is a short recurrence method. Therefore, in such cases, it is clear that IDR(s) has a distinct advantage over GMRES.

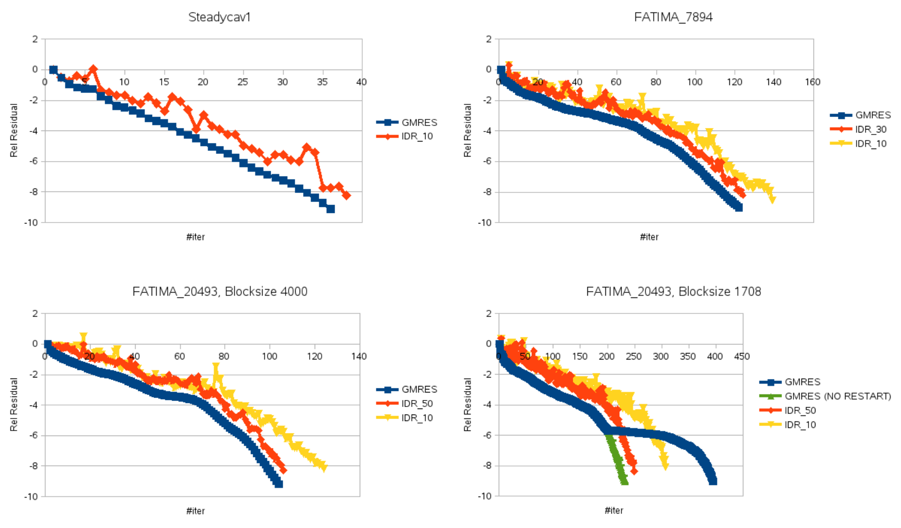


Figure 2 Relative residual plots for GMRES and IDR(s)

3.3.2 Timings

After studying the number of iterations required, the focus is now on the time required to solve the system. One important benefit that IDR(s) have over GMRES is that less amount of work is required per iteration. Thus in theory, IDR(s) can afford to have more iterations, and may still perform better than GMRES in terms of timings. However, because the system here is dense, the dominant work in every iteration is the matrix-vector multiplication. This can be seen from the timings presented in Table 4. It can be observed that the matvec time takes up about 70% to 90% of the solve time. Both GMRES and IDR(s) needs one matrix-vector multiplication per iteration. Thus although each iteration of IDR(s) may require less work than GMRES, the additional few number of iterations required for IDR(s) to solve the system dominates over this. Therefore, the time required for IDR(s) is slightly higher as compared to GMRES in most cases.

The case for the FATIMA_7894 system preconditioned with block Jacobi block size of 500 is one example where the gain of IDR(s) over GMRES can be observed. Although the number of iterations for IDR(s) is slightly higher as compared to GMRES (237 and 231 respectively), the time required for IDR(s) is 9.5s, as compared to the 9.8s required for GMRES. The time spent on work other than matvec for GMRES is about 1.3s while that for IDR(s) is 0.9s. While this gain is recognized, the overall effect on the total time is still not significant.

3.3.3 Memory requirement

The memory required for GMRES and IDR(s) differs in the number of vectors from previous iterations that have to be stored. In the Fortran implementation of the GMRES method, the amount of memory allocated to store the vectors from previous iterations is a size $N \times nrhs \times gmres\_restart$ array. The term $gmres\_restart$ defines the number of iterations at which a restart is invoke. In the case of IDR(s), the amount of memory allocated to store the vectors from previous iterations is three size $N \times nrhs \times s$ arrays. Since $3s$ is expected to be smaller than $gmres\_restart,$ the amount of memory required for IDR(s) is lower.

However, as the systems here are dense, the main bulk of the memory is allocated to store the system matrix. The reduction in memory required for IDR(s) as compared to GMRES is insignificant in these cases.

In conclusion, the performance of IDR(s) is close to that of GMRES for most cases. From the test matrices, it was observed that IDR(s) solver outperforms GMRES significantly in cases when GMRES restart is required. In the case of FATIMA_20493 with block Jacobi block size of 1708, the total time required to solve the system using GMRES is 122.7s, while that for IDR(s) is only 80.5s. Thus, using IDR(s) instead of GMRES can lead to a substantial performance gain when restart is required, in the expense of slightly higher computational time when restart is not invoked.

4 PART 2: VARIABLE SIZE BLOCK JACOBI PRECONDITIONER

For some applications in MARIN, variable Jacobi blocks are of interest. Consider the case of simulation of ships' interactions. The resulting test matrix has a natural block structure, illustrated below:

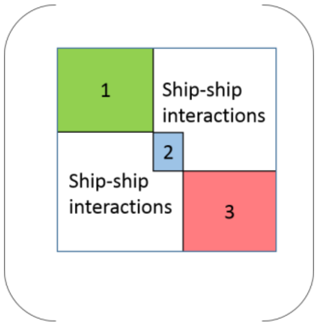


Figure 3 Structure of a typical matrix derived from ship simulations

The main diagonal has blocks with elements which represent interactions of panels belonging to the same ship. The off diagonal blocks represent interactions between panels belonging to different ships. For such applications, at each time step, the elements within the main diagonal blocks do not change. Only the off diagonal elements are updated. Thus, if variable sized blocks in the block Jacobi preconditioner can be implemented, the LU decomposition for the block Jacobi preconditioner needs only to be done once, and can be used for the rest of the time steps.

The subsections below discusses first the implementation of variable size block Jacobi preconditioner. The results are then discussed in the next section.

4.1 IMPLEMENTATION

To adapt the current implementation to cater to varying block Jacobi block sizes, the following changes were made:

1. A new derived type, LU_blocks, was defined to store the variable sized block Jacobi blocks. LU_blocks is made up of the following 1D arrays.
    a. $\boldsymbol{LU\_1D}$: used to store the elements in all the block Jacobi blocks. The size of this array is hence $\sum_{i=1}^{\#\ jacobi\ blocks} size_{jacobi\ block\ i}^{2}$

b. $\boldsymbol{pivot\_1D}$: used to store the pivot array that comes from LU decomposition. The size of this array is N.

c. $\boldsymbol{LUpos\_container}$: used to store the start index of the first element of each block in LU_1D. The purpose of this container is to allow the easy access of the elements in each Jacobi block. The size of this array is the total number of block Jacobi blocks.

d. $\boldsymbol{pivotpos\_container}$: used to store the start index of the first element of each block in $pivot\_1D$. The purpose of this container is to allow the easy access of the pivot element of each Jacobi block. The size of this array is the total number of block Jacobi blocks.

```
type, public ::    LUblocks
         DATATYPE1,           allocatable, dimension (:) :: LU_1D
         integer(kind=SHORT), allocatable, dimension (:) :: pivot_1D
         integer(kind=SHORT), allocatable, dimension (:) :: LUpos_container
         integer(kind=SHORT), allocatable, dimension (:) :: pivotpos_container
         integer(kind=SHORT)                             :: LU_1D_size
         integer(kind=SHORT)                             :: pivot_1D_size
end type LUblocks
```

Figure 4 Data structure for $LUblocks$

2. The existing codes were updated to use this derived type instead of the 3D arrays previously used to store the block Jacobi blocks. To illustrate how this was done, assume that a $LUblocks$ typed object named $LUpivot$ is declared and constructed. Then, the product of the LU-factorization of each block Jacobi blocks are stored in $LUpivot$ by the following code fragment.

   To help understand the code fragment, some definitions are given here:

   - $nrblocks$ denotes the total number of block Jacobi blocks
   - lwb denotes the first row or column number corresponding to a particular block
   - upb denotes the last row or column number corresponding to a particular block
   - $blockposition$ denotes the first element corresponding to a particular block in the LU_1D array
   - $Nloc$ denotes the total number of columns or rows belonging in a block
   - $return\_pos$ is a helper function that returns the position of an element in the LU_1D array given its local index i, j in its block Jacobi block number.

```
do kw = 1,nrblocks
       lwb=LUpivot%pivotpos_container(kw)
       blockposition=LUpivot%LUpos_container(kw)
       upb = min ( N,lwb-1+blocksize_container(kw) )

    ! Compute actual block size: may be smaller than 'blocksize'
    ! for the last block
    ! ---------------
    Nloc = upb+1-lwb

    ! Copy the 'kw-th' main diagonal block of A to LU(:,:,kw)
    ! --------------------------------------------------------
    do k = lwb,upb
        call F1COPY ( Nloc,A(lwb,k),1, &
       LUpivot%LU_1D(return_pos(blockposition, Nloc, 1, k-lwb+1)),1 )
    end do


    ! Compute LU-factorization of block on the main diagonal
    ! Note: sequential within the block
    ! -------------------------------------------------------

    call F1GETRF ( Nloc,Nloc,LUpivot%LU_1D(blockposition), &
     blocksize_container(kw), LUpivot%pivot_1D(lwb),info )

    ! Store 1/diag element
    ! ---------------------
    do k = 1,Nloc
       LUpivot%LU_1D(return_pos(blockposition, Nloc, k, k))&
       = F1ONE / LUpivot%LU_1D(return_pos(blockposition, Nloc, k, k))
   end do

end do
```

Figure 5 Code fragment illustrating the LU-factorization of the variable sized block Jacobi blocks

4.2 RESULTS AND DISCUSSION

Passcal is an example of a matrix that has the structure shown in Figure 3. It has three main diagonal blocks, with sizes 1600, 1600 and 1200 respectively. The results obtained using the variable size block Jacobi preconditioner is shown in Table 5, together with the best results obtained using the fixed size block Jacobi preconditioner (block size = 500) for comparison. The same setting as described in Section 3.3 is used here.

In this case, the variable size block Jacobi preconditioner does not provide a significant gain as expected. It was hypothesized that by using a preconditioner that takes into account the natural block structure, the number of iterations required could be lowered. However, it can be seen that the number of iterations required for the variable size block Jacobi preconditioner in this case (which is 80 for GMRES) is even slightly more than the corresponding fixed size block Jacobi preconditioner (block size of 1200 requires 77 iterations).

Moreover, although there is only a one time effort to construct the variable size block Jacobi preconditioner, the main diagonal blocks are usually large, hence, the time taken to apply the preconditioner at every time step is also high. On the other hand, although there is a need to construct the fixed size block Jacobi preconditioner at every iteration, the preconditioner that performs best in this case has a small block size. This means that the construction and application of the fixed size preconditioner is relatively cheap.

Table 6 shows a comparison in the time taken to solve the system with a fixed or variable size block Jacobi preconditioner for different number of time steps required. In the case of GMRES, only when the time steps required exceed 1000, then there will be about 1-2% gain in time. In the case of IDR(s), since the solve time for variable size block Jacobi preconditioner is already more than the total time required for the fixed size block Jacobi preconditioner, it can be seen that the fixed size block Jacobi preconditioner performs better.

To sum up, for this example, the use of variable size block Jacobi preconditioner does not show a significant gain. Since it was shown that the number of iterations did not reduce by exploiting the natural block structure, the benefit of a variable size block Jacobi preconditioner is limited.

Matrix	Blocksize for block Jacobi	GMRES							IDR(30)						
		Wall clock time (s)					#iter	Rel error	Wall clock time (s)					#iter	Rel error
		Prec const	Prec apply	Matvec	Solve	Total			Prec const	Prec apply	Matvec	Solve	Total		
Passcal	500	0.08	0.08	0.54	0.64	0.72	91	3.54E-07	0.08	0.088	0.57	0.69	0.76	96	5.19E-07
	1600, 1600, 1200	1.54	0.21	0.47	0.71	2.25	80	4.66E-07	1.54	0.24	0.54	0.81	2.35	88	2.35E-06

Table 5 Results for variable size block Jacobi preconditioner compared with fixed size block Jacobi preconditioner. The highlighted values are used in Table 6.

	GMRES		IDR(30)	
Total number of time steps	Time with fixed sized block Jacobi preconditioner (s)	Time with variable sized block Jacobi preconditioner (s)	Time with fixed sized block Jacobi preconditioner (s)	Time with variable sized block Jacobi preconditioner (s)
100	71.8	72.24	76.4	82.24
1000	718	708.54	764	808.54
10000	7180	7071.54	7640	8071.54

Table 6 Performance of fixed and variable block Jacobi blocks for different time steps

5 Part 3: Hierarchical Method to speed up matrix-vector multiplication

In the literature review [2], the theory of Fast Multipole Method was presented. The Fast Multipole Method (FMM) is introduced by Rokhlin and Greengard in 1980s [5]. It has the benefit of allowing matvec operations to be performed in $O(N)$ complexity, where N is the size of the system matrix. However, it requires kernel and domain information to build up the low rank approximation via series expansion.

In 1999, Hackbush introduced the hierarchical matrices (H-matrices) [6]. Here, it is assumed that the system matrix is given. The matrix is then split hierarchically, and each block is approximated by a low rank approximation. This structure allows the method to be implemented as a black box, without knowing any domain or kernel information. This makes it simpler to implement. In general, the use of H-matrices brings the complexity of matrix-vector multiplication down to $O(NlogN)$ [7]. Because of the possibility to implement this method as a black box, this project choose to focus on the H-matrices instead of FMM.

With the construction of H-matrix, each dense matvec performed by the solver can be replaced with a hierarchical matvec. This is desired, since the dense matvec is the most time consuming operation in the solver, with complexity $O(N^2)$.

The following subsections discuss first the theory of hierarchical method. The implementation details of the hierarchical matrix in Fortran and its integration in the solver are presented next. The results are then presented and the strategy is evaluated.

5.1 Theory

This section addresses the theory behind constructing a hierarchical form of a matrix, and how it can be used to reduce the complexity of matvec operation from $O(N^2)$ to $O(NlogN)$. The construction of a H-matrix includes the hierarchical splitting of a matrix A into blocks, determination of whether each block is low rank, and then obtaining the low rank approximation if they exist.

5.1.1 Hierarchical Splitting of a Matrix A

The process of hierarchically partitioning A into blocks is illustrated in Figure 6.

At level $l = 0$, A is not partitioned. At level $l = 1$, A is partition into 4 blocks. At $l = 2$, A is partition into 16 blocks and so on until $l = levels$, when each block is deemed to be small enough. Each matrix block obtained from level l is represented by the symbol $M_{\sigma,\tau}(l)$, where σ, τ are the local row and column block numbers of level l, and $\sigma, \tau \in \{0,1, \dots, 2^l\}$. Each block has row and column size indicated by $size_\sigma$ and $size_\tau$.

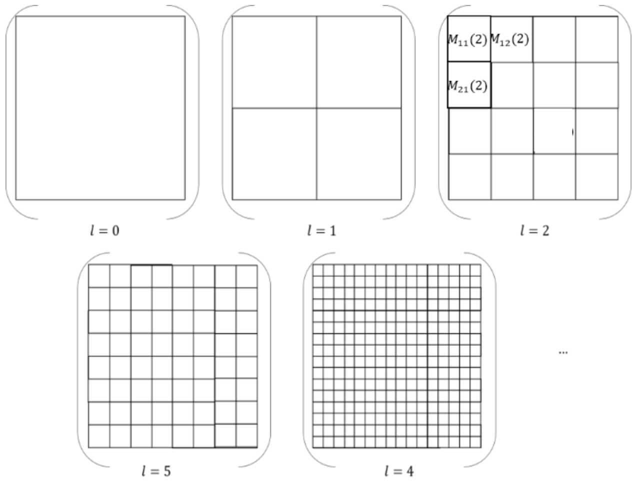


Figure 6 Hierarchical Partitioning of Matrix A

5.1.2 Low Rank Approximation

With A split hierarchically, the next step is to determine if each block is of low rank. There are different ways to decide if a block is low rank, and if so, construct the low rank approximation of the block. Two methods are studied here: the Lanzcos Bidiagonalization method and the Adaptive Cross Approximation (ACA) method.

A block is of low rank if the matrix block can approximated as shown in Equation 1 or Equation 2.

$$M_{\sigma,\tau}(l) \approx \widetilde{M}_{\sigma,\tau}(l) = U_{\sigma,\tau} B_{\sigma,\tau} V_{\sigma,\tau}^T$$

Equation 1

$$M_{\sigma,\tau}(l) \approx \widetilde{M}_{\sigma,\tau}(l) = \sum_{k=1}^{p} u_k v_k^T = U_{\sigma,\tau} V_{\sigma,\tau}^T$$

Equation 2

Where $U_{\sigma,\tau} \in \mathbb{C}^{size_\sigma \times p}$, $V_{\sigma,\tau} \in \mathbb{C}^{size_\tau \times p}$, $B_{\sigma,\tau} \in \mathbb{C}^{p \times p}$, $u_k \in \mathbb{C}^{size_\sigma \times 1}, v_k \in \mathbb{C}^{size_\tau \times 1}$, and $p \ll size_\sigma \ or \ size_\tau$.

If the blocks can be approximated by Equation 1 or Equation 2, it can be said that the rank of each block is approximately p. Lanzcos Bidiagonalization approximates a block with Equation 1, while ACA uses Equation 2. The two methods are discussed further in the following subsections.

5.1.2.1 Lanzcos bidiagonalization

A quick summary of the Lanzcos Bidiagonalization method is first given. Consider the reduction of $A \in \mathbb{C}^{mxn}$ into bidiagonal form [8]:

$$A \begin{bmatrix} v_1 & |\dots| & v_n \end{bmatrix} = \begin{bmatrix} u_1 & |\dots| & u_m \end{bmatrix} \begin{bmatrix} \alpha_1 & \beta_1 & & \\ & \alpha_2 & \ddots & \\ & & \ddots & \beta_{n-1} \\ & & & \alpha_n \end{bmatrix}$$

Equation 3

Let $U = [u_1 \quad |\dots| \quad u_m]$ and $V = [v_1 \quad |\dots| \quad v_n]$. U and V are required to be orthogonal matrices. Also, let B be the bidiagonal matrix $\begin{bmatrix} \alpha_1 & \beta_1 & & \\ & \alpha_2 & \ddots & \\ & & \ddots & \beta_{n-1} \\ & & & \alpha_n \end{bmatrix}$. Thus, Equation 3 can be written as $AV = UB$.

For the kth column, the following can be written

$$\alpha_k u_k = A v_k - \beta_{k-1} u_{k-1}$$

Equation 4

Since $AV = UB$, then $A^H U = V B^H$ must be also true. This provides us with another formula for the kth column:

$$\beta_k v_{k+1} = A^H u_k - \alpha_k v_k$$

Equation 5

Thus, if any unit vector v_1 is specified and assume β_0 is 0, then Equations 4 and 5 form the recurrence relation required to obtain u_k and v_{k+1} at every step k. The two scalers α_k and β_k are chosen to normalize u_k and v_{k+1}. This algorithm is described in many literature, for example [8].

The algorithm is modified such that the decomposition is terminated after p steps. Hence, the dimensions of the decomposed matrix block are:

$$U \in \mathbb{C}^{mxp}, \qquad V \in \mathbb{C}^{nxp}, \qquad B \in \mathbb{C}^{pxp}$$

Again, let each matrix block in the hierarchical division of A be $M_{\sigma,\tau}(l)$. Applying the Lanzcos Bidiagonalization algorithm to $M_{\sigma,\tau}(l)$ gives us an approximation $\widetilde{M}_{\sigma,\tau}(l) = UBV^H$ (Note that the subscripts σ, τ on U, V and B are dropped to avoid cluttering). To check if $M_{\sigma,\tau}(l)$ is admissible, the pth diagonal element from B, $B(p,p)$, is checked to see if it has decreased below a tolerance, tol_hie. The rationale for this is described in [2].

5.1.2.2 Adaptive Cross Approximation (ACA)

Bebendorf introduced the ACA method in 2000 [9]. Like the Lanzcos Bidiagonalization method, it seeks to find an approximation to blocks that are of low rank. While Lanzcos Bidiagonalization approximates the low rank blocks with Equation 1, ACA approximates the low rank blocks using the outer products as described in Equation 2 [9]. In this section, the main equations governing the ACA algorithm is first introduced. The formulation of the admissibility criterion is next presented. Lastly, a discussion on the different ways to choose the rows and columns required during the algorithm is discussed.

The theory behind ACA described here is referenced mainly from [7] and [9].

5.1.2.2.1 Main equations

The ACA works by first separating the matrix block M, into an approximation matrix S, and a residual matrix R.

$$M = R + S$$

Initially, $R_0 = M$ and $S_0 = 0$. At each iteration, a row i_k and a column j_k are chosen. It is assumed for now that these choices are known. Let $e_{i,size_\sigma}$ represents the i^{th} column of the identity matrix $I^{size_\sigma x size_\sigma}$. At each step of the iteration, R and S is computed based on the recursive relation:

$$\gamma_{k+1} = \left(e^T_{i_{k+1},size_\sigma}\, R_k\, e_{j_{k+1},size_\tau}\right)^{-1} = \frac{1}{R_k(i_{k+1}, j_{k+1})}$$

$$R_{k+1} = R_k - \gamma_{k+1} R_k e_{j_{k+1},size_\tau} e^T_{i_{k+1},size_\sigma} R_k$$

$$S_{k+1} = S_k + \gamma_{k+1} R_k e_{j_{k+1},size_\tau} e^T_{i_{k+1},,size_\sigma} R_k$$

Equations 6 [9]

It can be observe that $R_k e_{j_{k+1},size_\tau}$ just represents the j_{k+1} column of R_k, and $e^T_{i_{k+1},size_\sigma} R_k$ represents the i_{k+1} row of R_k. Let:

$$u_k = R_{k-1} e_{j_k}$$

$$v_k = \gamma_k e^T_{i_k} R_{k-1}$$

Equations 7

where u_k and v_k are the same as in Equation 2. The relation between Equations 7 and Equation 2 can be seen by considering the set of Equations 6, which now can be written as:

$$\gamma_{k+1} = \frac{1}{R_k(i_{k+1}, j_{k+1})}$$

$$R_{k+1} = R_k - u_{k+1} v_{k+1}$$

$$S_{k+1} = S_k + u_{k+1} v_{k+1}$$

Equations 8

Since $S_0 = 0$, we see that the equation for S_{k+1} in Equations 8 corresponds to Equation 2. Equations 7 and 8 together form the basis of the ACA algorithm.

For an efficient implementation, S and R matrices should not be explicitly built. This is because the building of these matrices require outer products, which is expensive ($O(N^2)$). Instead, throughout the algorithm, only u_k and v_k are considered, and the entire matrices of R and S are not needed. The column vector u_k is basically the j_k column of R_{k-1}. Consequently, only information regarding the j_k column of R_{k-1} needs to be known. Similarly, only information regarding the i_k row of R_{k-1} is required to compute v_k. With this in mind, we can reformulate the set of equations as:

$$u_k = R_{k-1}(:, j_k) = M - \sum_{i=1}^{k-1} v_i(j_k) * u_i$$

$$\gamma_k = \frac{1}{R_{k-1}(i_k, j_k)}$$

$$v_k = \gamma_k R_{k-1}(i_k, :) = \gamma_k \left(M - \sum_{i=1}^{k-1} u_i(i_k) * v_i \right)$$

Equation 9 [9]

5.1.2.2.2 Admissibility criterion

What is still lacking is an admissibility criterion to determine if the approximation S_p to M is good enough. The naïve implementation of $\|R_p\|_F \leq \varepsilon \|M\|_F$ is not feasible, since this requires again the expensive operation of building up of the matrix R_p. In addition, computation of the Frobenius norm of a matrix is also an $O(N^2)$ operation. Instead, $\|u_p v_p\|_F$ provides a good approximation to $\|R_p\|_F$ [9]. The outer product $u_p v_p$ can be avoided by using the following identity:

$$\|u_p v_p\|_F = \|u_p\|_2 \|v_p\|_2$$

$\|M\|_F$ can be approximated by $\|S_p\|_F$, which can also be computed without explicitly constructing the outer product using the recurrence relation [7]:

$$\|S_{k+1}\|^2{}_F = \|S_k\|^2{}_F + 2 \sum_{i=1}^{k-1} \left|u_i^T u_k\right| \cdot |v_i v_k^T| + \|u_k\|_2^2 \|v_k\|_2^2$$

With these, the appropriate admissibility criterion is $\|u_p\|_2 \|v_p\|_2 \leq \epsilon \|S_p\|_F$.

5.1.2.2.3 Choice of rows i_k and columns j_k at iteration k

The above equations were formulated based on the assumption that the choice of i_k and j_k is known at every iteration. This choice must be addressed to formulate the algorithm. One obvious choice is to choose i_k and j_k to coincide at the most dominant element in the matrix R_k at each step k. This is known as the Fully Pivoted ACA [10]. Choosing i_k and j_k this way ensures that the most dominant element of the matrix block is always included first in the approximation, making sure that the approximation is good if the block is low rank. Also, in this way, the approximation for $\|R_p\|_F$ is always valid and the admissibility criterion always works. However, the Fully Pivoted ACA requires the search for the maximum element of a matrix, which is an $O(N^2)$ operation. This makes it too expensive for practical use.

Instead, the Partially Pivoted ACA shows potential for practical implementations [10]. This algorithm works by choosing an arbitrary starting row i_1. j_k is then selected such that $R_{k-1}(i_k, j_k)$ is the largest element for the row $R_{k-1}(i_k,:)$. Then the next row i_{k+1} is selected such that $R_k(i_{k+1}, j_k)$ is the largest element for the column $R_k(:, j_k)$. In the Partially Pivoted ACA, it could happen that no pivot column or row can be chosen because $R_{k-1}(i_k,:)$ or $R_k(:, j_k)$ has available elements all equal to zero. In this case, a different non-zero row or column should be selected. Because the Partially Pivoted ACA only needs to search through a column or a row, it has computational complexity $O(N)$.

5.1.3 Matrix Vector Multiplication using the H-matrix Structure

The use of Lanzcos Bidiagonalization or ACA allows each block to be checked if it is admissible or inadmissible. This test for admissibility is applied from level $l = 2$ onwards. Pictorially, the result of this operation can look like Figure 7. Black boxes represents blocks that become admissible at that level, grey boxes indicates blocks that was already admissible in previous levels, and white boxes are blocks that are still inadmissible at that level.

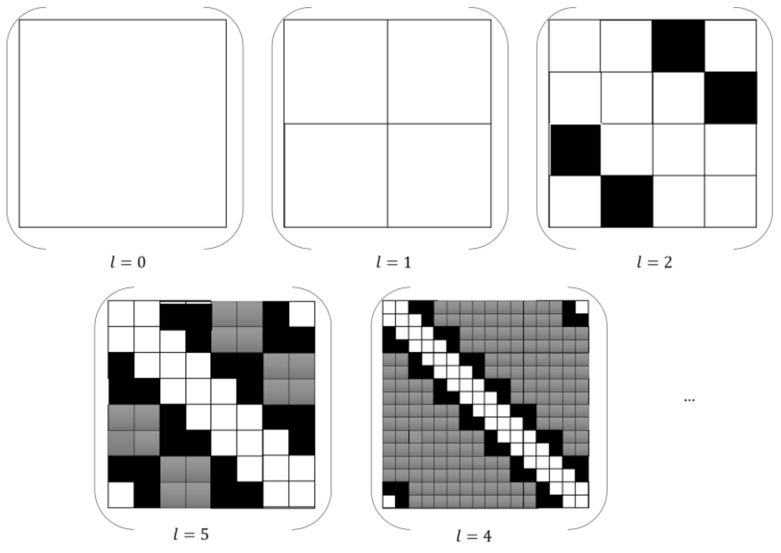


Figure 7 The hierarchically split matrix A

Define now M_l as the matrix made up of admissible blocks at that level, $\widetilde{M_l}$ is the matrix with the same structure as M_l, but with each blocks approximated by their low rank approximation as given in Equation 1 or 2. N_{levels} is the matrix made up of all the inadmissible blocks at the finest level. Refer to Figure 8 for an illustration of the sparsity structure of M_l and N_{levels} based on Figure 7.

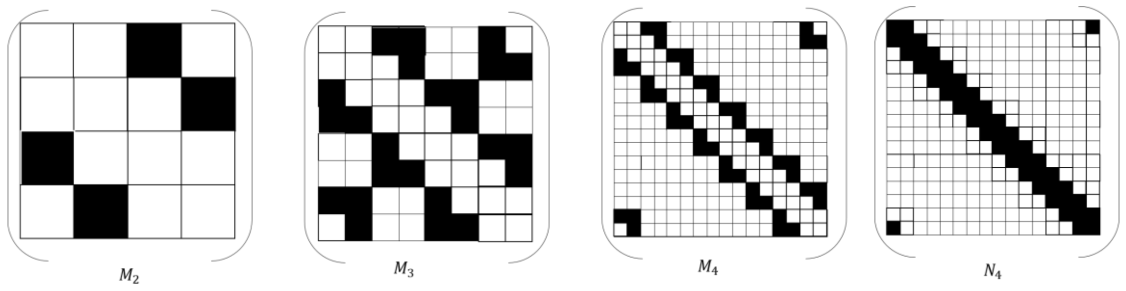


Figure 8 Sparsity structure of Matrix M_l and N_{levels} for a hierarchical matrix shown in Figure 7. White boxes indicates blocks which have elements all zeros.

With this, an approximation for the matrix vector product Ax can be written as:

$$Ax = \sum_{l=2}^{levels} M_l x + N_{levels} x \approx \sum_{l=2}^{levels} \widetilde{M}_l x + N_{levels} x$$

Consider $\widetilde{M}_l x$. First partition the vector x with respect to the column partition of $\widetilde{M}_l$, and the vector $\widetilde{M}_l x$ with respect to the row partition of $\widetilde{M}_l$. Each partition of the vector x is represented by x_τ, and each partition of vector $\widetilde{M}_l x$ by $\left(\widetilde{M}_l x\right)_\sigma$.

If Equation 2 is used, the vector $\widetilde{M}_l x$ can be written as:

$$\widetilde{M}_l x = \begin{bmatrix} \left(\widetilde{M}_l x\right)_1 \\ \vdots \\ \left(\widetilde{M}_l x\right)_\sigma \\ \vdots \\ \left(\widetilde{M}_l x\right)_{2^l} \end{bmatrix} = \begin{bmatrix} \sum_{\tau=1}^{2^l} U_{1,\tau} B_{1,\tau} V_{1,\tau}^T \, x_\tau \\ \vdots \\ \sum_{\tau=1}^{2^l} U_{2^l,\tau} B_{2^l,\tau} V_{2^l,\tau}^T \, x_\tau \end{bmatrix}$$

For Equation 3, the following is used instead:

$$\widetilde{M}_l x = \begin{bmatrix} \left(\widetilde{M}_l x\right)_1 \\ \vdots \\ \left(\widetilde{M}_l x\right)_\sigma \\ \vdots \\ \left(\widetilde{M}_l x\right)_{2^l} \end{bmatrix} = \begin{bmatrix} \sum_{\tau=1}^{2^l} U_{1,\tau} V_{1,\tau}^T \, x_\tau \\ \vdots \\ \sum_{\tau=1}^{2^l} U_{2^l,\tau} V_{2^l,\tau}^T \, x_\tau \end{bmatrix}$$

5.2 IMPLEMENTATION DETAILS

To implement the hierarchical method in Fortran, a data structure must first be defined to store the H-matrix. This is named the *hierarchy_class* object. With the data structure defined, subroutines that are required to handle the H-matrix must be formulated. Subsequently, the *hierarchy_class* object must be integrated with the current solver codes such that the dense matvec operations are replaced by hierarchical matvec. This section discusses all these implementations, and ends with some comments on the implementation issues faced.

5.2.1 Data structure of a Hierarchical Matrix (H-matrix)

A data structure needs to be defined to store the H-matrix. In this section, it is assumed that Equation 2 is used. The data structure can be adapted for Equation 1 with minor modifications.

The data structure for the H-matrix is given in Figure 9 below.

```
type, public :: hierarchy

        ! List of arrays
        ! ---------------------------------------------
        DATATYPE1,           allocatable, dimension(:) :: U
        DATATYPE1,           allocatable, dimension(:) :: V
        integer(kind=SHORT), allocatable, dimension(:) :: index_con
        integer(kind=SHORT), allocatable, dimension(:) :: index_lvl_con
        integer(kind=SHORT), allocatable, dimension(:) :: adm_row
        integer(kind=SHORT), allocatable, dimension(:) :: adm_col
        integer(kind=SHORT), allocatable, dimension(:) :: inadm_row
        integer(kind=SHORT), allocatable, dimension(:) :: inadm_col
        integer(kind=SHORT), allocatable, dimension(:) :: block_lvl_con
        integer(kind=SHORT), allocatable, dimension(:) :: Ublockpos_con
        integer(kind=SHORT), allocatable, dimension(:) :: Vblockpos_con


        ! List of variables
        ! ---------------------------------------------
        integer(kind=SHORT) :: p !rank of admissible block
        integer(kind=SHORT) :: b !minimum block size
        integer(kind=SHORT) :: N !size of original array
        integer(kind=SHORT) :: levels
        real                :: tol !tolerance of low rank approximation

    end type hierarchy
```

Figure 9 Data structure of H-matrix

Some definitions need to be specified here before delving into the details.

Firstly, the *global block number* is defined as the block number of a block with respect to all blocks at all levels, starting from level 2. The global block number starts from level 2 because the low rank approximation of blocks begin at level 2. Figure 10 below depicts the concept of global block number.

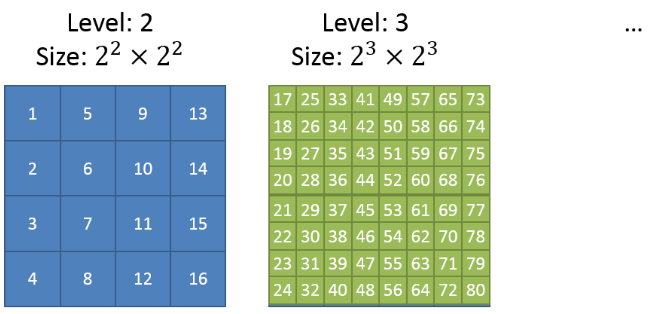
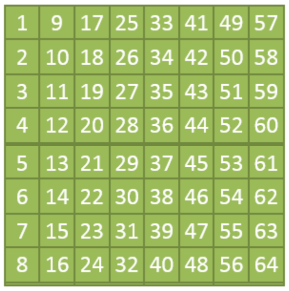


Figure 10 Definition of global block number. The first block at level 3 for instance have global block number 17

On the other hand, the *local block number* is defined as the block number of the block with respect to the blocks at that level. For example, the local block number of the blocks at level 3 are shown in Figure 11 below.



1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

Figure 11 Local block number at level 3

Next, the *1D global indices* at level l is the set of row or column indices that defines how the row or column is divided at that level. It can be assumed that the original matrix is square, hence, the row and column division is the same. Taking the example of the FATIMA_7894 matrix, the hierarchy division of the matrix from level 1 to 3 is shown in Figure 12. The 1D global indices for this matrix at $l = 2$ is $\{1, 1975, 3948, 5922, 7895\}$. Note that the last element stored is purely for computational reasons. Similarly, at $l = 3$, the 1D global indices are $\{1, 988, 1975, 2962, 3948, 4935, 5922, 6909, 7895\}$.

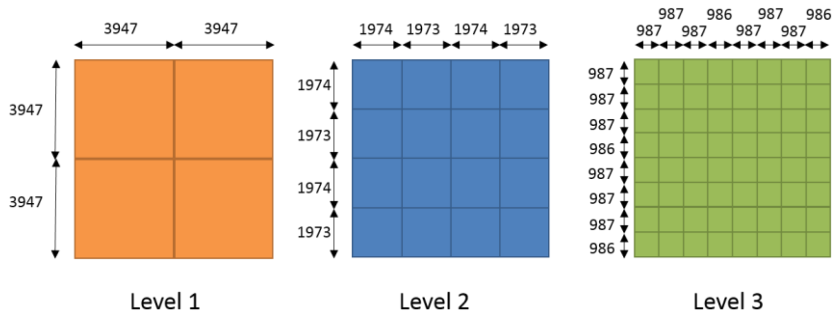


Figure 12 Hierarchical division of FATIMA_7894 from level 1 to 3

With these defined, each element in the data structure for H-matrix, as shown in Figure 9, is elaborated in the subsections below.

5.2.1.1 U and V

The U and V 1D arrays specified in the data structure of H-matrix are used to store the elements from the low rank approximation of admissible blocks. Before calling Lanzcos Bidiagonalization or ACA to hierarchically split the matrix, there is no way of knowing which blocks are admissible. Hence, memory is be allocated for $U_{\sigma,\tau}$ and $V_{\sigma,\tau}$ for all $2^l \times 2^l$ blocks at each level l. Let the finest recursion level be named $levels$. Then the size of the 1D array required for U or V is $Np(2^{levels+1} - 4)$, since the total number of elements can be computed as follows:

$$\sum_{l=2}^{levels} N \times 2^l \times p = Np\left(\frac{(1 - 2^{levels+1})}{1-2} - 2^0 - 2^1\right) = Np(2^{levels+1} - 4)$$

The 1D arrays store the low rank approximation of each block according to the global block number. That is, the matrix block corresponding to global block 1 is stored first in a column major format, followed by the matrix block corresponding to global block 2 and so on. This is illustrated in Figure 13 below:

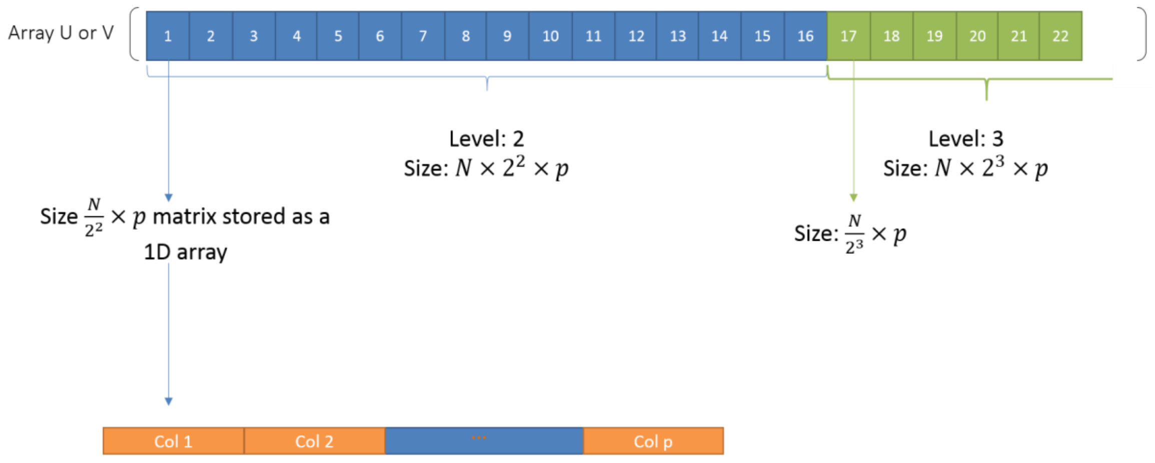

Array U or V
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
Level: 2
Size: $N \times 2^2 \times p$
Level: 3
Size: $N \times 2^3 \times p$
Size $\frac{N}{2^2} \times p$ matrix stored as a 1D array
Size: $\frac{N}{2^3} \times p$
Col 1
Col 2
...
Col p

Figure 13 Storage of admissible U or V matrix

Blocks that are admissible have the respective $U_{\sigma,\tau}$ and $V_{\sigma,\tau}$ elements computed either by ACA or Lanzcos Bidiagonalization (in which case a third 1D array, B, is required). The resulting $U_{\sigma,\tau}$ and $V_{\sigma,\tau}$ are then stored in their respective position in the 1D array. Blocks that are inadmissible have all zero elements in the 1D arrays.

5.2.1.2 *adm_row and adm_col*

It is helpful to know which blocks are admissible at each level without the need to sieve through the 1D arrays of U or V for non-zero blocks. Therefore, a data structure is required to store information regarding which blocks are admissible. Two 1D arrays named adm_row and adm_col are used to store this information. Each of these arrays are of size $\frac{4^{levels+1}-1}{3} - 5$, since

$$\sum_{l=2}^{levels} 2^l \times 2^l = \frac{4^{levels+1}-1}{3} - 4^0 - 4^1 = \frac{4^{levels+1}-1}{3} - 5$$

Each element of the 1D array adm_row or adm_col corresponds to a global block. This means that the first element of the 1D array corresponds to global block 1, the second corresponds to global block 2, and so on. If the block is admissible, the local block row or local block column number is stored. Else, 0 is stored. Figure 14 below shows how adm_row and adm_col looks like for the FATIMA_7894 matrix with $levels = 3, p = 50$. The corresponding admissible blocks are depicted in Figure 15.

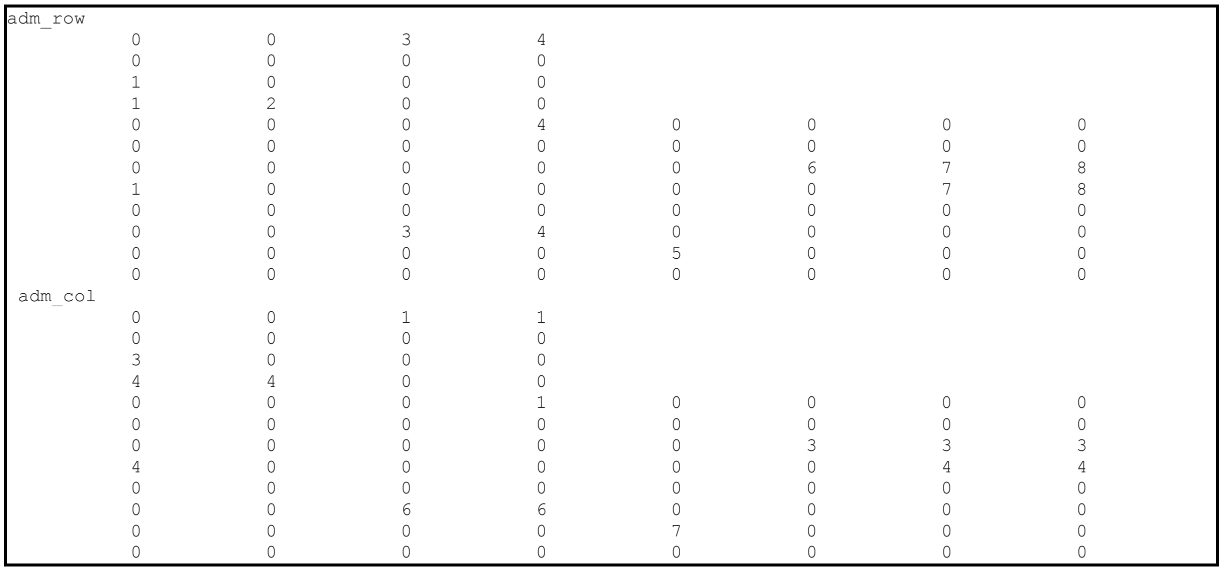

```
adm_row
            0            0            3            4
            0            0            0            0
            1            0            0            0
            1            2            0            0
            0            0            0            4            0            0            0            0
            0            0            0            0            0            0            0            0
            0            0            0            0            0            6            7            8
            1            0            0            0            0            0            7            8
            0            0            0            0            0            0            0            0
            0            0            3            4            0            0            0            0
            0            0            0            0            5            0            0            0
            0            0            0            0            0            0            0            0
 adm_col
            0            0            1            1
            0            0            0            0
            3            0            0            0
            4            4            0            0
            0            0            0            1            0            0            0            0
            0            0            0            0            0            0            0            0
            0            0            0            0            0            3            3            3
            4            0            0            0            0            0            4            4
            0            0            0            0            0            0            0            0
            0            0            6            6            0            0            0            0
            0            0            0            0            7            0            0            0
            0            0            0            0            0            0            0            0
```

Figure 14 Example of how "adm_row" and "adm_col" looks like for FATIMA_7894 with levels = 3 , p = 50

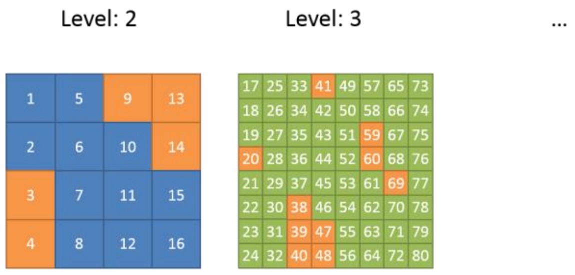
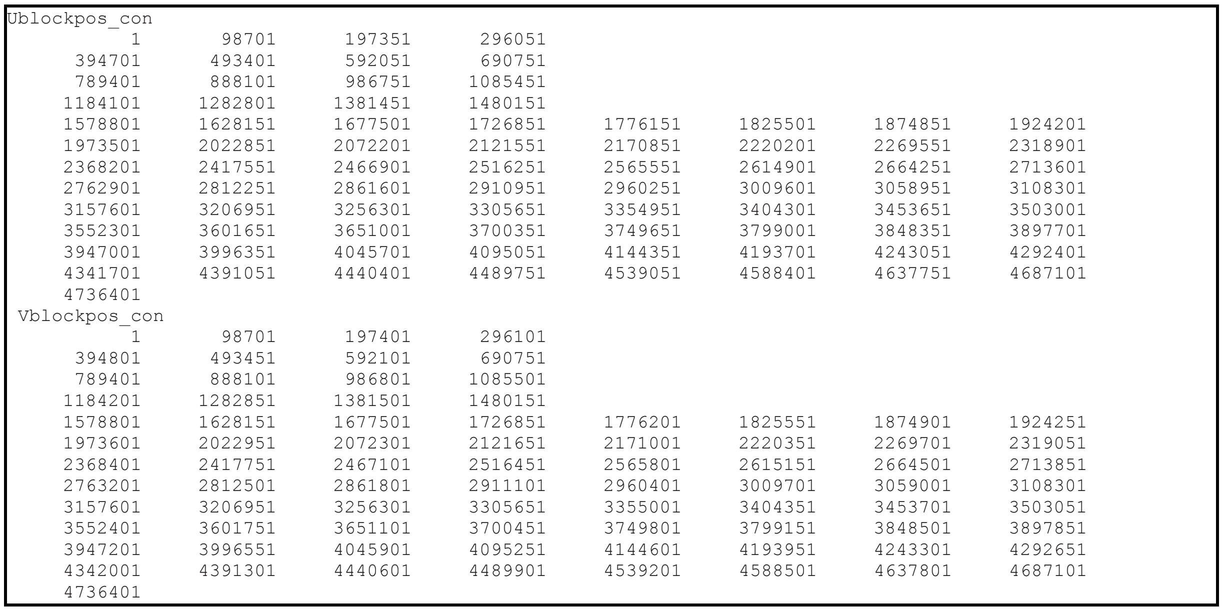


Figure 15 Corresponding admissible blocks (in orange) of the FATIMA_7894 matrix with reference to Figure 14

5.2.1.3 $\boldsymbol{Ublockpos\_con}$ and $\boldsymbol{Vblockpos\_con}$

As a link between the global block number and the U/V arrays, two 1D arrays are defined to store the index of the first element in the U/V arrays corresponding to every global block. This allows the easy access of the required $U_{\sigma,\tau}$ or $V_{\sigma,\tau}$ matrix blocks. These are known as $Ublockpos\_con$ and $Vblockpos\_con$. For instance, global block 1 always has its first element in the U and V container retrieved using $U(1)$ and $V(1)$. Global block 2 will have its first elements retrieved using $U\left(1+\left\lceil\frac{N}{2^2}\right\rceil\times p\right)$ and $V\left(1+\left\lceil\frac{N}{2^2}\right\rceil\times p\right)$. The index used to access the first element of every global block is stored in $Ublockpos\_con$ and $Vblockpos\_con$.

An example of how $Ublockpos\_con$ and $Vblockpos\_con$ look like, again using the example of FATIMA_7894 with $levels = 3$, is shown below:



```
Ublockpos_con
          1       98701      197351      296051
     394701      493401      592051      690751
     789401      888101      986751     1085451
    1184101     1282801     1381451     1480151
    1578801     1628151     1677501     1726851     1776151     1825501     1874851     1924201
    1973501     2022851     2072201     2121551     2170851     2220201     2269551     2318901
    2368201     2417551     2466901     2516251     2565551     2614901     2664251     2713601
    2762901     2812251     2861601     2910951     2960251     3009601     3058951     3108301
    3157601     3206951     3256301     3305651     3354951     3404301     3453651     3503001
    3552301     3601651     3651001     3700351     3749651     3799001     3848351     3897701
    3947001     3996351     4045701     4095051     4144351     4193701     4243051     4292401
    4341701     4391051     4440401     4489751     4539051     4588401     4637751     4687101
    4736401
 Vblockpos_con
          1       98701      197401      296101
     394801      493451      592101      690751
     789401      888101      986801     1085501
    1184201     1282851     1381501     1480151
    1578801     1628151     1677501     1726851     1776201     1825551     1874901     1924251
    1973601     2022951     2072301     2121651     2171001     2220351     2269701     2319051
    2368401     2417751     2467101     2516451     2565801     2615151     2664501     2713851
    2763201     2812501     2861801     2911101     2960401     3009701     3059001     3108301
    3157601     3206951     3256301     3305651     3355001     3404351     3453701     3503051
    3552401     3601751     3651101     3700451     3749801     3799151     3848501     3897851
    3947201     3996551     4045901     4095251     4144601     4193951     4243301     4292651
    4342001     4391301     4440601     4489901     4539201     4588501     4637801     4687101
    4736401
```

Figure 16 Example of how "Ublockpos_con" and "Vblockpos_con" looks like for FATIMA_7894 with levels = 3

5.2.1.4 $\boldsymbol{block\_lvl\_con}$

To convert from the local block numbers to global block numbers, repeated computations involving powers are required. To reduce this need, an auxiliary array $block\_lvl\_con$ with size $(levels - 1)$ is defined. This array simply stores the global block number of the first block at every level from level 2 till $levels$. Thus, $block\_lvl\_con$ consist of elements $\{1, 1 + (2^2 \times 2^2), 1 + (2^3 \times 2^3), \dots, 1 + (2^{levels-1} \times 2^{levels-1})\}$.

5.2.1.5 Example to illustrate how to retrieve a block $U_{\sigma,\tau}$/$V_{\sigma,\tau}$

With these in place, it is now easy to retrieve the low rank approximation of an admissible block. Assume that the level l and the local block row and column numbers i, j are known. The global block number, $block\_number$, can be easily computed with the help of $block\_lvl\_con$. The $U_{\sigma,\tau}$ and $V_{\sigma,\tau}$ matrix corresponding to this block can be retrieved from the 1D arrays U and V with the help of $Ublockpos\_con$ and $Vblockpos\_con$. The code fragment below illustrates how $U_{\sigma,\tau}$ and $V_{\sigma,\tau}$ corresponding to a block defined by l, i and j can be retrieved and called in a function:

```
!Read out the global block number of the first block of this level
! ----------------------------------------------
start_block_lvl = this%block_lvl_con(level-1)

!Compute the global block number
! ----------------------------------------------
block_number = start_block_lvl + i + (j-1)*2**l-1

!Retrieve the start and end position of Uσ,τ and Vσ,τ
! ----------------------------------------------
start_U_block = this%Ublockpos_con(block_number)
end_U_block = this%Ublockpos_con(block_number+1)-1

start_V_block = this%Vblockpos_con(block_number)
end_V_block = this%Vblockpos_con(block_number+1)-1

!Do work with this admissible block
! ----------------------------------------------
call sample_function(this%U(start_U_block:end_U_block), &
          this%V(start_V_block:end_V_block),…)
```

Figure 17 Code fragment depicting how to retrieve $U_{\sigma,\tau}$ and $V_{\sigma,\tau}$ for a particular block

5.2.1.6 *inadm_row* and *inadm_col*

Notice that the data structure of the H-matrix does not include an array for the inadmissible matrix N_{levels} (Refer to Figure 9). This is because the full matrix A is already stored, and the matrix elements in N_{levels} can be extracted from A. For this purpose, two 1D arrays named $inadm\_row$ and $inadm\_col$ are created. In this case, the two arrays are of size $2^{levels} \times 2^{levels}$, since inadmissible blocks only occur at the finest recursion level. Each element of the array corresponds to a local block at $l = levels$. If the block is inadmissible, the block row number and column number are stored. Else, 0 is stored. Refer to Figure 18 for an example of how these arrays look like.

```
inadm_row
           1           2           3           0           0           0           0           0
           1           2           3           4           0           0           0           0
           1           2           3           4           5           0           0           0
           0           2           3           4           5           6           0           0
           0           0           3           4           5           6           7           8
           0           0           0           0           5           6           7           8
           0           0           0           0           0           6           7           8
           0           0           0           0           5           6           7           8
 inadm_col
           1           1           1           0           0           0           0           0
           2           2           2           2           0           0           0           0
           3           3           3           3           3           0           0           0
           0           4           4           4           4           4           0           0
           0           0           5           5           5           5           5           5
           0           0           0           0           6           6           6           6
           0           0           0           0           0           7           7           7
           0           0           0           0           8           8           8           8
```

Figure 18 Example of how "inadm_row" and "inadm_col" looks like for FATIMA_7894 with levels = 3 , p = 50

5.2.1.7 $index\_con$

For the easy retrieval of inadmissible matrix blocks, 1D global indices can be used to specify the exact indices in the system matrix A corresponding to an inadmissible block. An auxiliary 1D array is defined to store the 1D global indices at every level from level 1 till $levels$. This is named $index\_con$, with size $levels + 2^{levels+1} - 2$, since

$$\sum_{l=1}^{levels} 2^l + 1 = levels + 2^{levels+1} - 2$$

As an illustration, the $index\_con$ array for the FATIMA_7894 matrix, with $levels = 3$, looks like:

```
index_con
         1        3948        7895
         1        1975        3948        5922        7895
         1         988        1975        2962        3948        4935        5922        6909        7895
```

Figure 19 Example of how "index_con" looks like for FATIMA_7894 with levels = 3

Thus, given an inadmissible block at $l = 3$, with $i, j = 2{,}2$, the indices of this block in the system matrix A starts from (988,988).

5.2.1.8 $index\_lvl\_con$

To reduce the need to compute powers repeatedly, another auxiliary array of size $levels$ will be used to store the start position in the $index\_con$ array of the first block of every level. This array is called $index\_lvl\_con$. Its elements will simply be $\{1, 1 + (2^1 + 1), 1 + (2^2 + 1), \dots, 1 + (2^{levels-1} + 1)\}$.

5.2.1.9 Example to illustrate how to retrieve an inadmissible block in N_{levels}

Assume that the local block row i, local block column j, and the level l are known. Two steps are required to retrieve an inadmissible block. First, a 1D array is used to store the 1D global indices at level l. Then, the global indices corresponding to this block can be found by simply reading out the respective elements in this 1D array. With the range of indices corresponding to this block known, the relevant block from the matrix A can be extracted to perform work. The code fragment below illustrates how this can be done.

```
        !Read out the global indices of all blocks at this level
        index_global = this%index_con(this%index_lvl_con(l):
        this%index_lvl_con(l+1)-1)
        !Read out the global index of this block
        i_start_global = index_global(block_start_i)
        i_end_global = index_global(block_start_i+1)-1
        j_start_global = index_global(block_start_j)
        j_end_global = index_global(block_start_j+1)-1
        !Do work with this inadmissible block
        ! ----------------------------------------------
        call sample_function(A(i_start_local:i_end_local,
        j_start_local:j_end_local),…)
```

Figure 20 Code fragment depicting how to retrieve an inadmissible block

5.2.1.10 Other variables

Apart from the 1D arrays defined, some constants need to be defined:

- **p:** rank of admissible block
- **b:** minimum block size
- **N:** size of matrix A
- **levels:** Depth of recursion
- **tol:** tolerance we set to determine if a block is admissible or not in ACA

5.2.2 Subroutines

The subroutines required to handle H-matrix include:

- **Hierarchy_construct**: to allocate memory and initialize the *hierarchy_class* object
- **Hierarchy_destruct**: to clean up memory
- **Hierarchy_split**: to split a matrix hierarchically and store the relevant information in the H-matrix
- **ACA**: contains the ACA algorithm.Called within hierarchy_split to check if a block is admissible and return $U_{\sigma,\tau}$ and $V_{\sigma,\tau}$

- **Lanzcos_bidiag**: contains the Lanzcos Bidiagonalization algorithm. Called within hierarchy_split to check if a block is admissible and return $U_{\sigma,\tau}$ and $V_{\sigma,\tau}$
- **Hie_matvec_A**: performs hierarchical matvec

The implementation details for the subroutines for hierarchy_split, ACA, lanzcos_bidiag and hie_matvec_A are elaborated in this section.

5.2.2.1 Hierarchy_split

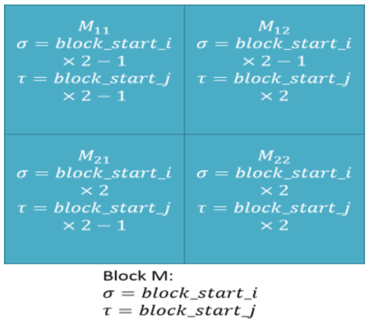
The hierarchy_split subroutine is a recursive subroutine that takes the following argument as input:

- The matrix block to be split hierarchically: $M(:,:)$
- The level at which M is at: $level$
- The current row and column block number of M: $block\_start\_i, block\_start\_j$

It returns a *hierarchy_class* object as output.

The pseudo-code for hierarchy_split subroutine is given below:

```
1)  Compute the 4 block row and column numbers corresponding to the 4 sub-blocks of
    M at the next level. This can be obtained easily from $block\_start\_i$ and $block\_start\_j$
    as depicted in the figure below.
```


M_{11}
$\sigma = block\_start\_i \times 2 - 1$
$\tau = block\_start\_j \times 2 - 1$
M_{12}
$\sigma = block\_start\_i \times 2 - 1$
$\tau = block\_start\_j \times 2$
M_{21}
$\sigma = block\_start\_i \times 2$
$\tau = block\_start\_j \times 2 - 1$
M_{22}
$\sigma = block\_start\_i \times 2$
$\tau = block\_start\_j \times 2$
Block M:
$\sigma = block\_start\_i$
$\tau = block\_start\_j$

```
2)  If $l = 0$
    a)  Do for blocks 1 to 4
        i)  Recursively call hierarchy_split
3)  Else
    a)  Do for blocks 1 to 4
        i)  Call ACA or Lanzcos_Bidiag to determine if block is admissible
            (1) If admissible, store $U_{\sigma,\tau}$ and $V_{\sigma,\tau}$ in the 1D array $U$ and $V$ and update
                $adm\_row$ and $adm\_col$
            (2) Else
                (a) If $l$ =finest recursion level, update $inadm\_row$ and $inadm\_col$
                (b) Else, recursively call hierarchy_split
```

Algorithm 1 Hierarchy_split

5.2.2.2 ACA

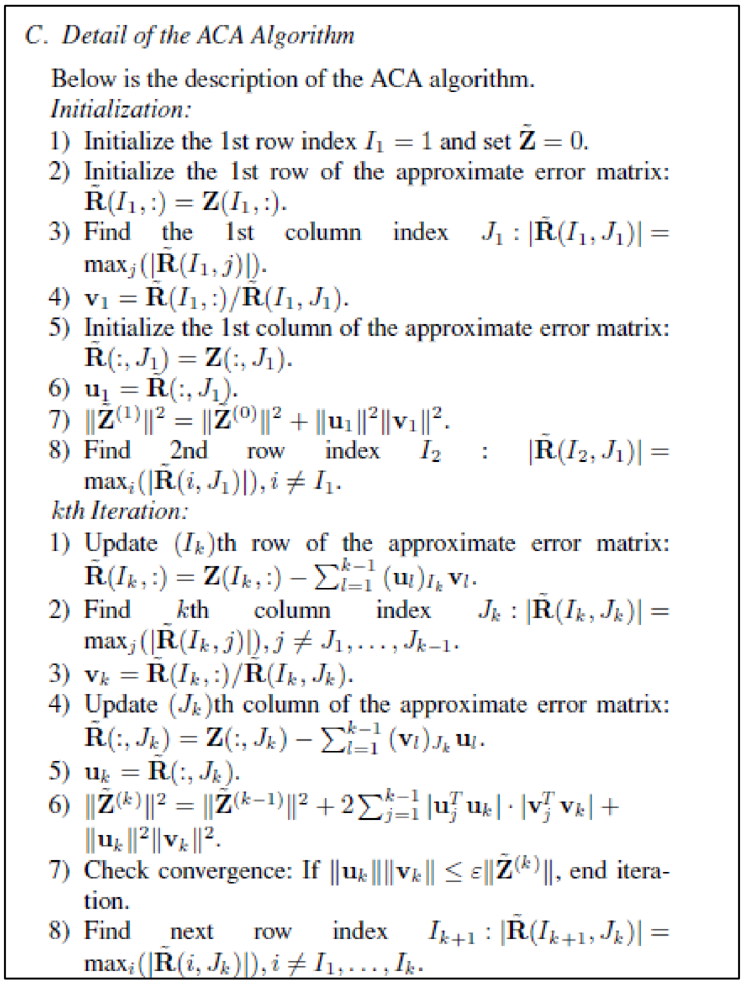
The input to subroutine ACA consist of:

- The matrix block in which ACA is called: $M(:,:)$
- The rank of each low rank approximated block: p
- The row and column size of the matrix block M: $size\_U\_row$ and $size\_V\_row$
- The tolerance below which the block is deemed admissible: tol

The output of the subroutine consist of:

- A flag which is 0 if the block is inadmissible, and 1 if admissible: $admissible$
- The matrix $U_{\sigma,\tau}$ and $V_{\sigma,\tau}$: $U(:), V(:)$

The pseudo-code for ACA routine is given below, extracted from [11]:



C. Detail of the ACA Algorithm

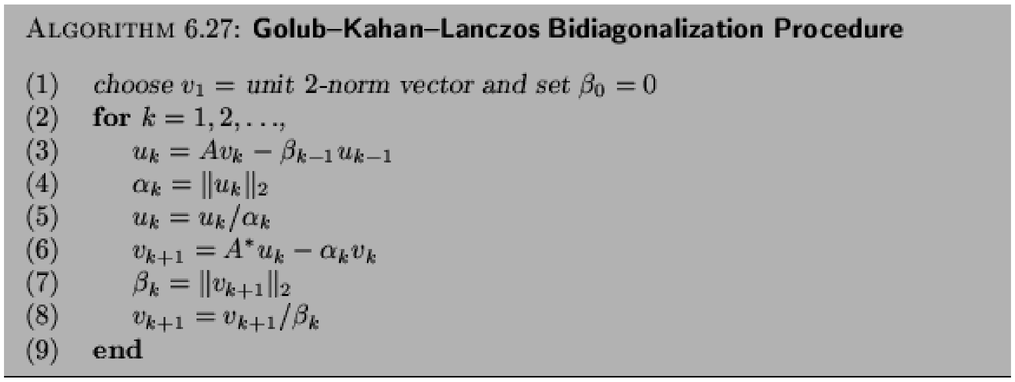
Below is the description of the ACA algorithm.

```
Initialization:
1) Initialize the 1st row index $I_1 = 1$ and set $\tilde{\mathbf{Z}} = 0$.
2) Initialize the 1st row of the approximate error matrix: $\tilde{\mathbf{R}}(I_1,:) = \mathbf{Z}(I_1,:)$.
3) Find the 1st column index $J_1 : |\tilde{\mathbf{R}}(I_1, J_1)| = \max_j(|\tilde{\mathbf{R}}(I_1, j)|)$.
4) $\mathbf{v}_1 = \tilde{\mathbf{R}}(I_1,:)/\tilde{\mathbf{R}}(I_1, J_1)$.
5) Initialize the 1st column of the approximate error matrix: $\tilde{\mathbf{R}}(:, J_1) = \mathbf{Z}(:, J_1)$.
6) $\mathbf{u}_1 = \tilde{\mathbf{R}}(:, J_1)$.
7) $\|\tilde{\mathbf{Z}}^{(1)}\|^2 = \|\tilde{\mathbf{Z}}^{(0)}\|^2 + \|\mathbf{u}_1\|^2\|\mathbf{v}_1\|^2$.
8) Find 2nd row index $I_2 : |\tilde{\mathbf{R}}(I_2, J_1)| = \max_i(|\tilde{\mathbf{R}}(i, J_1)|), i \neq I_1$.
kth Iteration:
1) Update $(I_k)$th row of the approximate error matrix: $\tilde{\mathbf{R}}(I_k,:) = \mathbf{Z}(I_k,:) - \sum_{l=1}^{k-1}(\mathbf{u}_l)_{I_k}\mathbf{v}_l$.
2) Find $k$th column index $J_k : |\tilde{\mathbf{R}}(I_k, J_k)| = \max_j(|\tilde{\mathbf{R}}(I_k, j)|), j \neq J_1, \ldots, J_{k-1}$.
3) $\mathbf{v}_k = \tilde{\mathbf{R}}(I_k,:)/\tilde{\mathbf{R}}(I_k, J_k)$.
4) Update $(J_k)$th column of the approximate error matrix: $\tilde{\mathbf{R}}(:, J_k) = \mathbf{Z}(:, J_k) - \sum_{l=1}^{k-1}(\mathbf{v}_l)_{J_k}\mathbf{u}_l$.
5) $\mathbf{u}_k = \tilde{\mathbf{R}}(:, J_k)$.
6) $\|\tilde{\mathbf{Z}}^{(k)}\|^2 = \|\tilde{\mathbf{Z}}^{(k-1)}\|^2 + 2\sum_{j=1}^{k-1}|\mathbf{u}_j^T\mathbf{u}_k| \cdot |\mathbf{v}_j^T\mathbf{v}_k| + \|\mathbf{u}_k\|^2\|\mathbf{v}_k\|^2$.
7) Check convergence: If $\|\mathbf{u}_k\|\|\mathbf{v}_k\| \leq \varepsilon\|\tilde{\mathbf{Z}}^{(k)}\|$, end iteration.
8) Find next row index $I_{k+1} : |\tilde{\mathbf{R}}(I_{k+1}, J_k)| = \max_i(|\tilde{\mathbf{R}}(i, J_k)|), i \neq I_1, \ldots, I_k$.
```

Algorithm 2 Pseudo-code for ACA [11]

5.2.2.3 Lanzcos_Bidiag

The input and output arguments for the Lanzcos_Bidiag routine is the same as ACA. The pseudo-code is given below, adapted from [8].



ALGORITHM 6.27: **Golub–Kahan–Lanczos Bidiagonalization Procedure**

```
(1)  choose $v_1$ = unit 2-norm vector and set $\beta_0 = 0$
(2)  for $k = 1, 2, \ldots,$
(3)      $u_k = Av_k - \beta_{k-1} u_{k-1}$
(4)      $\alpha_k = \|u_k\|_2$
(5)      $u_k = u_k / \alpha_k$
(6)      $v_{k+1} = A^* u_k - \alpha_k v_k$
(7)      $\beta_k = \|v_{k+1}\|_2$
(8)      $v_{k+1} = v_{k+1} / \beta_k$
(9)  end
```

Algorithm 3 Lanzcos_Bidiag [8]

The only changes made to the pseudo-code to adapt to this application is the loop-stopping criteria. The loop is stopped when $k > p$ or $\beta_k < tol$.

5.2.2.4 Hie_matvec_A

This is a recursive subroutine. The input to this subroutine are:

- A hierarchy_type object that contains the hierarchical form of A: hie
- The original system matrix: $A(:,:)$
- The bock row and block column number at which the multiplication is currently carried out: $block\_start\_i$ and $block\_start\_j$
- A flag to indicate if the current block is admissible: adm_A
- The current level: l
- The vector to be multiplied: $M\_in(:)$

The output to this subroutine is the result of the matrix-vector multiplication $M\_out(:)$.

To illustrate how this algorithm works, consider a block of the hierarchical form of A that is not at its finest level, and is not admissible. In this case, matrix vector multiplication can be performed using the following equation:

$$M\_out = \begin{bmatrix} M\_out_1 \\ M\_out_2 \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} M\_in_1 \\ M\_in_2 \end{bmatrix} = \begin{bmatrix} A_{11}M\_in_1 + A_{12}M\_in_2 \\ A_{21}M\_in_1 + A_{22}M\_in_2 \end{bmatrix}$$

Equation 10 Hierarchical matrix vector multiplication

With Equation 10, one can construct the algorithm. If A_{ij} is a block at the finest level, the individual matrix-vector multiplication can be carried out in either of the following ways:

1. If A_{ij} is admissible, then its low rank approximation is to be used for the matrix-vector multiplication. $A_{ij}\ M\_in_j = U_{ij}V_{ij}^T\ M\_in_j$. This is known as low rank matrix-vector multiplication
2. Otherwise, dense matrix vector multiplication is carried out instead.

Thus we have the following pseudo-code for hie_matvec_A:

```
1)  If $l$ = finest recursion level
    a)  If $adm\_A$ is inadmissible (0)
        i)  Dense matrix-vector multiplication is carried out
    b)  Else
        i)  Low rank matrix-vector multiplication is carried out
2)  Else
        i)  If $adm\_A$ is admissible (1)
            (1) low rank matrix-vector multiplication is carried out
        ii) Else
            (1) The block is split into 4 sub-blocks. The subroutine hie_matvec_A is
                recursively called on each sub-blocks, with the results compiled
                according to Equation 10.
```

Algorithm 4 Pseudo-code for hie_matvec_A

5.2.3 Adapting the solver to use hie_matvec_A instead of dense matvec

Both GMRES and IDR(s) solver calls one dense matvec per iteration. Hence, one just have to replace this matvec subroutine with hie_matvec_A in the solver routine. To do this, the data structure of the solver_class object is edited to include a *hierarchy_class* object. Before calling the solver, the *hierarchy_class* object is constructed and updated to contain the hierarchical form of the matrix.

5.2.4 Implementation issues and fixes

5.2.4.1 Skipping ACA or Lanzcos_Bidiag for blocks next to the main diagonal

It can be inferred from the theory of BEM [2] and verified from experience that the blocks near the main diagonal is not admissible. As such, instead of calling the ACA or Lanzcos_Bidiag routine for these blocks, it is cheaper to just assume the blocks are inadmissible. This is done in the hierarchy_split routine as shown in the code fragment below:

```
!if block near main diagonal, skip ACA check
if ( ABS(block_no_row(i) - block_no_col(j)) <= 1) then
    admissible =0
else
    call ACA(M(i_start_local:i_end_local, j_start_local:j_end_local), &
                         this%p, &
                         this%U(start_U_block:end_U_block), &
                         this%V(start_V_block:end_V_block), &
                         i_end_global - i_start_global+1, &
                         j_end_global - j_start_global+1, &
                         this%tol,&
                         admissible)

end if
```

Figure 21 Code fragment from hierarchy_split showing how the blocks next to the main diagonal are skipped

5.2.4.2 *Issues in ACA sub-routine*

Although the computational complexity of the partially pivoted ACA is much needed, it proves to have an issue with the admissibility criterion for certain matrices. Specifically, the approximation of $\|R_p\|_F$ with $\|u_p v_p\|_F$. In our case, the Steadycav matrices and the Passcal matrix proves to be problematic with partially pivoted ACA.

Focusing first on the Steadycav matrices, all of these matrices have blocks whose sparsity structure is shown in Figure 22. White region indicates zero elements. It can be seen that the last few columns of this matrix are completely filled with zeros, with the exception of four elements. These four elements have values that are order of magnitudes larger than the rest of the matrix elements. Imagine now that the ACA subroutine is applied to one such matrix.

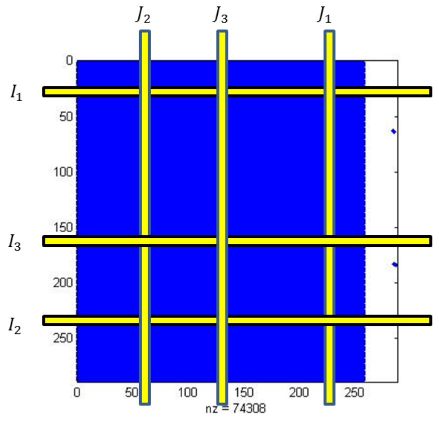


Figure 22 Sample matrix block from Steadycav1 matrix and an example of the selection of pivot columns and rows

A sample of how the first three pivot columns and rows can be selected is shown in yellow in the above figure. There is a high chance that the dominant elements will not be selected, and the admissibility criterion could be fulfilled, when in actual fact $\|R_p\|_F > \varepsilon \|M\|_F$.

To correct this, the Partially Pivoted ACA can be supplemented with an admissibility criterion based on the Completely Pivoted ACA. This will ensure that no blocks that are not admissible are wrongly classified as admissible. As this incur additional cost, it is only implemented for the Steadycav matrices.

Next, for the Passcal matrices, the sparsity structure for a block from the matrix can be as shown:

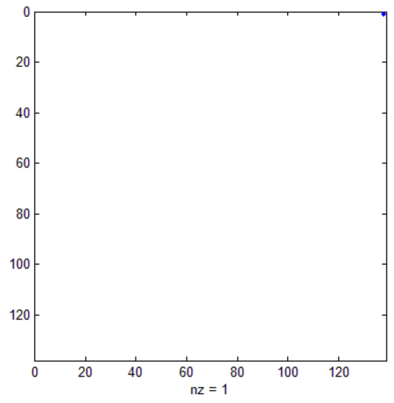


Figure 23 Sample matrix block from Passcal matrix

The entire matrix is almost zero, except for the element at the top right corner. In this case, partial pivoted ACA will fail because the algorithm as it is cannot detect that the entire matrix is almost zero, and division by zero will occur. To prevent this, the algorithm is updated so that it breaks out of the function whenever it detects that there are no more non-zero pivot elements to choose from.

5.3 RESULTS

In this section, the results for the hierarchical method are presented. First, the performance of Lanzcos Bidiagonalization and ACA are discussed. Next, the performance of the solver with hierarchical matvec is compared against the solver with dense matvec. This section then concludes with an assessment of this strategy.

5.3.1 Lanzcos Bidiagonalization versus ACA

Hierarchical splitting was performed using first Lanzcos Bidiagonalization, then ACA, for different levels of recursion $levels$ (defined by minimum block size allowed b) and rank of low rank approximation p. The results are tabulated in Table 7-10. The results for all the Steadycav matrices are similar, and therefore only the results for Steadycav 1 are shown.

The explanation of what each column in Table 7-10 represents are given below:

- **Time for normal matvec**: time for dense matrix-vector multiplication
- **Time split**: time taken to hierarchically split the matrix
- **Time for hie_matvec_A**: time for hierarchical matrix-vector multiplication
- $\|\boldsymbol{Ax_{hie}} - \boldsymbol{Ax_{exact}}\|$**:** 2-norm of the error defined by the result of the hierarchical matvec based on the result from dense matvec
- **Number of matvec to break even**: The number of matvec operations required to start gaining from the reduced time taken to do hierarchical matrix-vector multiplication. When the time taken for hierarchical matvec is more than or equal to that for dense matrix-vector multiplication, this is marked with "NA".

Looking at Table 8, 9 and 10, which depicts the result for Passcal, FATIMA_7894 and FATIMA_20493 respectively, it can be seen that ACA is the clear winner. The time required to perform hierarchical splitting by Lanzcos Bidiagonalization is simply too high. This can be seen also in Figure 24 below, which plots the time required to split using Lanzcos and ACA. An average of about 200 matvec operation is required to break even for Lanzcos Bidiagonalization, but for the ACA, the average is at about 30.

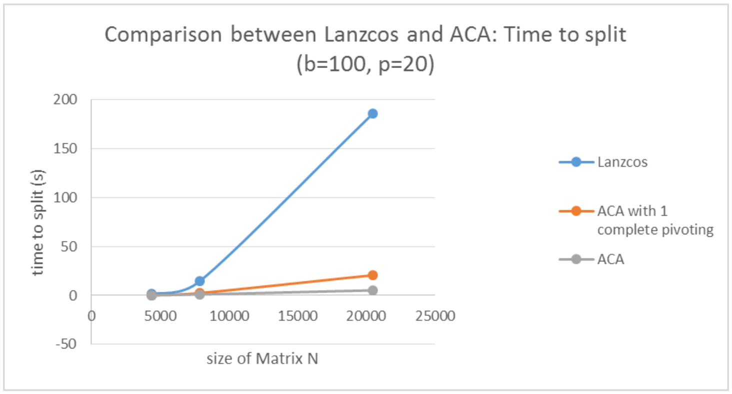


Figure 24 Comparison between Lanzcos Bidiagonalization an ACA on the time required to do hierarchical splitting

The graph in Figure 25 below compares the time required to do hierarchical matvec based on Lanzcos Bidiagonalization, ACA with an admissibility criterion based on complete pivoting and ACA (without complete pivoting in the admissibility criterion). It is clear that all three methods reduce the matvec time significantly as compared to the dense matvec operation. ACA has the lowest time required out of the three methods. This is because ACA is generally more relaxed in its admissibility criterion, and more blocks are deemed as admissible. Lanzcos Bidiagonalization comes in second, and ACA with one complete pivoting has the worst performance out of the three. This is expected, since ACA with an admissibility criterion based on complete pivoting causes less blocks to be admissible, and the time to do hierarchical matvec therefore increases.

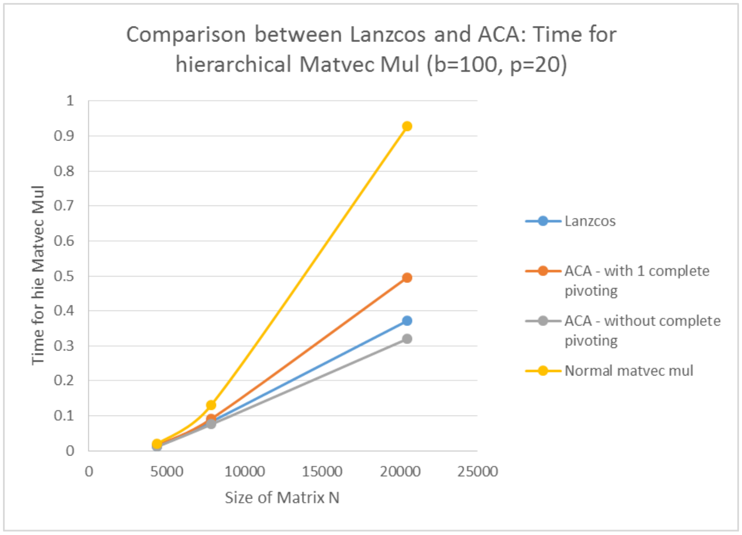


Figure 25 Comparison between Lanzcos Bidiagonalization an ACA on the time required to do hierarchical matvec

Referring to Table 7 for Steadycav1, it can be observed that ACA with one last iteration of complete pivoting does not work very well. Because of the final check with complete pivoting, too many blocks are inadmissible. This causes the hierarchical matvec time to be not competitive with the time required to do dense matrix-vector multiplication. Therefore, it can be concluded that this strategy is not suitable for the Steadycav type matrices.

In conclusion, ACA is preferred over Lanzcos Bidiagonalization due to its much lower time required to perform hierarchical splitting. Hence, ACA is used in hierarchy_split subroutine from this point on. It was assessed that this strategy is not applicable for the Steadycav matrices, and therefore, the next section does not address this class of matrices.

Steadycav1										
			Lanzcos Bidiagonalization				ACA with one last iteration of complete pivoting			
p	**b**	**time for normal matvec**	**Time_ split**	**time for hie_matvec_A**	$\|\|Ax_{hie} - Ax_{exact}\|\|$	**Number of matvec to break even**	**Time_ split**	**time for hie_matvec_A**	$\|\|Ax_{hie} - Ax_{exact}\|\|$	**Number of matvec to break even**
10	60	0.024	1.47	0.028	2.28E-05	NA	0.15	0.04	6.20E-06	NA
20		0.024	2.52	0.016	1.45E-05	316	0.43	0.036	1.60E-05	NA
30		0.024	3.28	0.016	1.13E-05	411	0.77	0.032	3.52E-05	NA
40		0.024	3.71	0.02	8.89E-06	928	1.23	0.032	2.46E-05	NA
10	100	0.024	1.11	0.024	1.24E-05	NA	0.072	0.028	4.24E-15	NA
20		0.024	2.07	0.016	9.62E-06	259	0.20	0.024	1.45E-06	NA
30		0.024	2.83	0.012	9.05E-06	236	0.37	0.024	1.24E-05	NA
40		0.024	3.14	0.016	6.85E-06	784	0.60	0.02	2.43E-05	150
10	200	0.024	0.78	0.024	2.96E-06	NA	0.028	0.024	4.23E-05	NA
20		0.024	1.51	0.02	4.60E-06	378	0.084	0.024	1.27E-05	NA
30		0.024	2.12	0.016	6.07E-06	266	0.17	0.024	1.13E-05	NA
40		0.024	2.57	0.016	5.26E-06	321	0.26	0.024	2.36E-05	NA
50		0.024	2.78	0.012	5.14E-06	231	0.38	0.024	1.16E-05	NA

Table 7 Comparison between Lanzcos Bidiagonalization and ACA (with one last iteration of complete pivoting) for Steadycav1. The lowest time or number of iterations recorded for each column are highlighted in green.

Passcal										
			Lanzcos Bidiagonalization				ACA			
p	**b**	**time for normal matvec**	**Time_ split**	**time for hie_matvec_A**	$\|\|Ax_{hie} - Ax_{exact}\|\|$	**Number of matvec to break even**	**Time_ split**	**time for hie_matvec_A**	$\|\|Ax_{hie} - Ax_{exact}\|\|$	**Number of matvec to break even**
10	60	0.02	0.988	0.012	9.87E-06	124	0.096	0.016	4.65E-05	24
20		0.02	1.408	0.012	5.12E-06	176	0.18	0.012	1.54E-05	23
30		0.02	1.772	0.008	5.15E-06	148	0.272	0.012	4.76E-05	34
10	100	0.02	0.836	0.012	7.04E-06	105	0.052	0.016	3.80E-06	13
20		0.02	1.304	0.012	3.35E-06	163	0.108	0.012	1.30E-05	14
30		0.02	1.724	0.012	3.98E-06	216	0.184	0.012	1.94E-05	23
10	200	0.02	0.676	0.02	4.09E-06	NA	0.02	0.02	2.71E-6	NA
20		0.02	0.964	0.016	2.65E-06	241	0.052	0.016	7.66E-6	13
30		0.02	1.304	0.012	3.36E-06	163	0.104	0.012	7.01E-6	13

Table 8 Comparison between Lanzcos Bidiagonalization and ACA for Passcal. The lowest time or number of iterations recorded for each column are highlighted in green.

FATIMA_7894										
			Lanzcos Bidiagonalization				ACA			
p	b	time for normal matvec	Time_split	time for hie_matvec_A	$\lVert Ax_{hie} - Ax_{exact} \rVert$	Number of matvec to break even	Time_split	time for hie_matvec_A	$\lVert Ax_{hie} - Ax_{exact} \rVert$	Number of matvec to break even
10	60	0.132	9.62	0.1	3.86E-07	301	1.05	0.12	1.23E-03	87
20		0.132	16.75	0.092	2.22E-07	419	1.89	0.088	1.42E-03	43
30		0.132	23.29	0.096	1.99E-07	647	3.06	0.084	1.24E-03	64
40		0.132	29.94	0.108	1.93E-07	1248	4.40	0.088	2.39E-05	100
10	100	0.132	8.16	0.104	3.65E-07	291	0.56	0.108	1.20E-03	23
20		0.132	14.52	0.084	2.06E-07	303	1.04	0.076	1.42E-03	19
30		0.132	20.42	0.08	1.92E-07	393	1.70	0.076	1.24E-03	30
40		0.132	26.51	0.084	1.84E-07	552	2.46	0.072	2.39E-04	41
10	200	0.132	6.39	0.116	2.91E-07	400	0.30	0.124	5.25E-07	37
20		0.132	11.36	0.092	1.50E-07	258	0.55	0.084	2.92E-04	11
30		0.132	15.87	0.084	1.37E-07	331	0.88	0.08	3.53E-04	17
40		0.132	20.24	0.084	1.40E-07	422	1.34	0.076	1.67E-04	24

Table 9 Comparison between Lanzcos Bidiagonalization and ACA for Fatima_7894. The lowest time or number of iterations recorded for each column are highlighted in green.

FATIMA_20493										
			Lanzcos Bidiagonalization				ACA			
p	b	time for normal matvec	Time_split	time for hie_matvec_A	$\lVert Ax_{hie} - Ax_{exact} \rVert$	Number of matvec to break even	Time_split	time for hie_matvec_A	$\lVert Ax_{hie} - Ax_{exact} \rVert$	Number of matvec to break even
10	70	0.932	111.68	0.30	1.19E-04	176	4.20	0.29	1.73E-03	7
10	100	0.932	108.68	0.40	1.65E-05	203	3.04	0.39	1.70E-03	6
20		0.932	186.05	0.37	1.65E-05	332	5.24	0.32	4.60E-03	9
10	200	0.932	80.17	0.70	9.81E-07	352	1.75	0.71	1.62E-03	8
20		0.932	182.41	0.56	9.80E-07	496	3.14	0.45	4.51E-03	7
30		0.932	250.39	0.53	9.79E-07	623	4.79	0.40	1.26E-03	9

Table 10 Comparison between Lanzcos Bidiagonalization and ACA for FATIMA_20493. The lowest time or number of iterations recorded for each column are highlighted in green.

5.3.2 Results after integration with solver

The *hierarchy_class* object using ACA is integrated with the current solver and the performance of the solver with hierarchical matvec as compared to dense matvec is shown in the Table 11-13. For clarity, the results are also illustrated in Figure 26 below. Note that in this section, the codes were ran in serial, and the tolerance ϵ in the admissibility criterion for the *hierarchy_class* object is set to 1e-4. The columns in Table 11-13 has the same definition as that stated in Section 3.3.

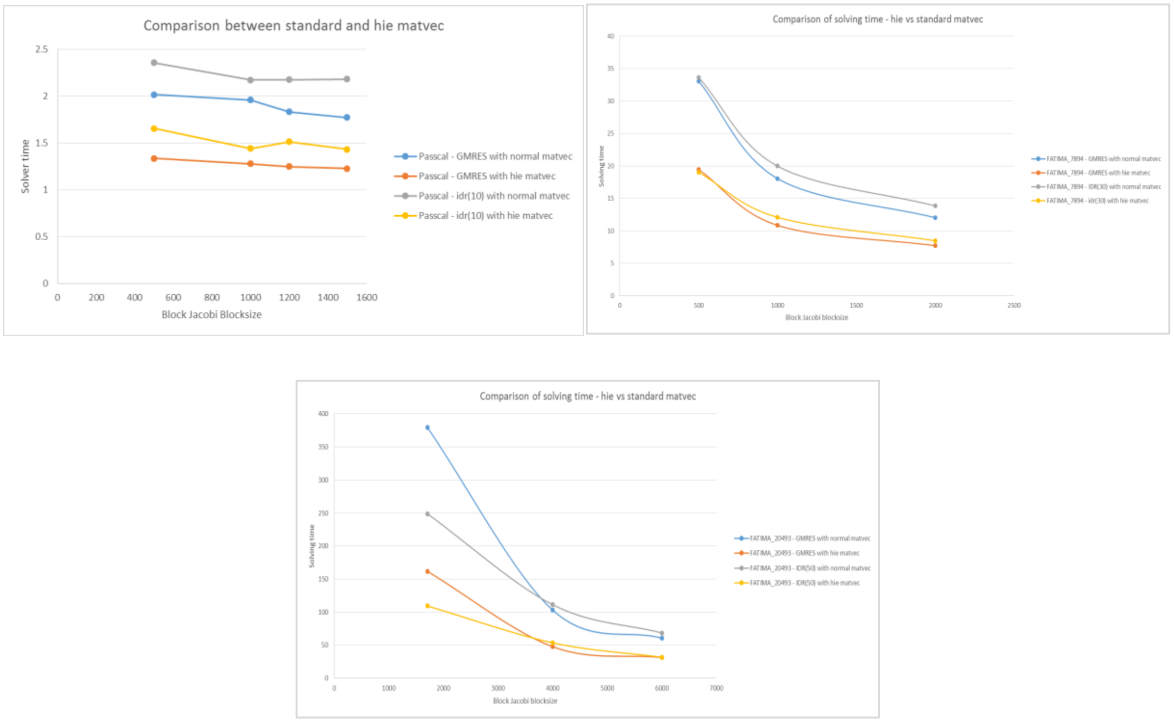


Figure 26 Performance of solver with hierarchical matvec as compared to dense matvec

Regardless of whether IDR(s) or GMRES is used, the time to solve the system using hierarchical matvec instead of dense matvec is significantly lower. This can be seen clearly In Figure 26. The larger the matrix, the more the gain in time when using hierarchical matvec. With Passcal, the solve time drops by about 30% when hierarchical matvec is used. For FATIMA_7894, the solve time drops by about 40%, and for FATIMA_20493, the solve time drops further by about 50%.

While this is true, the accuracy of the solution is not acceptable with hierarchical matvec. When each dense matvec is replaced with the hierarchical matvec, the effect is that the system matrix is perturbed. How well the solution of this approximate system estimates the exact solution depends on the condition of the system matrix. The accuracy at which this perturbed matrix approximates the system matrix is determined by the tolerance of the admissibility criterion. In the case when this tolerance is set to 1e-4, relative errors of about 1e-03, 1e-02 and 1e-01 are obtained for Passcal, FATIMA_7895 and FATIMA_20493 respectively.

To improve the situation, the tolerance of the admissibility criterion can be raised. The effect of raising this criterion for the case of FATIMA_20493 matrix with b=200, p=50 is shown in Table 14. As the tolerance is raised, the number of inadmissible blocks increases. This caused the time to do hierarchical matvec to increase. At a tolerance of 1e-5, the number of inadmissible blocks is so high that the hierarchical matvec becomes even more expensive than the matvec due to the overheads involved.

Therefore, it can be concluded that this method is not favorable for this application. The idea is that if this approximation can be constructed as a preconditioner instead, it may be more applicable. The next section explores the use of Hierarchical-LU decomposition as a preconditioner.

Passcal								
Block Jacobi block size	Wall clock time (s)		#iter	Rel error	Wall clock time (s)		#iter	Rel error
	Solve	Total			Solve	Total		
	GMRES with dense matvec				GMRES with hierarchical matvec			
500	2.02	2.20	91	3.54E-07	1.34	1.68	91	3.99E-03
1000	1.96	2.68	81	5.59E-07	1.28	2.22	81	3.99E-03
1200	1.83	2.83	77	2.99E-07	1.2	2.40	77	4.00E-03
	IDR(30) with dense matvec				IDR(30) with hierarchical matvec			
500	2.16	2.34	96	5.19E-07	1.42	1.76	96	3.99E-003
1000	2.04	2.81	86	1.58E-06	1.38	2.31	86	4.00E-003
1200	2.01	3.00	78	2.06E-07	1.38	2.52	83	4.00E-003

Table 11 Comparison of results between solvers with with dense matvec as compared to hierarchical matvec for Passcal. The results were obtained based on b=200, p=40.

FATIMA_7894								
Block Jacobi block size	Wall clock time (s)		#iter	Rel error	Wall clock time (s)		#iter	Rel error
	Solve	Total			Solve	Total		
	GMRES with dense matvec				GMRES with hierarchical matvec			
500	33.08	33.89	231	7.03E-07	19.47	22.02	232	6.51E-02
1000	18.07	21.28	121	1.57E-07	10.87	15.76	121	6.51E-02
2000	12.05	22.91	74	1.50E-07	7.74	21.99	75	6.51E-02
	IDR(30) with dense matvec				IDR(30) with hierarchical matvec			
500	33.64	34.47	236	3.86E-07	19.04	21.59	232	6.51E-02
1000	19.99	23.18	132	1.43E-07	12.11	17.00	135	6.51E-02
2000	13.87	26.38	83	1.08E-07	8.50	22.73	81	6.51E-02

Table 12 Comparison of results between solvers with with dense matvec as compared to hierarchical matvec for FATIMA_7894. The results were obtained based on b=200, p=50.

FATIMA_20493								
Block Jacobi block size	Wall clock time (s)		#iter	Rel error	Wall clock time (s)		#iter	Rel error
	Solve	Total			Solve	Total		
	GMRES with dense matvec				GMRES with hierarchical matvec			
1708	379.60	386.19	393	1.49E-06	161.77	175.04	396	6.15E-01
4000	103.22	156.81	103	1.17E-07	47.67	107.57	104	6.15E-01
6000	60.70	327.70	60	2.27E-07	31.34	304.33	61	6.15E-01
	IDR(50) with dense matvec				IDR(50) with hierarchical matvec			
1708	248.87	255.46	259	2.47E-07	109.23	123.10	258	6.15E-01
4000	111.30	164.81	109	1.66E-07	53.60	113.88	113	6.15E-01
6000	68.48	335.77	66	1.36E-07	31.68	304.75	65	6.15E-01

Table 13 Comparison of results between solvers with with dense matvec as compared to hierarchical matvec for FATIMA_20493. The results were obtained based on b=200, p=40.

ε	Number of inadm blocks	Time for hie matvec	Time for normal matvec	$\|Ax_{hie} - Ax_{exact}\|$
1e-4	1270	0.392	0.928	7.36E-04
5e-5	1393	0.448	0.928	3.80E-04
2e-5	3234	0.912	0.928	1.59E-04
1e-5	3572	1.032	0.928	5.25E-07

Table 14 Effect of increasing tolerance for admissibility criterion for the FATIMA_20493 matrix with b=200, p=50. The total number of inadmissible blocks at this level is 4096

6 PART 4: HIERARCHICAL LU-DECOMPOSITION

In the previous section, the implementation of the solver with hierarchical matvec was explored. However, the large relative error prohibits its use. Despite this, it has been shown that operations using the hierarchical form is indeed much cheaper as compared to the dense form. In this section, the use of hierarchical-LU decomposition to construct a hierarchical-LU preconditioner is explored. It is hopeful that this method will allow us to reap the benefits from the cheaper hierarchical operations while maintaining the accuracy of the computed solution.

The hierarchical-LU decomposition are discussed in various text. The ones referred to in this report are given in [7] and [12]. This section begins with a theoretical review of hierarchical LU decomposition. The implementation details and results follow after.

6.1 THEORY

The hierarchical-LU decomposition is a process that decomposed a hierarchical matrix A into a lower triangular hierarchical matrix L and an upper triangular hierarchical matrix U. This is illustrated in the figure below:

$$A \approx L_{\mathcal{H}} U_{\mathcal{H}}$$

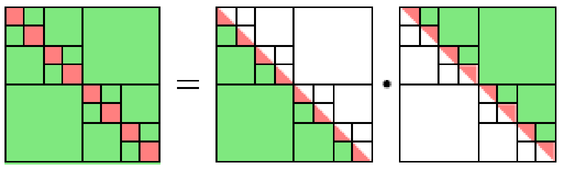


Figure 27 Illustration of the Hierarchical LU decomposition [12]

Note that L and U have the same hierarchical structure as A.

To formulate the algorithm for hierarchical-LU decomposition, first split the matrix A, L and U into four blocks:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \times \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}$$

Equation 11

Thus, the problem of solving for L and U is divided into four sub-problems [7]:

1. Compute L_{11}, U_{11} from the hierarchical-LU decomposition of A_{11}. Since L_{11} is in general not a lower triangular matrix unless a pivot P_{11} is used, thus, in general, $L_{11} = P_{11}^{-1}\hat{L}_{11}$, where $\hat{L}_{11}$ is lower triangular.
2. Compute U_{12} from the lower triangular equation $\hat{L}_{11}U_{12} = P_{11}A_{12}$
3. Compute L_{21} from the upper triangular equation $L_{21}U_{11} = A_{21}$
4. Compute L_{22}, U_{22} from the hierarchical-LU decomposition of $A_{22} - L_{21}U_{12}$. Again $L_{22} = P_{22}^{-1}\hat{L}_{22}$

Each sub-block of A is again a hierarchical matrix. Therefore, it can be seen now that the following major hierarchical matrix operations need to be defined:

1. Multiplication and Subtraction to obtain $A = A - LU$. We shall term this operation rounded subtraction. This is elaborated in Section 6.1.2.
2. Lower triangular Solver $LB = A$. This is elaborated in Section 6.1.3.
3. Upper triangular Solver $BU = A$. This is elaborated in Section 6.1.4.

To define these major operations, some basic operations with hierarchical matrix are needed. These are elaborated in the next sub-section. The subsequent sub-sections then elaborate on the three major operations. This section then ends by bringing together all the operations defined into the final hierarchical-LU decomposition algorithm.

6.1.1 Basic Hierarchical Matrix Operations

All hierarchical matrices have a recursive structure, where the main matrix is split into four blocks, and each block is further split into four blocks until the finest level of recursion $levels$. At this finest level, there exist essentially either low rank matrix operations or full matrix operations. Full matrix operations are well understood. Hence the first part of this section deals with how to handle low rank matrix operations. Namely, multiplication and addition (or subtraction) are considered.

The remaining parts of this section focuses on some of the operations that need to be defined for the three major operations required for hierarchical LU decomposition. These operations are hierarchical matrix multiplication, truncation of a hierarchical matrix into its low rank approximation, and addition (or subtraction) of a hierarchical matrix with low rank matrices.

6.1.1.1 Low rank matrix operations

6.1.1.1.1 Low rank matrix multiplication

Given a low rank matrix $R = AB^T \in \mathbb{C}^{N\times M}$ of rank p, and a full rank matrix $M \in \mathbb{C}^{M\times L}$, the result of the multiplication RM will be another rank p matrix, since $RM = AB^TM = A(M^TB)^T$.

Similarly, given another full rank matrix $N \in \mathbb{C}^{L\times N}$, NR is yet another rank p matrix, since $NR = (NA)B^T$.

Two rank p matrix multiplication also gives another rank p matrix. Let $= UV^T \in \mathbb{C}^{M\times L}$, then $RT = (A(U^TB)^T)V^T$ [12].

6.1.1.1.2 Formatted addition

Given two low rank matrix $R = AB^T \in \mathbb{C}^{N\times M}$ and $T = UV^T \in \mathbb{C}^{N\times M}$ of rank p, the result of $R + T$ is a rank $2p$ matrix. This is because $M = R + T = [A \quad U]\begin{bmatrix} B^T \\ V^T \end{bmatrix} = [A \quad U][B \quad V]^T$. To obtain the rank p approximation to M, the truncation operation which truncates a rank $2p$ matrix to a rank p matrix must be performed. This is termed here as RK-truncation, and is described in the next section. The addition operation, followed by RK-truncation, is termed as a whole as formatted addition.

6.1.1.1.3 RK-truncation: truncation of a rank k matrix to a rank p matrix

It is assumed here that $p < k$. Given a rank k matrix in the form $V^T \in \mathbb{C}^{N\times M}$, the rank p approximation $\widetilde{M} = U'V'^T$ can be obtained using a reduced singular value decomposition (SVD) operation [12]:

1. Perform QR-factorization of $U = Q_uR_u$ and $V = Q_vR_v$. Note that $Q_u \in \mathbb{C}^{N\times k}, R_u, R_v \in \mathbb{C}^{k\times k}$ and $Q_v \in \mathbb{C}^{M\times k}$. Thus $M = Q_uR_uR_v^TQ_v^T$.
2. Next, perform reduced SVD on $R_uR_v^T$. This is can be done using the SVD, Lanzcos Bidiagonalization, or ACA. In our case, ACA is chosen since it is the cheapest option. This gives $R_uR_v^T \approx U_RV_R^T$, $U_RV_R^T$ is of rank p.
3. Lastly, set $U' = Q_uU_R$ and $V' = Q_vV_R$

Because of the truncation operation, it is expected that formatted addition introduces an error. The error introduced by the truncation operation is discussed in [7] and [12]. Since the hierarchical-LU decomposition is to be used as a preconditioner, the elaborated error analysis is not presented here.

6.1.1.2 *Hierarchical matrix multiplication*

The result of the multiplication of a hierarchical matrix $A \in \mathbb{C}^{N\times M}$ with a full matrix $M \in \mathbb{C}^{M\times L}$ is another full matrix $AM \in \mathbb{C}^{N\times L}$. The multiplication can be carried out recursively. At the finest level, A is either a low rank or a full matrix. Thus AM can be obtained either by low rank matrix multiplication, or by full matrix-matrix multiplication. If not at the finest level and A is not admissible, A is divided into four sub-blocks, and M into two:

$$AM = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} M_1 \\ M_2 \end{bmatrix} = \begin{bmatrix} A_{11}M_1 + A_{12}M_2 \\ A_{21}M_1 + A_{22}M_2 \end{bmatrix}$$

Equation 12

The hierarchical matrix multiplication operation is then called recursively accordingly to Equation 12.

6.1.1.3 *Truncation: truncation of hierarchical matrix A to a low rank matrix*

The operation to truncate a hierarchical matrix to its low rank approximation is required for the rounded-subtraction operation. The truncation operation can be illustrated with the diagram below [12]:

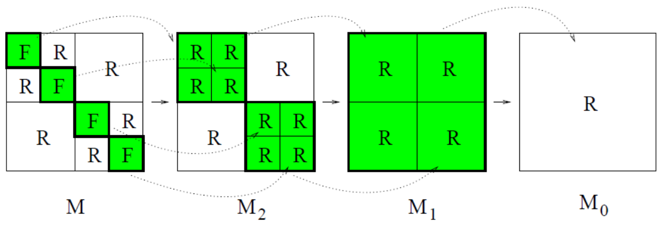


Figure 28 Illustration of truncation operation [12]

The hierarchical matrix in this case consists of full matrix blocks F and low rank matrix blocks R. Starting from the finest level (represented by matrix M in Figure 28), each F block is truncated to a low rank matrix block R using reduced SVD. Next, combine four low rank matrix sub-blocks into one low rank matrix block by formatted addition (note that here, $R_i = U_i V_i^T$):

$$\begin{bmatrix} R_1 & R_2 \\ R_3 & R_4 \end{bmatrix} = \begin{bmatrix} R_1 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & R_2 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ R_3 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & R_4 \end{bmatrix}$$
$$= \begin{bmatrix} U_1 \\ 0 \end{bmatrix} [V_1^T \quad 0] + \begin{bmatrix} U_2 \\ 0 \end{bmatrix} [0 \quad V_2^T] + \begin{bmatrix} 0 \\ U_3 \end{bmatrix} [V_3^T \quad 0] + \begin{bmatrix} 0 \\ U_4 \end{bmatrix} [0 \quad V_4^T] = [R]$$

This is performed until the coarsest level, level 0. By this time, a low rank approximation is attained for the entire hierarchical matrix.

6.1.1.4 Subtract-lowrank: Addition (or Subtraction) of a Hierarchical Matrix with a low rank matrix

Consider here the operation $A = A + UV^T$, where A is a hierarchical matrix. At the finest level, A is either a full matrix or a low rank matrix. If A is a full matrix, UV^T has to be formed explicitly and added to the full matrix. Otherwise, formatted addition is performed.

At coarser levels, if A is admissible, formatted addition is performed. If A is not admissible, then the function has to be recursively called. A is split into four blocks, while U and V are split into two blocks as follows:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} + \begin{bmatrix} U_1 \\ U_2 \end{bmatrix} [V_1^T \quad V_2^T]$$
$$= \begin{bmatrix} A_{11} + U_1V_1^T & A_{12} + U_1V_2^T \\ A_{21} + U_2V_1^T & A_{22} + U_2V_2^T \end{bmatrix}$$

Equation 13

The function is then recursively called in each of the four sub-blocks according to Equation 13.

The operation $A = A - UV^T$ is defined similarly.

6.1.2 Rounded subtraction Operation $A = A - LU$

The implementation of the rounded subtraction operation uses the functions that have been defined till now. It is assumed that A, L and U are all hierarchical matrices, but they may not be of the same hierarchical structure. This assumption is required because although the full A, L and U matrices share the same hierarchical structure, in rounded subtraction, A, L and U represent different sub-blocks of their respective full hierarchical matrices.

First consider the case when rounded subtraction function is called at the finest level. In this case, there are eight scenarios, depicted in the figure below.

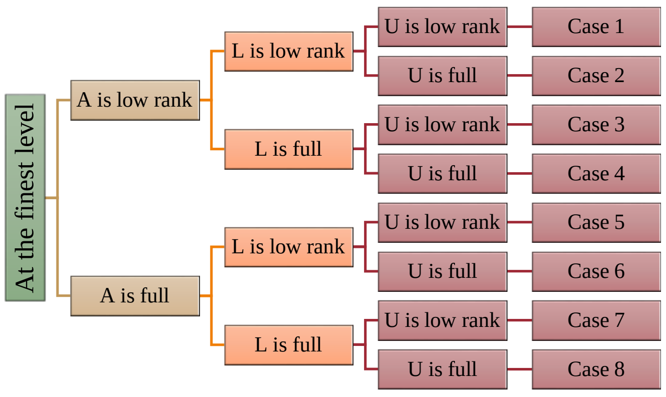


Figure 29 Cases to consider for rounded subtraction at the finest level

For Cases 1 to 4, since A is low rank, the result of $A - LU$ is also low rank. Now consider case 1 2 and 3, low rank matrix multiplication is first used to compute the product LU. Next, formatted addition is used to compute the rank p approximation to $A - LU$.

For Case 4, both L and U are full matrices. To subtract a full matrix from a low rank matrix efficiently, L and U have to be truncated into their low rank approximation. With the approximation, this case can then proceed like in Case 1, 2 or 3.

For Cases 4 to 8, the result of $A - LU$ is a full matrix. In all cases, either low rank multiplication or full multiplication is performed, followed by full matrix subtraction to arrive at the result.

When rounded subtraction is called at a higher level, the following cases have to be considered:

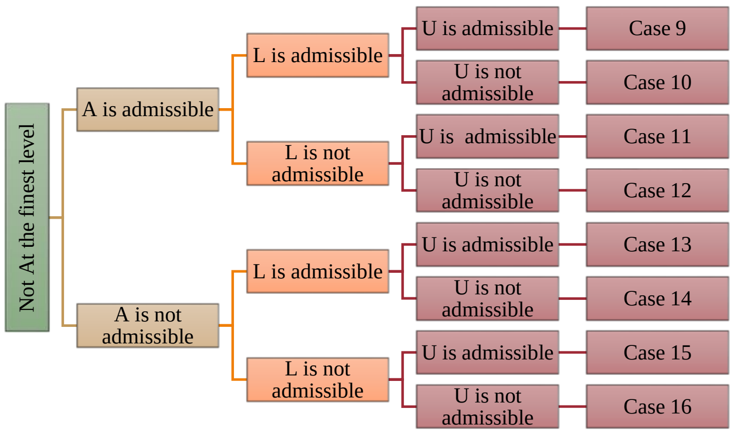


Figure 30 Cases to consider for rounded subtraction not at the finest level

If A is admissible, as it is for Case 9 to 12, this is very similar to Case 1 to 4. If L and U are both not admissible, then they have to be truncated to their low rank approximation. Otherwise, hierarchical matrix multiplication can be carried out. Once A and LU are both in their low rank form, then formatted addition can be applied to obtain $A - LU$.

When A is not admissible, the result of the rounded subtraction operation is another hierarchical matrix with the same structure as A. Consider Case 13. Since L and U are both admissible, the product LU is another low rank matrix. The subtract-lowrank operation can then be used to subtract a low rank matrix from a hierarchical matrix A.

When either L or U becomes admissible, as in Case 14 or 15, the recursive hierarchical matrix multiplication is called to form another low rank matrix. For instance, when U is admissible, $U = CD^T$, thus the operation becomes $A = A - (LC)D^T$.Since L is a hierarchical matrix, recursive hierarchical matrix multiplication can be used to evaluate LC, which is of size $N \times p$. Thus, the operation is again reduced to the subtraction of a low rank matrix $(LC)D^T$ from a hierarchical matrix A. This can be evaluated by the subtract-lowrank operation.

Lastly, Case 16 occurs when all three matrices are hierarchical. In this case the rounded subtraction function has to be recursively called. All three matrices can be split into four blocks as shown:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} - \begin{bmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11} - L_{11}U_{11} - L_{12}U_{21} & A_{12} - L_{11}U_{12} - L_{12}U_{22} \\ A_{21} - L_{21}U_{11} - L_{22}U_{21} & A_{22} - L_{21}U_{12} - L_{22}U_{22} \end{bmatrix}$$

Equation 14

For each block, rounded subtraction is first called on $A_{ij} = A_{ij} - L_{i1}U_{1j}$, then on $A_{ij} = A_{ij} - L_{i2}U_{2j}$.

6.1.3 Lower Triangular Solver $LB = A$

A lower triangular solver solves for B in $LB = A$, where L, B and A are all hierarchical matrices and L is lower triangular. With respect to our application, it can be assumed that B and A share the same hierarchical structure, while L need not. Again, the lower triangular solver is a recursive solver. Like all recursive functions, it is easier to deal first with the case at the finest level of recursion.

All lower triangular matrices lie on the diagonal. Therefore, at the finest level of recursion, L can only be a full rank matrix. Therefore, there are essentially only 2 cases to consider: first, when A/B is low rank, and second when A/B is full rank.

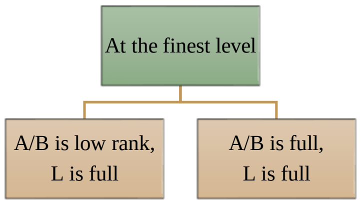


Figure 31 Cases to consider for lower triangular solver at the finest level

If A/B is full, then the usual lower triangular solver can be called to solve the lower triangular system. Otherwise, first, express $A = UV^T, B = CD^T$ to arrive at $LCD^T = UV^T$. The aim is to solve for matrix C and D. In this case, D can be solved by just letting $D = V$, and employ a lower triangular solver to the subsystem $LC = U$.

Consider now the case where the recursion level is not at the finest. There are again two cases: A/B is not admissible and A/B is admissible.

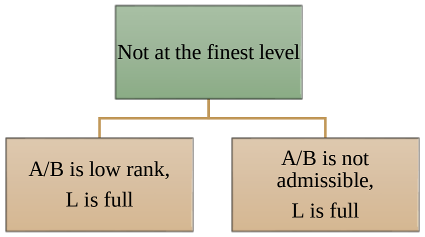


Figure 32 Cases to consider for lower triangular solver not at the finest level

When A/B is admissible, the case is handled similarly as that at the finest level. If A/B is not admissible, consider the following equation:

$$\begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

Equation 15

The lower triangular solver has to be recursively called as follows to solve for B [7]:

1. Solve for B_{11} by calling lower triangular solver on $L_{11}B_{11} = A_{11}$.
2. Solve for B_{12} by calling lower triangular solver on $L_{11}B_{12} = A_{12}$.
3. Solve for B_{21} by first calling rounded subtraction to obtain $A_{21} - L_{21}B_{11}$, then calling lower triangular solver on $L_{22}B_{21} = A_{21} - L_{21}B_{11}$.
4. Solve for B_{22} by first calling rounded subtraction to obtain $A_{22} - L_{21}B_{12}$, then calling lower triangular solver on $L_{22}B_{22} = A_{22} - L_{21}B_{12}$.

6.1.4 Upper Triangular Solver $\boldsymbol{BU = A}$

The upper triangular solver is very similar to the lower triangular solver. At the finest level, if A/B is full, dense upper triangular solver is used to solve for B. Otherwise, let $A = EF^T, B = CD^T$ to arrive at $CD^TU = EF^T$, where C and D are the unknowns. In this case, let $C = E$, and solve the smaller upper triangular system $D^TU = F^T$.

For the two cases when the level of recursion is not the finest, the case when A/B is admissible is handled the same way as at the finest level of recursion. The case when A/B is not admissible is solved by considering the following equation:

$$\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \times \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

Equation 16

The upper triangular solver is called recursively as follows [7]:

1. Solve for B_{11} by calling upper triangular solver on $B_{11}U_{11} = A_{11}$.
2. Solve for B_{21} by calling upper triangular solver on $B_{21}U_{11} = A_{21}$.
3. Solve for B_{12} by first calling rounded subtraction to obtain $A_{12} - B_{11}U_{12}$, then calling upper triangular solver on $B_{12}U_{22} = A_{12} - B_{11}U_{12}$.
4. Solve for B_{22} by first calling rounded subtraction to obtain $A_{22} - B_{21}U_{12}$, then calling lower triangular solver on $B_{22}U_{22} = A_{22} - B_{21}U_{12}$.

6.1.5 Hierarchical LU Decomposition – the Algorithm

Now with all the building blocks in place, it is time to put everything together into the hierarchical-LU decomposition algorithm introduced at the start of this section. At the finest level, there is only the case when A is full, since this algorithm is called only on diagonal blocks. Thus, the usual LU-decomposition routine is called to solve for L and U.

When not at the finest level, again only the case when A is hierarchical is considered (since A cannot be admissible as it is a diagonal block). Thus, A is split into four sub-blocks, and the hierarchical LU decomposition function, hierarchical lower triangular solver, and the hierarchical upper triangular solver is applied to the respective sub-blocks, according to Equation 11.

6.2 PROGRAM IMPLEMENTATION DETAILS

This section discusses how the hierarchical-LU decomposition is implemented and integrated with the solver. The *hierarchy_class* object data structure must first be updated to accommodate for the additional storage required for hierarchical-LU decomposition. Its subroutines must also be expanded to include the operations defined in Section 6.1. The resulting hierarchcial upper triangular U and lower triangular L matrix are then applied as a preconditioner to the solver. These are discussed in the subsections below:

6.2.1 Update to the data structure of a H-matrix for hierarchical-LU decomposition

The data structure of the *hierarchy_class* object is updated as shown below. The updates are explained in the following subsections.

```
type, public :: hierarchy

     !private
     ! List of arrays
     ! ----------------------------------------------
     DATATYPE1,                allocatable, dimension(:) :: U
     DATATYPE1,                allocatable, dimension(:) :: V
     integer(kind=SHORT),      allocatable, dimension(:) :: index_con
     integer(kind=SHORT),      allocatable, dimension(:) :: index_lvl_con
    integer(kind=SHORT),       allocatable, dimension(:) :: adm_row
     integer(kind=SHORT),      allocatable, dimension(:) :: adm_col
     integer(kind=SHORT),      allocatable, dimension(:) :: inadm_row
     integer(kind=SHORT),      allocatable, dimension(:) :: inadm_col
     integer(kind=SHORT),      allocatable, dimension(:) :: block_lvl_con
     integer(kind=SHORT),      allocatable, dimension(:) :: Ublockpos_con
     integer(kind=SHORT),      allocatable, dimension(:) :: Vblockpos_con
     DATATYPE1,                allocatable, dimension(:) :: matrix_N
     integer(kind=SHORT),      allocatable, dimension(:) :: Nblockpos_con
     DATATYPE1,                allocatable, dimension(:) :: U_compressed
     integer(kind=SHORT),      allocatable, dimension(:) :: Ublockpos_com_con
     DATATYPE1,                allocatable, dimension(:) :: V_compressed
     integer(kind=SHORT),      allocatable, dimension(:) :: Vblockpos_com_con
     DATATYPE1,                allocatable, dimension(:) :: LU
     DATATYPE1,                allocatable, dimension(:) :: LU_U
     DATATYPE1,                allocatable, dimension(:) :: LU_V
     integer(kind=SHORT),      allocatable, dimension(:) :: Pivot

     ! List of variables
     ! ----------------------------------------------
     integer(kind=SHORT) :: p
     integer(kind=SHORT) :: b
     integer(kind=SHORT) :: N
     integer(kind=SHORT) :: levels
     integer(kind=SHORT) :: alloc_stat
     real                :: tol

end type hierarchy
```

Figure 33 Updated data structure of the hierarchy_class object to cater for the hierarchical-LU decomposition

6.2.1.1 *Compression*

In the hierarchical-LU decomposition routine, there is a need to alter the hierarchical matrix A, since $A = A - LU$ have to be performed during the algorithm. In order to not change the original A matrix, which is required for the matrix-vector multiplication in the solver, there is a need to make a copy of this matrix to form the inadmissible matrix N_{levels}. It is however too expensive to make a copy of the entire matrix A. Instead, only the inadmissible blocks should be stored. In the same way, it is also helpful to compress the 1D arrays U and V after hierarchical splitting have been performed, since U and V are non zeros only in parts where the blocks become admissible. Therefore, the following new items are added to the hierarchy class data structure. The rationale behind each item is described below.

- $matrix\_N(:)$: contains the inadmissible blocks that made up N_{levels}
- $Nblockpos\_con(:)$: contains the index of the first element in $matrix\_N$ corresponding to the local block number at the finest level
- $U\_compressed(:)$: contains the compressed U array
- $Ublockpos\_com\_con(:)$: contains the index of the first element in the $U\_compressed$ array corresponding to the global block number
- $V\_compressed(:)$: contains the compressed V array
- $Vblockpos\_com\_con(:)$: contains the index of the first element in the $V\_compressed$ array corresponding to the global block number

To construct $matrix\_N$, the size of matrix_N required must first be computed. This is just the sum of the block sizes of all inadmissible blocks, $\sum_{inadmissible\ blocks} block\_size$. The inadmissible blocks are then stored in ascending order of the local block number at the finest recusion level $levels$, again, in column major format. Figure 34 below illustrates the structure of the 1D array used to store $matrix\_N$.

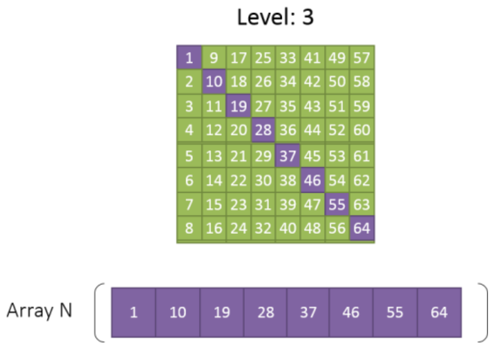


Figure 34 Structure of $Matrix\_N$ when the finest recursion level is 3. Purple boxes indicate inadmissible blocks, while green blocks indicate admissible ones.

To easily retrieve the matrix block corresponding to a local block number, the $Nblockpos\_con$ array is defined. This is an array of size $2^{levels} \times 2^{levels}$, and each element within the array corresponds to one local block at level $levels$. If the block is inadmissible, the position of the first element of that block in $matrix\_N$ is stored in $Nblockpos\_con$ array. Else, a 0 is stored. An example of how $Nblockpos\_con$ looks like for the FATIMA_7894 matrix at $levels = 3$ and $p = 50$ is shown below:

```
Nblockpos_con
           1      974170           0           0           0           0           0           0
     1948339     2922508     3896677           0           0           0           0           0
           0     4870846     5845015     6819184           0           0           0           0
           0           0     7792366     8765548     9737744           0           0           0
           0           0           0    10710926    11684108    12658277           0           0
           0           0           0           0    13632446    14606615    15580784    16554953
           0           0           0           0           0    17528135    18502304    19476473
           0           0           0           0           0           0    20449655    21422837
    22395033
```

Figure 35 Example of how $Nblockpos\_con$ looks like for FATIMA_7894 matrix with $levels = 3$ and $p = 50$

Next, arrays U and V are compressed into $U\_compressed$ and $V\_compressed$ arrays. The size of the arrays required to store the inadmissible blocks can be computed as $\sum_{l=2}^{levels}\sum_{admissible\ blocks} block\_row\_size \times p$ for $U\_compressed$ and $\sum_{l=2}^{levels}\sum_{admissible\ blocks} block\_col\_size \times p$ for V_compressed. After $U\_compressed$ and $V\_compressed$ array are allocated with this size, the low rank approximations for the admissible blocks are stored in ascending order of the global block number. The figure below illustrates the structure of the compressed arrays.

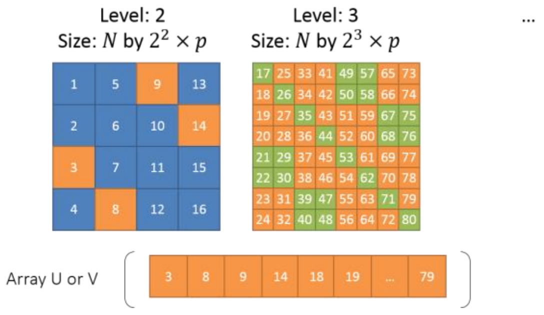


Figure 36 Structure of Array U and V after compression

Similar to $Ublockpos\_con$ and $Vblockpos\_con$ for the U and V 1D arrays, the $Ublockpos\_com\_con$ and $Vblockpos\_com\_con$ serves as a link between the global block number and their respective blocks in the $U\_compressed$ and $V\_compressed$ 1D array. There is an element allocated in $Ublockpos\_com\_con$ and $Vblockpos\_com\_con$ for every global block from level 2 onwards. If the global block is admissible, the element is equal to the index of the first element of the block in the $U\_compressed$ or $V\_compressed$ container. Otherwise, it is just set to 0. An example of how the $Ublockpos\_com\_con$ and $Vblockpos\_com\_con$ looks like for the FATIMA_7894 matrix with $levels = 3$ and $p = 50$ is shown below:

```
Ublockpos_com_con
          0           0           1       98701
          0           0           0      197351
     296001           0           0           0
     394701      493401           0           0
          0           0      592051      641401           0           0           0           0
          0           0           0      690701           0           0           0           0
     740001           0           0           0      789351      838701           0           0
     888051      937401           0           0           0      986751           0           0
          0           0     1036101           0           0           0     1085451     1134801
          0           0     1184101     1233451           0           0           0           0
          0           0           0           0     1282751           0           0           0
          0           0           0           0     1332101     1381451           0           0
    1430801


Vblockpos_com_con
          0           0           1       98701
          0           0           0      197401
     296051           0           0           0
     394751      493401           0           0
          0           0      592051      641401           0           0           0           0
          0           0           0      690751           0           0           0           0
     740101           0           0           0      789451      838801           0           0
     888151      937451           0           0           0      986751           0           0
          0           0     1036051           0           0           0     1085401     1134751
          0           0     1184101     1233451           0           0           0           0
          0           0           0           0     1282801           0           0           0
          0           0           0           0     1332151     1381451           0           0
    1430751
```

Figure 37 Example of what $Ublockpos\_com\_con$ and $Vblockpos\_com\_con$ looks like for FATIMA_7894 with $levels = 3$ and $p = 50$

After these arrays are constructed, the arrays $U(:), V(:), Ublockpos\_con(:)$, and $Vblockpos\_con(:)$ can be deallocated to free up memory.

6.2.1.2 Arrays to store result of LU-decomposition

The arrays required to store the results of the LU decomposition are named $LU, LU\_U,\ LU\_V$ and $Pivot$. Note that the upper triangular matrix $U$and lower triangular matrix L H-matrix are stored together to save storage space. Since L and U have the same hierarchical structure as $A,$the LU, LU_U and LU_V arrays have the same size as $matrix\_N, U\_compressed$ and $V\_compressed$ respectively. In addition, a 1D array named $Pivot$ is required to store the pivoting elements formed with the LU decomposition of the inadmissible blocks. This is of size N.

6.2.2 Additional subroutines

In addition to the subroutines already defined in Section 5.2.2, subroutines required to perform hierarchical-LU decomposition are listed. The theory behind these subroutines are provided in Section 6.1.

- **Hierarchy_compress**: to construct the compressed arrays to free up memory
- **RK_truncation**: truncates a matrix with rank $2p$ to rank p

- **truncate**: truncates a matrix block from L or U into its rank p approximation
- **subtract_lowrank**: performs $A = A - UV^T$, where A is a H-matrix and UV^T is a rank p matrix
- **hie_matmul_A**: performs $M\_out = M\_out\ + AM\_in$, where A is a H-matrix
- **hie_matmul_L**: performs $M\_out = M\_out\ + LM\_in$, where L is a lower triangular H-matrix
- **hie_matmul_U**: performs $M\_out = M\_out\ + UM\_in$, where U is an upper triangular H-matrix
- **hie_matmul_U_T**: performs $M\_out\ = M\_out + U^T M\_in$, where U is an upper triangular H-matrix
- **rounded_subtraction:** performs $A = A - LU$, A is a H-matrix, L is a lower triangular H-matrix and U is an upper triangular H-matrix
- **hie_LTS:** solves B in $LB = A$, A and B are H-matrices, L is a lower triangular H-matrix
- **hie_LTS_RK:** solves $C\ in\ LC = U$, C and U are full matrices, L is a lower triangular H-matrix
- **hie_UTS:** solves B in $BU = A$, A and B are H-matrices, U is an upper triangular H-matrix
- **hie_UTS_RK:** solves D in $DU = F$, D and F are full matrices, U is an upper triangular H-matrix
- **hie_LU:** solves L and U in $A = LU$, A is a H-matrix, L is a lower triangular H-matrix and U is an upper triangular H-matrix

The following subsections describe the implementation details of these subroutines, namely their input, output and pseudo-codes. Optimized library routines from LAPACK are used as much as possible. The routines name are provided where applicable.

6.2.2.1 Hierarchy_compress

The input to this subroutine are:

- A *hierarchy_class* object
- the size of the system: N
- The original system matrix: A

The *hierarchy_class* object is returned as an output, with the compressed arrays constructed and assigned.

The pseudo-code of this subroutine can be found below.

```
1. Compute the sum of all inadmissible blocks ($size\_N$)and fill up $Nblockpos\_con$
2. Allocate $matrix\_N$ with $size\_N$
3. Copy the appropriate inadmissible blocks from $A$ to $matrix\_N$ according to
   global block numbers
4. Compute the sum of the size of $U_{\sigma,\tau}$ and $V_{\sigma,\tau}$ for all admissible blocks ($size\_U$,
   $size\_V$)and fill up $Ublockpos\_com\_con$ and $Vblockpos\_com\_con$
5. Allocate $U\_compressed$ with $size\_U$ and $V\_compressed$ with $size\_V$
6. Copy the appropriate admissible blocks from $U$ and $V$ to $U\_compressed$ and
   $V\_compressed$ respectively
7. Deallocate $U$, $V$, $Ublockpos\_con$ and $Vblockpos\_con$
```

Algorithm 5 Hierarchy_compress

6.2.2.2 RK_truncation

The inputs to this subroutine are

- two arrays that define the rank $2p$ matrix, $A_{temp} \in \mathbb{C}^{size\_row \times 2p}$, $B_{temp} \in \mathbb{C}^{size\_col \times 2p}$
- the integer p that defines the rank of the low rank approximation
- tolerance tol_hie that defines the tolerance of the ACA algorithm

The subroutine outputs two arrays that define the truncated matrix of rank p, $U \in \mathbb{C}^{size\_row \times p}$ and $V \in \mathbb{C}^{size\_col \times p}$.

The pseudo-code for this subroutine is already outlined in Section 6.1.1.1.3.

6.2.2.3 Truncate

The inputs to this subroutine are:

- A *hierarchy_class* object for which defines L or U
- The local row and column block number that defines which block of L or U is to be truncated: $block\_start\_i$, $block\_start\_j$
- A flag to define if the block is admissible or not: adm_LU. adm_LU is 0 if block is inadmissible, and 1 if block is admissible.
- The level at which the block belongs to: l

- A flag to define if the block belongs to the L or U: L_flag. L_flag is 0 if block belongs to U, and 1 if block belongs to L.

The output to this subroutine are two arrays that define the truncated matrix of rank p, $U \in \mathbb{C}^{size\_\sigma \times p}$ and $V \in \mathbb{C}^{size\_\tau \times p}$.

With reference to the concept presented in Section 6.1.1.3, the pseudo-code for truncate subroutine is given below.

```
1)  If $adm\_LU = 1$
        i)  The block is already low rank. Return its low rank approximation
2)  Else
    a)  If $l =$ finest recursion level
        i)  call ACA to obtain its low rank approximation
    b)  Else
        i)  If this is a diagonal block
            (1) If $L\_flag = 1$
                (a) Split the block into three lower triangular sub-blocks and
                    recursively call truncate on each of these sub-blocks. Formatted
                    addition are then done on low rank approximations for the three
                    sub-blocks
            (2) Else
                (a) Split the block into three upper triangular sub-blocks and
                    recursively call truncate on each of these sub-blocks. Formatted
                    addition are then done on low rank approximations for the three
                    sub-blocks
        ii) Else
            (1) Split the block into four sub-blocks and recursively call truncate on
                each of these sub-blocks. Formatted addition are then done on low rank
                approximations for the four sub-blocks.
```

Algorithm 6 truncate

6.2.2.4 Subtract_lowrank

The inputs to this subroutine are:

- A *hierarchy_class* object which defines A in $A = A - UV^T$
- The local row and column block number that defines the block in A: $block\_start\_i$, $block\_start\_j$
- A flag to define if the block is admissible or not: adm_A. adm_A is 0 if block is inadmissible, and 1 if block is admissible.
- The level at which the block belongs to: l
- Two arrays which define U, V

The output to this subroutine is to update the *hierarchy_class* object A with $A - UV^T$.

The pseudo-code is given by:

```
1) If $adm\_A = 1$
      i) Perform formatted addition
2) Else
   a) If $l$ = finest recursion level
      i) Construct the product $UV^T$ explicitly (XGEMM) and subtract A with
         the product
   b) Else
      i) Split the block into four sub-blocks, $U$ and $V$ into halves as
         shown in Equation 13. Update each sub-blocks by recursively
         calling subtract-lowrank.
```

Algorithm 7 Subtract_lowrank

6.2.2.5 Hie_matmul_A, hie_matmul_L, hie_matmul_U

These are hierarchical matrix multiplications for a general H-matrix, lower or upper triangular H-matrix respectively. The input, output and pseudo-codes of these routines are similar, and therefore are discussed together in this section.

The inputs to these routines consist of:

- A *hierarchy_class* object which defines A, L or U
- The local row and column block number that defines which block is to be multiplied: $block\_start\_i$, $block\_start\_j$
- A flag to define if the block is admissible or not: adm_A, adm_L or adm_U. These are 0 if block is inadmissible, and 1 if block is admissible.
- The level at which the block belongs to: l
- An array which defines the input matrix M_in

The output to this routine is the matrix M_out.

The pseudo-code to these routines are given:

```
1) If $adm\_A, adm\_L$ or $adm\_U$ = 1
   a) Low rank matrix multiplication (XGEMM
2) Else
   a) If $l$ =finest recursion level
      i) For A, dense matrix multiplication for $A$ (XGEMM)
      ii) For $L$ and $U$, if this is a diagonal block
         (1) Dense upper or lower triangular matrix multiplication (XTRMM)
      iii)   Else
         (1) Dense matrix multiplication (XGEMM)
   b) Else
      i) For A, Split the block into four sub-blocks, and $M\_in$ and $M\_out$ into two
         halves according to Equation 12.Recursively call hie_matmul_A on each of
         the sub-blocks and update $M\_out$ according to Equation 12.
      ii) For L and U, if this is a diagonal block
         (1) Split the block into three lower or upper triangular blocks instead.
             Recursively call hie_matmul_L or hie_matmul_U on each of these sub-
             blocks and update $M\_out$.
      iii)   Else
         (1) Split the block into four sub-blocks like in 2.b.i.
```

Algorithm 8 hie_matmul_A, hie_matmul_L or hie_matmul_U

6.2.2.6 Hie_matmul_U_T

U^T is essentially a lower triangular matrix that looks like the following in block form:

$$U^T = \begin{bmatrix} U_{11}^T & 0 \\ U_{12}^T & U_{22}^T \end{bmatrix}$$

Thus, the pseudo-code for hie_matmul_U_T looks like that for hie_matmul_L, just that the transpose matrix multiplication is called instead. The inputs and output are also the same.

6.2.2.7 Rounded_subtraction

The subroutines defined before this are put together here to complete the operation $A = A - LU$. The inputs to this subroutine are:

- The *hierarchy_class* object that contains information on A, L and U
- The local row and column block number that defines the block in A to be updated: $block\_A\_i$, $block\_A\_j$
- A flag to define if this block is admissible or not: adm_A. adm_A is 0 if block A is inadmissible, and 1 if block is admissible.
- The local row and column block number that defines the block in L: $block\_L\_i$, $block\_L\_j$
- A flag to define if this block is admissible or not: adm_L
- The local row and column block number that defines the block in U: $block\_U\_i$, $block\_U\_j$

- A flag to define if this block is admissible or not: adm_U
- The level at which all these blocks belong to: l

The output is that the *hierarchy_class* object will have its $matrix\_N$, $U\_compressed$ and $V\_compressed$ updated according to $A = A - LU$.

The pseudo-code for this subroutine is given below. The cases referred to here are defined in Figures 30 and 31.

```
1)  Case 1:
    a)  Low rank multiplication of $L$ and $U$ to obtain low rank $LU$ (XGEMM)
    b)  Formatted subtraction between low rank matrices $A$ and $LU$
2)  Case 2:
    a)  Low rank multiplication of $L$ and $U$ to obtain low rank $LU$ (XGEMM)
    b)  Formatted subtraction between low rank matrices $A$ and $LU$
3)  Case 3:
    a)  Low rank multiplication of $L$ and $U$ to obtain low rank $LU$ (XGEMM)
    b)  Formatted subtraction between low rank matrices $A$ and $LU$
4)  Case 4:
    a)  Truncate $L$ and $U$ to obtain their low rank approximation
    b)  Low rank multiplication between the low rank approximations of $L$ and $U$ to
        obtain a low rank $LU$ (XGEMM)
    c)  Formatted subtraction between low rank matrices $A$ and $LU$
5)  Case 5:
    a)  Low rank multiplication of $L$ and $U$. The full matrix $LU$ is explicitly formed
        from the low rank multiplication (XGEMM)
    b)  Dense matrix subtraction between $A$ and $LU$
6)  Case 6:
    a)  Low rank multiplication of $L$ and $U$. The full matrix $LU$ is explicitly formed
        from the low rank multiplication (XGEMM)
    b)  Dense matrix subtraction between $A$ and $LU$
7)  Case 7:
    a)  Low rank multiplication of $L$ and $U$. The full matrix $LU$ is explicitly formed
        from the low rank multiplication(XGEMM)
    b)  Dense matrix subtraction between $A$ and $LU$
8)  Case 8:
    a)  Dense matrix multiplication of $L$ and $U$ to form full matrix $LU$ (XGEMM)
    b)  Dense matrix subtraction between $A$ and $LU$
9)  Case 9:
    a)  Low rank multiplication of L and U to obtain low rank $LU$ (XGEMM)
    b)  Formatted subtraction between low rank matrices $A$ and $LU$
10) Case 10
    a)  Low rank multiplication of $L$ and $U$ to obtain low rank $LU$ (XGEMM)
    b)  Formatted subtraction between low rank matrices $A$ and $LU$
```

```
11) Case 11
    a) Low rank multiplication of $L$ and $U$ to obtain low rank $LU$ (XGEMM)
    b) Formatted subtraction between low rank matrices $A$ and $LU$
12) Case 12:
    a) Truncate $L$ and $U$ to obtain their low rank approximation
    b) Low rank multiplication between the low rank approximation of $L$ and $U$ to
       obtain low rank $LU$ (XGEMM)
    c) Formatted subtraction between low rank matrices $A$ and $LU$
13) Case 13
    a) Low rank multiplication of $L$ and $U$ to obtain low rank $LU$ (XGEMM)
    b) Call subtract_lowrank to update $A$ with $A-LU$
14) Case 14
    a) Low rank multiplication of $L$ and $U$ to obtain low rank $LU$ (XGEMM)
    b) Call subtract_lowrank to update $A$ with $A-LU$
15) Case 15
    a) Low rank multiplication of $L$ and $U$ to obtain low rank $LU$ (XGEMM)
    b) Call subtract_lowrank to update $A$ with $A-LU$
16) Case 16
    a) Split all three $A$, $L$ and $U$ blocks into four sub-blocks each and apply
       rounded-subtraction recursively according to Equation 14.
```

Algorithm 9 rounded_subtraction

6.2.2.8 Hie_LTS and hie_LTS_RK

The subroutine hie_LTS is as described in Section 6.1.3.

An additional helper routine hie_LTS_RK is defined is for the case when block A/B becomes admissible at level l, while L is still hierarchical. In this case, there is the problem that A/B and L does not belong to the same level. While L has to be hierarchically divided into finer levels, the admissible blocks to be read from or to be updated in A or B belongs to level l. In addition, when this happens, a smaller lower triangular system, $LC = U$, where $C, U \in \mathbb{C}^{size_\tau \times p}$ needs to be solved. C and U are dense matrices, not hierarchical matrices. Therefore, Equation 15 should not be used. Instead, the following equation can be used instead:

$$\begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{pmatrix} C_1 \\ C_2 \end{pmatrix} = \begin{pmatrix} U_1 \\ U_2 \end{pmatrix}$$

$$\begin{pmatrix} L_{11}C_1 \\ L_{21}C_1 + L_{22}C_2 \end{pmatrix} = \begin{pmatrix} U_1 \\ U_2 \end{pmatrix}$$

Equation 17

Because of these reasons, a separate sub-routine hie_LTS_RK is created to deal with the case when block A/B becomes admissible while L is still hierarchical.

Hie_LTS have the following inputs:

- A *hierarchy_class* object that contains information for L, B and A. Note that B is a non-diagonal block that belongs to U.
- The local row and column block number that defines the block in A: $block\_A\_i$, $block\_A\_j$
- A flag to define if this block is admissible or not: adm_A. adm_A is 0 if block A is inadmissible, and 1 if block is admissible.
- The local row and column block number that defines the block in L: $block\_L\_i$, $block\_L\_j$
- A flag to define if this block is admissible or not: adm_L
- The level at which all these blocks belong to: l

Note that the local row and column block number for block B is the same as A.

The output to hie_LTS is an updated upper triangular hierarchical matrix U in the *hierarchy_class* object.

The inputs to hie_LTS_RK consist of:

- A *hierarchy_class* object that contains information for L
- An array that defines the matrix U in $LC = U$
- The local row and column block number that defines the block in L: $block\_L\_i$, $block\_L\_j$
- A flag to define if this block is admissible or not: adm_L
- The level at which block L belongs to: l

The output to hie_LTS_RK consist of the array defining C.

The pseudo-code for hie_LTS_RK and hie_LTS are given below:

```
1) If $l =$ finest recursion level
   a) Swap the rows in $U$ according to $Pivot$ (XLASWP)
   b) Solve the lower triangular system (XTRTRS)
2) Else
   a) Split $L$ into 3 sub-blocks and $U/C$ into halves according to Equation 17.
   b) Recursively call hie_LTS_RK on the first sub-block to solve $C_1$
   c) Call hie_matmul_L to obtain $L_{21}C_1$
   d) Dense matrix subtraction to obtain $U_2 = U_2 - L_{21}C_1$
   e) Recursively call hie_LTS_RK on the third sub-block to solve $C_2$
```

Algorithm 10 hie_LTS_RK

```
1) If $l$ =finest recursion level
   a) If $adm\_A = 0$
      i) Swap the rows in the dense matrix $A$ according to $Pivot$(XLASWP)
      ii) Solve the lower triangular system (XTRTRS)
   b) Else
      i) Let $B = CD^T$, $A = UV^T$. Solve for $D$ using $D = V$
      ii) Swap the rows in the matrix $U$ according to $Pivot$(XLASWP)
      iii) Solve the lower triangular system $LC = U$ (XTRTRS)
2) Else
   a) If $adm\_A = 0$
      i) Split block $A$,$B$ and $L$ into four sub-blocks each according to Equation 15.
      ii) Solve for $B_{11}$ by recursively calling hie_LTS
      iii) Solve for $B_{12}$ by recursively calling hie_LTS
      iv) Call rounded_subtraction to update $A_{21}$ with $A_{21} - L_{21}B_{11}$ and $A_{22}$ with $A_{22} -$
         $L_{21}B_{12}$
      v) Solve for $B_{21}$ and $B_{22}$ by recursively calling hie_LTS
   b) Else
      i) Let $B = CD^T$, $A = UV^T$. Solve for $D$ using $D = V$
      ii) Call hie_LTS_RK to solve for $C$.
```

Algorithm 11 hie_LTS

6.2.2.9 Hie_UTS and hie_UTS_RK

The upper triangular solver differs from the lower triangular solver mainly because an additional step is required to use the LAPACK routine. To solve a system $BU = A$ for B, one needs to transpose the system to obtain $U^T B^T = A^T$. This becomes then a lower triangular system and the same concept is applied.

The need to define hie_UTS_RK is similar to the reasons described in the previous section for the lower triangular solver. The inputs and outputs are very similar as well, and are stated here for completion.

The inputs to hie_UTS are:

- A *hierarchy_class* object that contains information for U, B and A. Note that B is a non-diagonal block that belongs to L.
- The local row and column block number that defines the block in A: $block\_A\_i$, $block\_A\_j$
- A flag to define if this block is admissible or not: adm_A. adm_A is 0 if block A is inadmissible, and 1 if block is admissible.
- The local row and column block number that defines which the block in U: $block\_U\_i$, $block\_U\_j$
- A flag to define if this block is admissible or not: adm_U

- The level at which all these blocks belong to: l

The output to hie_UTS is an updated lower triangular matrix L in the *hierarchy_class* object.

The inputs to hie_UTS_RK are:

- A hierarchy_class object that contains information for U in $D^T U = F^T$
- An array that defines the matrix F
- The local row and column block number that defines the block in U: $block\_U\_i$, $block\_U\_j$
- A flag to define if this block is admissible or not: adm_U
- The level at which block U belongs to: l

The pseudo-code is given below:

```
1) If $l =$ finest recursion level
   a) Solve the upper triangular system $U^T D = F$ (XTRTRS)
2) Else
   a) Split block $U$ into three upper triangular sub-blocks, $D$/F into halves
   b) Recursively call hie_UTS_RK on the first sub-block to solve $D_1$
   c) Call hie_matmul_U_T to obtain $U_{12}^T D_1$
   d) Dense matrix subtraction to obtain $F_2 = F_2 - U_{12}^T D_1$
   e) Recursively call hie_UTS_RK on the third sub-block to solve $D_2$
```

Algorithm 12 hie_UTS_RK

```
1) If $l$ =finest recursion level
   a) If $adm\_A = 0$
      i) Solve the upper triangular system $U^T B^T = A^T$(XTRTRS)
   b) Else
      i) Let $B = CD^T$, $A = EF^T$. Solve for $C$ using $C = E$
      ii) Solve the upper triangular system $U^T D = F$ (XTRTRS)
2) Else
   a) If $adm\_A = 0$
      i) Split block $A$,$B$ and $U$ into four sub-blocks each according to Equation 16.
      ii) Solve for $B_{11}$ by recursively calling hie_UTS
      iii)   Solve for $B_{21}$ by recursively calling hie_UTS
      iv) Call rounded_subtraction to update $A_{12}$ with $A_{12} - B_{11}U_{12}$ and $A_{22}$ with $A_{22} -$
          $B_{21}U_{12}$
      v) Solve for $B_{12}$ and $B_{22}$ by recursively calling hie_UTS
   b) Else
      i) Let $B = CD^T$, $A = EF^T$. Solve for $C$ using $C = E$
      ii) Call hie_UTS_RK to solve for $D$.
```

Algorithm 13 hie_UTS

6.2.2.10 Hie_LU

The final algorithm of hie_LU has inputs:

- A *hierarchy_class* object that contains information on A in $A = LU$.
- The local row and column block number that defines the block in A: $block\_A\_i$, $block\_A\_j$
- A flag to define if this block is admissible or not: adm_A. adm_A is 0 if block A is inadmissible, and 1 if block is admissible.
- The level at which block A belongs to: l

The output to this routine is an updated *hierarchy_class* object that now contains information in LU, LU_U, LU_V and $Pivot$.

The pseudo-code is given below:

```
1)  If $l =$ finest recursion levels
    a)  Dense matrix LU-decomposition (XGETRF)
2)  Else
    a)  Split block A into 4 sub-blocks according to Equation 11.
    b)  Recursively call hie_LU on the first sub-block $A_{11}$
    c)  Call hie_LTS on the second sub-block $A_{12}$
    d)  Call hie_UTS on the third sub-block $A_{21}$
    e)  Call rounded_subtraction to update $A_{22}$ with $A_{22} = A_{22} - L_{21}U_{12}$
    f)  Recursively call hie_LU on the last sub-block $A_{22}$
```

Algorithm 14 hie_LU

6.2.3 Integration with the solver

To use the result of the hierarchical-LU decomposition as a preconditioner, the original system is transformed into:

$$(LU)^{-1}Ax = (LU)^{-1}b$$

where L and U are hierarchical lower and upper triangular matrix respectively. Therefore, one needs to define a hierarchical lower triangular solver that solves $Ly = z$ for y, and an upper triangular solver that solves $Ux = y$ for x. Before the start of the iterative solver, the lower triangular solver is first applied to b, then the upper triangular solver is applied to obtain $(LU)^{-1}b$. In each iteration, the lower triangular and upper triangular solver is applied in the same way to the vector/matrix (Ax). Here, the lower triangular solver is named hie_LTS_vec and the upper triangular solver is named hie_UTS_vec.

The following subroutines are added to the *hierarchy_class* object to be able to perform lower and upper triangular solve in the solver:

- **hie_matvec_L:** This is similar to hie_matvec_A described in Section 5.2.2.4, but for a lower triangular matrix instead. This is used in hie_LTS_vec.
- **hie_matvec_U:** This is again similar to hie_matvec_A, but for an upper triangular matrix instead. This is used in hie_UTS_vec.
- **hie_LTS_vec:** Lower triangular solver to be called in the solver routine
- **hie_UTS_vec:** Upper triangular solver to be called in the solver routine

This subroutine hie_LTS_vec and hie_UTS_vec are very similar to the subroutines hie_LTS_RK and hie_UTS_RK. Therefore, one can refer to the pseudo-codes defined for hie_LTS_RK and hie_UTS_RK in algorithm 10 and 12. The only difference is that the hie_LTS_vec and hie_UTS_vec must cater for mixed precision, where the matrix Ax or b is of double precision, while L and U are of single precision. The reason for the mixed precision is detailed in [1].

For hie_matvec_L and hie_matvec_U, one can refer to the pseudo-code defined for hie_matvec_A in Section 5.2.2.4. The only difference is that the two new routines are modified for lower or upper triangular matrix multiplication instead.

With these subroutines defined, the hierarchical-LU preconditioner is constructed and applied as follows. First, hierarchy_split, hierarchy_compress and hie_LU subroutines are called to construct the preconditioner. To apply the preconditioner, hie_LTS_vec is called first, followed by hie_UTS_vec.

6.2.4 Parallelization

The hierarchical-LU preconditioner is parallelized using OpenMP. This is a preliminary attempt made so that the performance of parallel hierarchical-LU preconditioner can be compared to parallel block Jacobi. The comparison is desired, since the main benefit of block Jacobi lies in its parallelizability. However, there are better parallelizing strategies available where near optimal speedup can be attained. The implementation of these more complicated strategies are however not within the scope of this project. These are discussed in Section 7 as part of the recommendation for future works.

The main consideration for parallelizing the codes here is to ensure that the tasks distributed to the processor is large enough to justify the overhead required to run OpenMP on these routines. In addition, the routines associated with the *hierarchy_class* object is recursive in nature. Hence, one has to enable nested parallelism in OpenMP to achieve reasonable results. With this in mind, the following subroutines are parallelized:

- **hierarchy_split**
- **subtract_lowrank**
- **hie_matmul_A**
- **hie_matmul_L**
- **hie_matmul_U**
- **hie_matmul_U_T**
- **rounded_subtraction**
- **hie_LTS**
- **hie_UTS**
- **hie_LU**
- **hie_matvec_A**
- **hie_matvec_L**
- **hie_matvec_U**

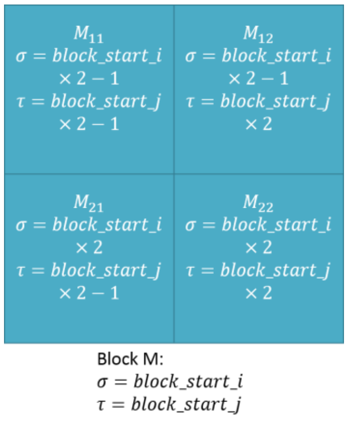
Subroutines like truncate are not parallelized because results showed that the tasks distributed to each processor is too small. Thus, the amount of overhead involved in setting up OpenMP dominates over the benefit of parallelism. Other subroutines hie_LTS_RK, hie_UTS_RK, are intrinsically sequential. Hence, they are not parallelized as well.

The following subsections discussed how the subroutines are parallelized. All parallel regions are enclosed in red.

6.2.4.1 Hierarchy_split

Here, each block can be assigned to a processor. The level of nested parallelism is limited to $l = 2$ here because the amount of work required to hierarchically split a block of matrix is relatively small. Hence, if the work is reduced to pieces that are too small, the overhead required to parallelize dominates over the gain in performance.

```
1) Compute the 4 block row and column numbers corresponding to the 4 sub-blocks of
   M at the next level. This can be obtained easily from $block\_start\_i$ and $block\_start\_j$
   as depicted in the figure below.
```


M_{11}
$\sigma = block\_start\_i \times 2 - 1$
$\tau = block\_start\_j \times 2 - 1$
M_{12}
$\sigma = block\_start\_i \times 2 - 1$
$\tau = block\_start\_j \times 2$
M_{21}
$\sigma = block\_start\_i \times 2$
$\tau = block\_start\_j \times 2 - 1$
M_{22}
$\sigma = block\_start\_i \times 2$
$\tau = block\_start\_j \times 2$
Block M:
$\sigma = block\_start\_i$
$\tau = block\_start\_j$

```
2) If $l = 0$
   PARALLEL DO
   a) Do for block 1 to 4
      i) Recursively call hierarchy_split
   END PARALLEL DO
3) Else
   PARALLEL DO IF($l \leq 2$)
   a) Do for block 1 to 4
      i) Call ACA or Lanzcos_Bidiag to determine if block is admissible
         (1) If admissible, store $U_{\sigma,\tau}$ and $V_{\sigma,\tau}$ in the 1D array $U\_compressed$ and
             $V\_compressed$ and update $adm\_row$ and $adm\_col$
         (2) Else
             (a) If $l$ =finest recursion level, update $inadm\_row$ and $inadm\_col$
             (b) Else, recursively call hierarchy_split
    END PARALLEL DO
```

Algorithm 15 Parallel hierarchy_split

6.2.4.2 *Subtract_lowrank*

The level of nested parallelism is again limited.

```
1) If $l$ = finest recursion level
   a) If $adm\_A = 0$
      i) Construct the product $UV^T$ explicitly and subtract A with the product
   b) Else
      i) Perform formatted addition
2) Else
   a) If $adm\_A = 1$
      i) Perform formatted addition
   b) Else, split the block into four sub-blocks, $U$ and $V$ into halves as shown in
      Equation 13.
      PARALLEL SECTIONS if ($l \le 3$)
      SECTION 1:
      i) $A_{11} = A_{11} - U_1 V_1^T$
      SECTION 2:
      ii) $A_{12} = A_{12} - U_1 V_2^T$
      SECTION 3:
      iii) $A_{21} = A_{21} - U_2 V_1^T$
      SECTION 4:
      iv) $A_{22} = A_{22} - U_2 V_2^T$
      END PARALLEL SECTIONS
```

Algorithm 16 Parallel subtract_lowrank

6.2.4.3 Hie_matmul_A, hie_matmul_L, hie_matmul_U, hie_matmul_U_T, hie_matvec_A, hie_matvec_L, hie_matvec_U

All these subroutines are parallelized in the same way. Hence, only the pseudo-code for hie_matmul_A is shown. Note that the level of nested parallelism is not limited in this case, since the amount of work required at the leaf nodes is high.

```
1) If $l$ =finest recursion level
   a) If $adm\_A = 0$
      i)  Dense matrix multiplication (XGEMM)
   b) Else
      i) Low rank matrix multiplication (XGEMM)
2) Else
   a) If $adm\_A = 1$
      i) Low rank matrix multiplication (XGEMM)
   b) Else, split the block into four sub-blocks, and $M\_in$ and $M\_out$ into two
      halves according to Equation 12.
      PARALLEL SECTIONS
      SECTION 1:
      i) $M\_out_1 = M\_out_1 + A_{11} M\_in_1 + A_{12} M\_in_2$
      SECTION 2:
      ii) $M\_out_2 = M\_out_2 + A_{21} M\_in_1 + M_{22} M\_in_2$
      END PARALLEL SECTIONS
```

Algorithm 17 Parallel hie_matmul_A

6.2.4.4 *Rounded_subtraction*

In the case of rounded_subtraction, only case 16 is parallelized, since the rest of the cases are leaf nodes. The algorithm below thus shows only case 16.

```
…

16) Case 16: Split all three $A$, $L$ and $U$ blocks into 4 sub-blocks each and apply
    rounded-subtraction recursively according to Equation 14.
    PARALLEL SECTIONS
    SECTION 1:
    a) $A_{11} = A_{11} - L_{11}U_{11} - L_{12}U_{21}$
    SECTION 2:
    b) $A_{12} = A_{12} - L_{11}U_{12} - L_{12}U_{22}$
    SECTION 3:
    c) $A_{21} = A_{21} - L_{21}U_{11} - L_{22}U_{21}$
    SECTION 4:
    d) $A_{22} = A_{22} - L_{21}U_{12} - L_{22}U_{22}$
    END PARALLEL SECTIONS
```

Algorithm 18 Parallel rounded_subtraction

6.2.4.5 *Hie_LTS and Hie_UTS*

The parallelization of hie_LTS and hie_UTS is again similar. Hence, only hie_LTS is shown here.

```
1) If $l$ =finest recursion level
    a) If $adm\_A = 0$
        i) Swap the rows in the dense matrix $A$ according to $Pivot$ (XLASWP)
        ii) Solve the lower triangular system (XTRTRS)
    b) Else
        i) Let $B = CD^T$, $A = UV^T$. Solve for $D$ using $D = V$
        ii) Swap the rows in the matrix $U$ according to $Pivot$ (XLASWP)
        iii) Solve the lower triangular system $LC = U$ (XTRTRS)
2) Else
    a) If $adm\_A = 0$
        i) Split block $A, B$ and $L$ into four sub-blocks each according to Equation 15.
        PARALLEL SECTIONS
        SECTION 1:
        ii) Solve for $B_{11}$ by recursively calling hie_LTS
        iii) Call rounded_subtraction to update $A_{21}$ with $A_{21} - L_{21}B_{11}$
        iv) Solve for $B_{21}$ by recursively calling hie_LTS
        SECTION 2:
        ii) Solve for $B_{12}$ by recursively calling hie_LTS
        iii) Call rounded_subtraction to update $A_{22}$ with $A_{22} - L_{21}B_{12}$
        iv) Solve for $B_{22}$ by recursively calling hie_LTS
        END PARALLEL SECTIONS
    b) Else
        ii) Let $B = CD^T$, $A = UV^T$. Solve for $D$ using $D = V$
        iii) Call hie_LTS_RK to solve for $C$.
```

Algorithm 19 Parallelized hie_LTS

6.2.4.6 Hie_LU

The algorithm for hie_LU is intrinsically sequential in the sense that work described in (2c) and (2d) can only start when (2b) is completed, and (2e) can only begin when (2c) and (2d) are done. Hence, only (2c) and (2d) can be parallelized here.

```
1) If $l =$ finest recursion levels
   a) Dense matrix LU-decomposition (XGETRF)
2) Else
   a) Split block A into 4 sub-blocks according to Equation 11.
   b) Recursively call hie_LU on the first sub_block $A_{11}$
   PARALLEL SECTIONS
   SECTION 1
   c) Call hie_LTS on the second sub-block $A_{12}$
   SECTION 2
   d) Call hie_UTS on the third sub-block $A_{21}$
   END PARALLEL SECTIONS
   e) Call rounded_subtraction to update $A_{22}$ with $A_{22} = A_{22} - L_{21}U_{12}$
   f) Recursively call hie_LU on the last sub-block $A_{22}$
```

Algorithm 20 Parallel hie_LU

6.3 RESULTS

This section presents the results obtain from solving the test problems with hierarchical-LU preconditioner. There are three main parameters that can be varied to influence the performance of the hierarchical-LU preconditioner, the tolerance below which the blocks are admissible tol_hie, the level of recursion determined by the minimum block length allowed b and the rank of the low rank approximation p. These parameters are varied, and the results are recorded. The best results obtained from these tests are then compared against the best results obtained using block Jacobi preconditioner. Results for both GMRES and IDR(s) are presented in this section.

This section is organized such that the results obtained using sequential computations are first presented. This is then followed by the results obtained when OpenMP is enabled. In view of the long test time involved for sequential computations with multiple RHS, results for multiple RHS are only obtained with OpenMP enabled.

6.3.1 Results based on sequential computations

The solve times obtained using the hierarchical-LU decomposition is shown in Table 17-20.

The columns in Table 17-20 has the same definitions as those in Section 3.3. The only difference is the **Prec const** is now the time required to perform hierarchy_split, hierarchy_compress and hie_LU, and **Prec apply** is now the time to apply the hie_LU preconditioner through calling hie_LTS_vec and hie_UTS_vec. Note that when the number of iteration exceeds 500, the results are not recorded.

Before discussing the many test results recorded in Table 17-20, the performance of the hierarchical-LU preconditioner as compared to the block Jacobi preconditioner is first studied. Table 15 below shows the best solve time obtained sequentially with block Jacobi preconditioner for the test matrices. This is the baseline results with which the solve times for hierarchical-LU decomposition are compared against.

Matrix	**Block-size for Block Jacobi Preconditioner**	**GMRES**		**IDR(s)**	
		Wall clock time (s)	**# iters**	**Wall clock time (s)**	**# iters**
FATIMA_20493	4000	239.8	103	249.836	110
FATIMA_7894	1000	21.3	121	23.23	133
PASSCAL	500	2.2	91	2.7	106
STEADYCAV1 (representing all steadycav matrices)	500	1.7	61	1.9	68

Table 15 Baseline results using block Jacobi preconditioner – Sequential. Results that are better than those obtained with hierarchical-LU preconditioner are highlighted in green

The best solve time attained using hierarchical-LU decomposition is shown in the table below:

Matrix	**Variables for hie_LU**	**GMRES**		**IDR(s)**	
		Wall clock time (s)	**# iters**	**Wall clock time (s)**	**# iters**
FATIMA_20493	Tol: 1e-03 b = 200 p = 50	134.74	51	140.35	55
FATIMA_7894	Tol: 1e-03 b = 200 p = 50	18.72	11	19.03	11
PASSCAL	Tol: 1e-02 b = 100 p = 20	2.62	44	2.72	46
STEADYCAV1 (representing all steadycav matrices)	Tol: 1e-02 b = 100 p = 30	2.65	32	2.86	37

Table 16 Best results obtained using hierarchical-LU preconditioner - Sequential. Results that are better than those obtained with block Jacobi preconditioner are highlighted in green

Although for the smaller matrices, hierarchical-LU preconditioner takes slightly more time as compared to block Jacobi, as the size of the system grows, hierarchical-LU preconditioner outperforms block Jacobi. The time required to solve the larger FATIMA_20493 system is 44% less when using the hierarchical-LU preconditioner as compared to the block Jacobi preconditioner.

The advantage of the hierarchical-LU preconditioner is illustrated more clearly by looking at Figure 38. As the size of the matrix increases, the time required to solve the system using hierarchical-LU preconditioner becomes much cheaper, as compared to that required for the block Jacobi preconditioner. This can be explained by the fact that with the block Jacobi preconditioner, the time to construct the preconditioner scales with $O\left(\left(\frac{N}{number\ of\ blocks}\right)^3\right)$, while that required for the hierarchical-LU preconditioner scales with $O(N(logN)^2)$ [12]. In addition, the use of the hierarchical-LU preconditioner with a reasonable tolerance preconditions the system very well, as can be seen by the low number of iterations required to solve the system. This reduces the number of dense matvec operations required, and therefore, improves the performance of the solver.

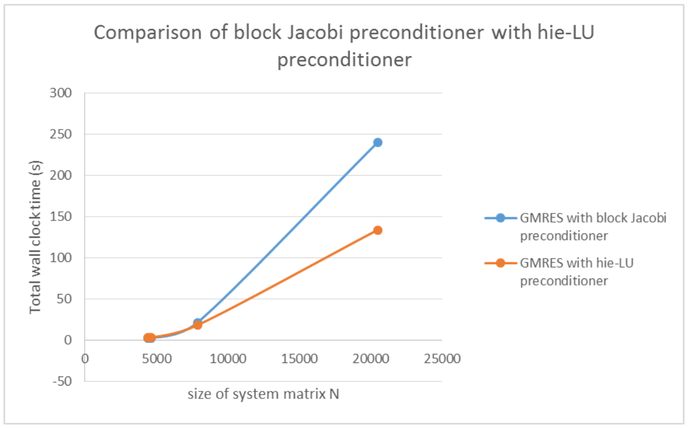


Figure 38 Comparison of block Jacobi preconditioner with hie-LU preconditioner

For the Steadycav matrices, it was mentioned in Section 5 that due to their structure, ACA algorithm with partial pivoting is unable to approximate the Steadycav matrices well. Using the hierarchical matrices as a preconditioner instead of an approximation to the system matrix, allows the ACA algorithm with partial pivoting to be used for the Steadycav matrices. One can still observe a significant reduction in number of iterations required.

Having established the success of the hierarchical-LU preconditioner, its dependence on the 3 parameters, tol_hie, b and p is now discussed.

FATIMA_20493																
Tol_hie	b	p	GMRES							IDR(s)						
			Wall clock time (s)					#iter	Rel error	Wall clock time (s)					#iter	Rel error
			Prec const	Prec apply	Matvec	Solve	Total			Prec const	Prec apply	Matvec	Solve	Total		
1.00E-02	100	10	27.69					>500		27.66	40.87	393.99	436.77	467.70	420	1.65E-05
		20	24.38					>500		24.34	34.22	348.55	384.47	409.27	371	8.07E-06
		30	25.46					>500		25.55	38.43	396.99	437.35	463.56	421	1.30E-05
		40	29.32					>500		29.30					>500	
		50	39.51					>500		37.64					>500	
	200	10	70.71	22.88	132.57	156.32	227.48	142	2.06E-05	70.80	28.94	152.99	182.72	256.75	162	8.03E-06
		20	42.82	22.03	169.20	192.61	235.93	181	6.12E-06	43.04	29.78	211.33	242.18	285.54	223	2.59E-06
		30	36.60	40.99	332.65	376.62	413.82	354	4.85E-06	36.62	29.33	237.33	267.84	304.79	252	4.00E-06
		40	35.18					>500		35.22	42.96	331.30	375.88	411.55	352	9.42E-07
		50	34.03					>500		34.09	50.84	405.45	458.26	492.92	431	3.43E-06
	600	10	122.61	24.91	97.79	123.18	246.21	105	6.54E-06	122.59	32.18	114.83	147.62	273.32	121	2.71E-06
		20	73.18	25.65	129.12	155.59	229.20	138	4.69E-06	73.41	32.72	151.51	185.01	258.59	160	1.24E-06
		30	61.04	22.29	121.27	144.29	205.82	130	1.32E-06	60.94	27.13	136.56	164.40	225.60	145	6.38E-07
		40	58.34	23.66	128.32	152.79	211.66	137	1.53E-06	58.43	27.22	147.52	175.49	237.28	156	7.10E-07
		50	53.55	23.62	131.11	155.58	209.68	140	4.49E-06	53.86	27.13	151.22	179.13	236.33	160	2.45E-06
		60	53.79	25.52	140.33	166.81	220.91	150	1.24E-06	53.86	32.38	168.32	201.56	255.78	179	1.77E-06
1.00E-03	100	10	136.41					>500		136.51					>500	
		20	72.66	13.39	88.90	102.69	176.10	95	2.86E-06	72.64	15.63	103.84	120.00	193.37	109	7.43E-07
		30	65.83	8.88	63.56	72.64	139.39	68	3.27E-07	65.85	10.00	71.68	82.06	148.90	75	1.30E-07
		40	70.87	8.63	61.39	70.53	142.58	66	5.84E-07	68.99	9.79	69.82	79.99	150.13	73	2.21E-07
		50	77.50	8.06	63.90	72.55	151.81	61	5.52E-07	90.65	9.54	78.48	88.39	180.63	68	1.99E-07
	200	10	406.40	42.51	110.88	154.00	560.90	119	2.08E-05	408.53	48.88	129.48	179.03	588.90	137	5.86E-06
		20	126.00	14.81	65.46	80.50	207.08	70	1.99E-06	125.88	18.39	74.55	93.33	219.55	78	8.32E-07
		30	87.70	10.42	57.94	68.54	156.63	62	5.84E-07	88.19	12.73	65.67	78.77	167.33	69	5.20E-07
		40	81.47	8.53	49.62	58.28	140.58	53	1.84E-07	81.47	9.41	54.22	63.94	145.85	57	1.84E-07
		50	77.97	8.00	47.76	55.88	134.74	51	1.22E-07	77.99	8.85	52.70	61.84	140.35	55	1.80E-07
		60	81.33	8.31	48.69	57.13	139.44	52	2.21E-07	81.37	9.83	53.59	63.73	145.78	56	1.64E-07
	600	10	963.87	32.20	53.34	85.70	1050.11	57	3.63E-06	971.32	35.79	59.34	98.66	1070.51	62	1.40E-06
		20	276.34	15.22	43.98	59.30	336.18	47	6.47E-07	276.69	16.32	47.21	64.10	340.96	49	4.33E-07
		30	164.34	14.52	52.45	67.12	231.99	56	1.66E-06	164.63	17.95	59.20	77.49	245.42	62	4.70E-07

		40	121.25	12.09	49.67	61.90	183.72	53	7.22E-08	121.33	13.60	55.62	69.54	191.12	58	3.06E-08
		50	92.82	10.04	46.87	57.04	150.46	50	1.28E-07	92.83	11.10	51.88	63.28	156.40	54	6.58E-08
		60	92.79	10.15	46.83	57.11	150.57	50	1.03E-06	92.67	12.14	52.54	64.99	158.03	55	4.17E-07
1.00E-04	100	10	331.32					>500		331.14	111.20	340.50	453.36	784.77	362	6.18E-05
		20	215.48	11.71	44.88	56.71	272.98	48	3.53E-07	214.12	13.07	50.05	63.41	277.99	52	8.84E-08
		30	194.35	6.77	28.06	34.88	229.94	30	1.34E-08	193.96	7.52	31.32	39.04	236.76	32	8.20E-09
		40	218.94	7072.00	31.70	39.14	259.01	31	2.28E-08	222.17	7.87	37.37	45.44	271.60	33	9.93E-09
		50	264.96	7.43	43.49	54.14	320.32	33	1.29E-08	271.13	8.05	50.39	58.78	334.16	34	3.85E-08
	200	10	1111.75	31.11	50.46	84.87	1197.17	54	9.13E-07	1123.46	33.89	58.13	92.34	1219.34	58	1.34E-06
		20	371.70	14.74	41.06	55.89	428.21	44	2.72E-07	371.99	15.90	44.51	60.68	432.95	46	3.46E-07
		30	301.42	8.83	27.15	36.03	337.90	29	3.29E-08	301.80	9.86	30.34	40.40	342.57	31	3.00E-08
		40	279.68	8.25	26.20	34.50	314.76	28	1.75E-08	279.79	8.98	28.49	37.66	317.94	29	2.78E-08
		50	247.97	8.01	27.17	35.36	283.88	29	1.92E-08	248.11	9.28	31.16	40.64	292.41	32	6.09E-09
		60	244.93	8.10	28.07	36.21	281.84	30	2.47E-08	244.82	8.76	30.23	39.19	287.76	31	1.92E-08
	600	20	697.21	16.64	31.67	48.37	745.84	34	2.97E-07	700.10	17.85	33.97	52.04	752.36	35	2.52E-07
		30	396.08	11.30	27.95	39.30	435.65	30	1.62E-07	397.48	12.60	31.24	44.05	441.78	32	9.26E-08
		40	338.43	10.24	27.16	37.45	376.19	29	1.39E-08	339.97	11.21	29.45	40.86	381.11	30	3.22E-08
		50	282.32	11.50	32.80	44.36	327.05	35	9.54E-08	282.99	12.64	35.89	48.75	332.06	37	3.43E-08
		60	266.34	11.45	33.46	44.99	311.11	36	1.76E-07	267.18	12.70	36.82	49.74	317.29	38	1.97E-07

Table 17 Results using hierarchy-LU preconditioner for FATIMA_20493. The best results obtained for GMRES and IDR(s) are highlighted in green

FATIMA_7894																
Tol_ hie	b	p	GMRES							IDR(s)						
			Wall clock time (s)					#iter	Rel error	Wall clock time (s)					#iter	Rel error
			Prec const	Prec apply	Matve c	Solve	Total			Prec const	Prec apply	Matvec	Solve	Total		
1.00E -02	100	10	19.47	7.93	23.18	31.60	51.08	173	9.19E-05	19.59	11.10	32.40	43.77	63.38	243	3.73E-05
		20	9.47	12.01	52.96	66.23	75.74	395	1.40E-05	9.38	9.93	41.16	51.42	60.84	305	7.94E-06
		30	8.51					>500		8.51					>500	
		40	9.38	10.57	47.49	59.10	68.56	355	2.22E-06	9.35	9.38	40.30	50.00	59.43	301	1.41E-06
		50	10.45	10.64	45.88	57.51	68.06	343	1.67E-06	10.38	8.60	36.97	45.86	56.34	278	1.25E-06
		60	11.73	10.80	44.01	55.73	67.57	328	4.64E-06	11.72	9.38	36.89	46.56	58.41	277	7.50E-07
	200	10	29.46	2.42	5.43	7.89	37.36	41	1.11E-07	29.41	2.67	5.99	8.72	38.14	44	6.89E-08
		20	12.52	2.05	7.15	9.26	21.79	53	1.16E-06	12.37	2.58	8.37	11.03	23.41	61	1.00E-07
		30	9.64	1.97	7.56	9.59	19.26	57	4.20E-07	9.71	2.43	9.53	12.04	21.78	68	2.93E-07
		40	9.45	2.25	8.81	11.14	20.62	65	5.41E-07	9.47	2.65	10.31	13.05	22.57	75	6.44E-07
		50	9.32	3.33	13.07	16.56	25.93	98	6.85E-07	9.35	3.78	15.02	18.93	28.33	109	4.87E-07
		60	9.88	3.69	14.11	17.98	27.92	105	5.23E-07	9.81	4.30	16.37	20.80	30.67	122	2.42E-07
	250	10	40.39	1.70	3.03	4.74	45.14	23	1.16E-08	40.16	2.03	3.35	5.41	45.58	24	1.12E-08
		20	19.69	1.62	3.97	5.62	25.32	30	1.92E-08	19.80	1.83	4.50	6.37	26.18	32	4.72E-08
		30	13.42	1.49	4.14	5.65	19.08	31	1.52E-07	13.45	1.69	4.75	6.49	19.96	34	8.04E-08
		40	13.14	1.53	4.43	5.98	19.14	33	8.79E-08	13.16	1.73	5.04	6.82	20.00	36	2.51E-08
		50	12.56	1.88	5.49	7.40	19.99	41	9.98E-08	12.58	2.03	5.96	8.04	20.64	43	9.29E-08
		60	12.34	1.85	5.35	7.23	19.60	40	1.30E-07	12.31	2.11	6.05	8.21	20.55	44	2.50E-08
1.00E -03	100	10	58.01	4.10	7.25	11.40	69.43	54	9.71E-08	57.71	4.91	8.05	13.03	70.76	58	1.62E-07
		20	23.62	1.17	3.32	4.50	28.17	25	2.59E-08	23.63	1.33	3.80	5.17	28.84	27	6.91E-09
		30	18.52	1.08	3.46	4.55	23.13	26	1.09E-08	18.58	1.19	3.81	5.03	23.67	27	1.09E-08
		40	17.73	0.98	3.21	4.20	22.01	24	9.54E-09	17.68	1.08	3.53	4.64	22.41	25	2.05E-08
		50	18.70	1.00	3.18	4.19	23.00	24	1.80E-08	18.82	1.14	3.64	4.82	23.73	26	8.13E-09
		60	20.84	1.04	3.19	4.24	25.21	24	8.27E-09	20.76	1.20	3.51	4.75	25.63	25	1.92E-08
	200	10	77.57	1.32	1.86	3.18	80.76	14	8.07E-08	77.61	1.53	2.17	3.73	81.34	15	6.92E-08
		20	28.11	0.81	1.86	2.68	30.81	14	3.47E-09	28.21	0.95	2.21	3.18	31.41	15	1.16E-09
		30	19.27	0.49	1.32	1.82	21.12	10	1.77E-08	19.39	0.60	1.67	2.30	21.72	11	3.49E-09
		40	16.93	0.50	1.46	1.97	18.94	11	9.19E-10	17.03	0.57	1.67	2.26	19.33	11	2.09E-09
		50	16.69	0.51	1.47	1.98	18.72	11	1.81E-09	16.71	0.57	1.68	2.27	19.03	11	3.61E-09
		60	17.30	0.56	1.59	2.15	19.51	12	5.70E-09	17.32	0.67	1.92	2.61	19.99	13	2.77E-09
	250	10	103.16	1.07	1.19	2.26	105.43	9	2.47E-10	103.03	1.21	1.37	2.61	105.64	9	2.31E-09
		20	37.98	0.55	0.92	1.47	39.46	7	1.10E-08	38.00	0.65	1.12	1.79	39.80	7	6.15E-09

		30	24.57	0.52	1.06	1.59	26.17	8	3.11E-09	24.61	0.60	1.26	1.88	26.51	8	1.23E-08
		40	20.02	0.54	1.19	1.73	21.77	9	1.36E-09	19.91	0.61	1.39	2.02	21.95	9	3.08E-09
		50	18.23	0.53	1.22	1.75	20.01	9	5.76E-09	18.34	0.65	1.57	2.24	20.60	10	1.38E-09
		60	18.86	0.57	1.35	1.93	20.81	10	3.29E-09	18.81	0.65	1.55	2.23	21.07	10	1.28E-08
1.00E -04	100	10	93.57	2.36	3.18	5.56	99.15	24	1.14E-08	93.72	2.59	3.51	6.14	99.88	25	9.39E-09
		20	49.68	2.03	3.99	6.04	55.76	30	2.39E-08	49.72	2.25	4.43	6.72	56.49	32	5.23E-08
		30	38.47	2.06	4.94	7.03	45.56	37	3.72E-08	38.42	2.29	5.57	7.92	46.40	40	2.31E-08
		40	36.94	1.40	3.51	4.92	41.95	26	2.86E-08	37.05	1.53	3.77	5.33	42.46	27	7.67E-08
		50	37.98	1.03	2.52	3.56	41.64	19	2.39E-08	37.95	1.24	3.02	4.30	42.35	21	9.76E-09
		60	40.90	0.58	1.34	1.93	42.95	10	3.43E-09	40.99	0.67	1.51	2.20	43.32	10	9.19E-09
	200	10	121.17	0.88	0.93	1.81	122.99	7	1.47E-09	121.26	1.04	1.12	2.17	123.44	7	2.96E-09
		20	53.50	0.96	1.60	2.57	56.09	12	1.65E-08	53.44	1.14	1.93	3.10	56.56	13	7.23E-09
		30	36.90	0.91	1.86	2.78	39.71	14	7.13E-09	36.96	1.07	2.20	3.29	40.28	15	3.24E-09
		40	32.80	0.79	1.73	2.52	35.36	13	3.25E-09	32.80	0.93	2.06	3.01	35.85	14	1.42E-09
		50	31.49	0.64	1.46	2.11	33.65	11	6.42E-08	31.54	0.78	1.79	2.59	34.18	12	1.38E-08
		60	31.27	0.48	1.08	1.57	32.89	8	4.31E-10	31.34	0.56	1.25	1.83	33.22	8	1.90E-09
	250	10	153.53	0.75	0.66	1.41	154.94	5	3.93E-10	153.37	0.93	0.85	1.79	155.17	5	1.89E-09
		20	68.42	0.81	1.06	1.87	70.30	8	3.65E-08	68.61	1.04	1.39	2.45	71.07	9	2.47E-09
		30	44.48	0.82	1.33	2.15	46.65	10	2.11E-09	44.46	0.94	1.54	2.50	46.98	10	3.20E-09
		40	39.12	0.56	0.94	1.50	40.64	7	3.41E-10	39.22	0.68	1.12	1.81	41.05	7	1.06E-09
		50	34.66	0.60	1.07	1.67	36.36	8	6.03E-09	34.66	0.78	1.38	2.18	36.87	9	4.87E-10
		60	32.31	0.39	0.67	1.06	33.40	5	1.13E-08	32.25	0.55	0.99	1.56	33.84	6	7.72E-10

Table 18 Results using hierarchy-LU preconditioner for FATIMA_7894. The best results obtained for GMRES and IDR(s) are highlighted in green.

PASSCAL																
Tol_ hie	b	p	GMRES							IDR(s)						
			Wall clock time (s)					#iter	Rel error	Wall clock time (s)					#iter	Rel error
			Prec const	Prec apply	Matve c	Solve	Total			Prec const	Prec apply	Matvec	Solve	Total		
1.00E -02	100	10	2.37	0.25	0.47	0.73	3.10	23	4.62E-08	2.38	0.27	0.52	0.81	3.19	24	1.33E-08
		20	1.37	0.34	0.91	1.25	2.62	44	3.34E-08	1.36	0.36	0.98	1.36	2.72	46	2.38E-07
		30	1.46	0.35	0.97	1.33	2.80	47	1.01E-06	1.48	0.40	1.08	1.49	2.98	51	1.69E-07
		40	1.70	0.40	1.12	1.53	3.24	49	7.71E-07	1.70	0.45	1.13	1.59	3.30	53	3.33E-07
		50	1.85	0.44	1.03	1.48	3.35	50	7.72E-07	1.85	0.48	1.14	1.65	3.51	54	3.60E-07
	200	10	5.22	0.30	0.41	0.72	5.94	20	1.68E-07	5.18	0.35	0.49	0.85	6.03	22	2.42E-08
		20	2.20	0.40	0.79	1.19	3.40	38	9.54E-08	2.21	0.46	0.89	1.37	3.58	42	3.45E-07
		30	1.74	0.43	0.95	1.39	3.13	46	7.41E-07	1.74	0.48	1.07	1.56	3.31	50	2.57E-07
		40	1.70	0.44	0.99	1.44	3.15	48	1.14E-06	1.71	0.51	1.13	1.66	3.38	53	2.27E-07
		50	1.84	0.48	1.04	1.53	3.37	50	9.29E-07	1.85	0.53	1.17	1.72	3.58	55	2.36E-06
1.00E -03	100	10	3.99	0.12	0.17	0.29	4.28	8	4.87E-09	3.97	0.15	0.22	0.37	4.35	9	3.60E-10
		20	2.82	0.13	0.23	0.36	3.18	11	1.01E-08	2.84	0.15	0.26	0.41	3.26	11	1.14E-08
		30	2.98	0.19	0.39	0.59	3.58	19	7.42E-09	3.03	0.23	0.46	0.70	3.74	21	1.34E-09
		40	3.01	0.18	0.37	0.56	3.58	18	1.78E-08	3.02	0.21	0.42	0.64	3.67	19	6.54E-09
		50	3.26	0.21	0.41	0.62	3.89	20	1.21E-08	3.31	0.24	0.46	0.71	4.04	21	9.70E-09
	200	10	7.45	0.10	0.10	0.21	7.66	5	3.62E-10	7.45	0.13	0.13	0.27	7.71	5	6.93E-09
		20	4.92	0.17	0.23	0.40	5.32	11	6.88E-10	4.97	0.19	0.26	0.45	5.43	11	2.65E-09
		30	3.58	0.24	0.37	0.61	4.19	18	2.50E-08	3.63	0.29	0.45	0.74	4.38	20	1.40E-08
		40	2.96	0.21	0.39	0.61	3.57	19	8.70E-09	2.96	0.23	0.42	0.67	3.63	19	2.41E-08
		50	2.91	0.22	0.39	0.61	3.53	19	2.29E-08	2.92	0.25	0.49	0.74	3.67	20	2.58E-08
1.00E -04	100	10	6.15	0.13	0.15	0.28	6.42	7	3.20E-09	6.12	0.15	0.18	0.33	6.46	7	5.80E-09
		20	3.96	0.10	0.14	0.25	4.21	7	4.46E-09	3.95	0.12	0.17	0.30	4.26	7	1.22E-08
		30	3.87	0.10	0.17	0.27	4.15	8	2.27E-09	3.90	0.12	0.20	0.32	4.23	8	8.58E-09
		40	4.81	0.09	0.14	0.24	5.06	7	1.78E-10	4.67	0.11	0.17	0.29	4.97	7	4.19E-10
		50	4.97	0.09	0.15	0.24	5.22	7	3.12E-10	4.96	0.11	0.18	0.30	5.27	7	7.20E-10
	200	10	12.88	0.13	0.10	0.24	13.12	5	4.08E-09	12.88	0.19	0.15	0.35	13.24	6	3.52E-11
		20	6.92	0.08	0.08	0.17	7.09	4	8.12E-12	6.94	0.11	0.11	0.22	7.17	4	9.60E-12
		30	4.73	0.11	0.14	0.26	4.99	7	3.87E-09	4.78	0.15	0.20	0.35	5.14	8	1.01E-10
		40	4.18	0.10	0.15	0.25	4.43	7	3.68E-09	4.18	0.12	0.17	0.30	4.49	7	6.46E-09
		50	4.41	0.09	0.12	0.21	4.63	6	1.56E-09	4.41	0.11	0.15	0.27	4.68	6	1.04E-08

Table 19 Results using hierarchy-LU preconditioner for PASSCAL. The best results obtained for GMRES and IDR(s) are highlighted in green

STEADYCAV1																
			GMRES							IDR(s)						
			Wall clock time (s)							Wall clock time (s)						
Tol_ hie	b	p	Prec const	Prec apply	Matve c	Solve	Total	#iter	Rel error	Prec const	Prec apply	Matvec	Solve	Total	#iter	Rel error
1.00E -02	100	10	5.76	0.57	0.78	1.35	7.12	34	3.78E-06	5.77	0.70	0.94	1.65	7.43	40	6.09E-07
		20	1.98	0.34	0.73	1.07	3.06	32	5.76E-05	1.99	0.38	0.96	1.35	3.34	37	3.48E-05
		30	1.63	0.28	0.73	1.01	2.65	32	9.35E-05	1.63	0.34	0.88	1.22	2.86	37	1.43E-06
		40	1.61	0.28	0.78	1.06	2.69	34	1.58E-05	1.61	0.33	0.99	1.33	2.96	38	1.98E-05
		50	1.93	0.30	0.78	1.08	3.02	34	1.37E-05	1.94	0.34	0.88	1.22	3.17	37	7.82E-05
	200	10	7.66	0.60	0.73	1.34	9.00	32	2.52E-05	7.65	0.71	0.86	1.57	9.23	36	2.54E-06
		20	2.36	0.32	0.66	0.98	3.34	29	3.45E-05	2.36	0.36	0.72	1.08	3.44	30	5.53E-06
		30	1.85	0.32	0.73	1.05	2.90	32	8.87E-05	1.85	0.38	0.85	1.23	3.08	36	4.83E-05
		40	1.67	0.31	0.73	1.04	2.72	32	2.36E-05	1.66	0.36	0.86	1.22	2.89	36	3.86E-05
		50	1.78	0.31	0.73	1.05	2.83	32	5.50E-05	1.78	0.36	0.83	1.19	2.97	35	3.77E-05
1.00E -03	100	10	17.54	0.59	0.48	1.07	18.61	21	3.66E-05	17.55	0.73	0.58	1.31	18.86	24	2.53E-05
		20	7.09	0.36	0.45	0.81	7.90	20	4.02E-06	7.09	0.41	0.53	0.95	8.05	22	2.12E-06
		30	3.84	0.28	0.50	0.78	4.63	22	1.41E-05	3.84	0.32	0.57	0.90	4.75	24	2.75E-06
		40	3.29	0.24	0.50	0.74	4.05	22	3.18E-05	3.30	0.28	0.58	0.87	4.18	24	5.85E-05
		50	3.77	0.25	0.52	0.78	4.57	23	1.12E-05	3.76	0.30	0.60	0.90	4.67	25	1.92E-05
	200	10	18.02	0.42	0.34	0.76	18.78	15	8.97E-06	18.02	0.51	0.42	0.93	18.96	17	2.59E-06
		20	8.57	0.41	0.46	0.87	9.44	20	6.35E-05	8.55	0.49	0.56	1.05	9.61	23	2.92E-06
		30	3.91	0.29	0.45	0.74	4.65	20	1.82E-05	3.92	0.35	0.56	0.91	4.83	23	2.05E-06
		40	2.97	0.24	0.45	0.70	3.68	20	7.27E-06	2.95	0.28	0.53	0.82	3.77	22	6.69E-05
		50	3.06	0.24	0.46	0.70	3.76	20	4.04E-05	3.04	0.28	0.53	0.81	3.86	22	4.23E-05
1.00E -04	100	10	20.68	0.40	0.30	0.70	21.38	13	4.78E-07	20.83	0.48	0.35	0.83	21.67	14	3.24E-06
		20	13.75	0.55	0.50	1.05	14.81	22	1.56E-05	13.78	0.67	0.62	1.30	15.08	26	2.21E-06
		30	11.30	0.50	0.54	1.05	12.36	24	5.53E-05	11.21	0.61	0.67	1.28	12.50	28	1.43E-05
		40	9.58	0.38	0.48	0.86	10.45	21	1.68E-05	9.57	0.46	0.58	1.05	10.63	24	4.42E-06
		50	9.39	0.29	0.39	0.68	10.08	17	9.41E-06	9.42	0.35	0.46	0.81	10.25	19	6.34E-06
	200	10	19.49	0.35	0.27	0.62	20.11	12	3.53E-06	19.50	0.42	0.33	0.75	20.25	13	9.52E-06
		20	14.60	0.43	0.39	0.82	15.42	17	6.95E-06	14.62	0.50	0.44	0.94	15.57	18	7.93E-07
		30	11.09	0.47	0.48	0.95	12.04	21	2.27E-05	11.09	0.61	0.62	1.23	12.33	26	1.26E-06
		40	8.22	0.35	0.41	0.76	8.99	18	1.37E-05	8.27	0.40	0.47	0.87	9.15	19	1.55E-05
		50	7.21	0.28	0.36	0.65	7.86	16	7.98E-07	7.23	0.34	0.44	0.79	8.03	18	2.68E-06
		60	7.01	0.26	0.34	0.60	7.62	15	4.53E-06	7.02	0.30	0.40	0.70	7.73	16	2.36E-08

Table 20 Results using hierarchy-LU preconditioner for STEADYCAV1. The best results obtained for GMRES and IDR(s) are highlighted in green

6.3.1.1 *Effect of tol_hie on the performance of hierarchical-LU preconditioner*

The tolerance tol_hie affects the accuracy of how well the hierarchical-LU preconditioner approximates the original system matrix A. Therefore, the general trend expected is that the number of iterations required to solve the system increase as tol_hie decreases. This can be observed from the results shown in Table 17-20. For FATIMA_20493 at $b = 100$, when tol_hie drops to 1e-2, the preconditioner became so inaccurate that the number of iterations required exceeds 500. When tol_hie is increased to 1e-4, the average number of iterations required is only about 35 (disregarding the case when p=10).

However, the lower the tolerance, the cheaper it is to perform hierarchical-LU decomposition. This is because more blocks are allowed to be admissible. Again using FATIMA_20493 matrix at $b = 100$ as an example, the average time required to construct the preconditioner when tol_hie is 1e-4 is roughly 245 s, while that required when tol_hie is 1e-2 is only about 30 s. Hence, there is a tradeoff between the time to construct the preconditioner and the solve time required.

From Table 17 -20, one can obtain the graphs in Figure 39 below by looking at the various timings recorded for a fixed p and b while varying the tolerance. The same trends were observed for GMRES and IDR(s) solver, hence, only results for GMRES are plotted. The tradeoff between hie-LU time and GMRES solver time as tol_hie increases is clearly shown for FATIMA_20493. Because of this trade-off, the optimal tolerance for FATIMA_20493 is around 1e-3.

As the size of the matrix decreases, the time to perform dense matvec decreases with complexity $O(N^2)$. As such, the substantial decrease in time required to construct the preconditioner as tol_hie decreases dominates over the increase in number of iterations required. This can be observed from the results of the smallest test matrix Passcal. The optimal tolerance here is found to be 1e-2.

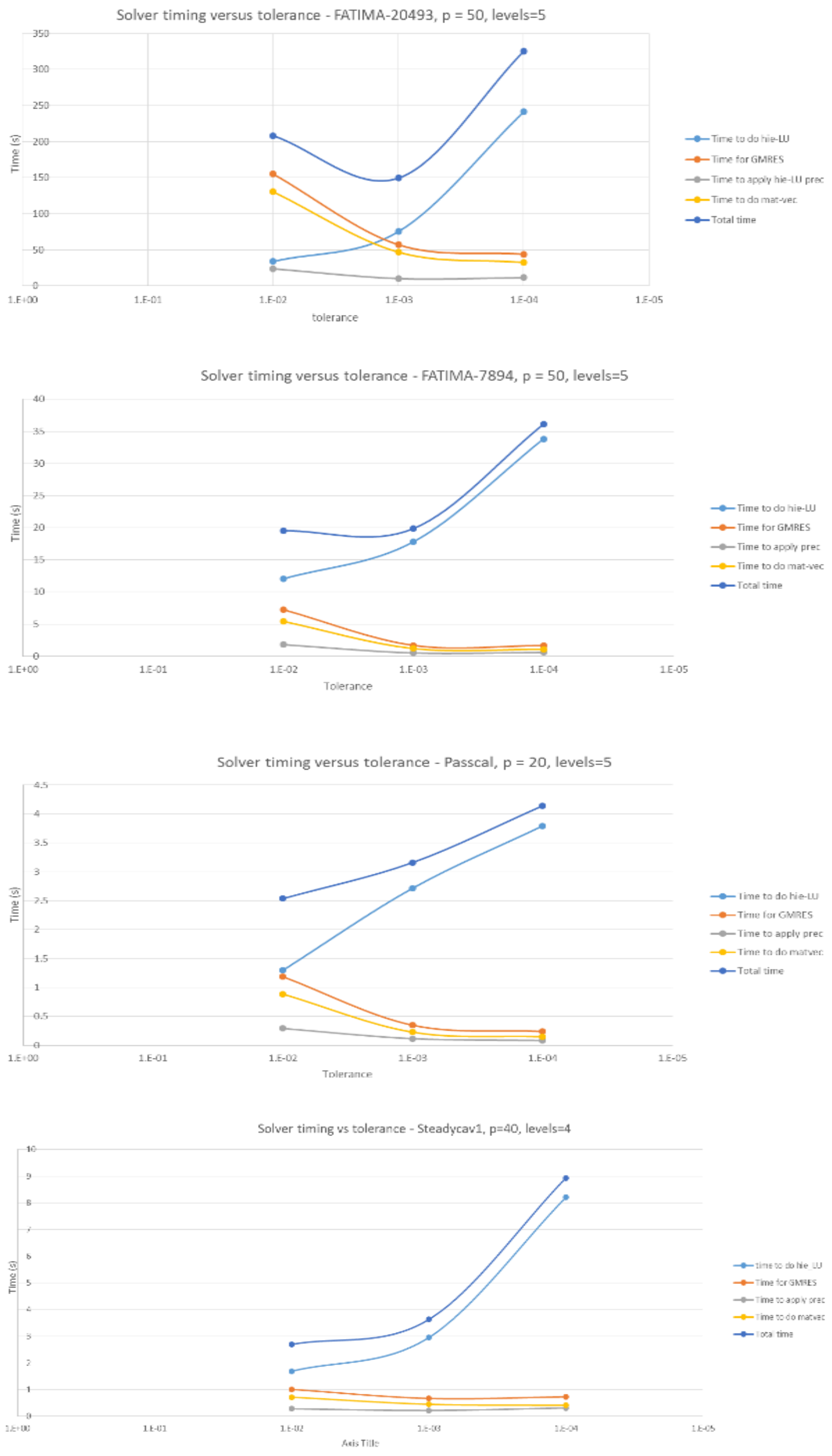


Figure 39 Solver timings versus tolerance for the test matrices

6.3.1.2 *Effect of b on the performance of hierarchical-LU preconditioner*

The smaller b is, the deeper the recursion. There is an optimal b, because while deeper recursion implies additional work and storage, it also means more blocks can become admissible. This is illustrated using Figure 40, which shows the sparsity plot of N_{levels} at different $levels$ for Passcal. When $levels$ increase from 4 to 5, a large part of the matrix becomes admissible. One can then expect the amount of work save from the additional admissible blocks to dominate over the extra work required from the deeper level of recursion. However, if $levels$ increase to 6, the proportion of the matrix that becomes admissible is relatively smaller. In this case, the additional admissible blocks may not justify the deeper level of recursion. Because of this reason, it is not surprising to find that the time required to perform the hierarchy-LU decomposition in this case is 4.89s for $levels = 4$, 4.02s for $levels = 5$ and 5.64 for $levels = 6$.

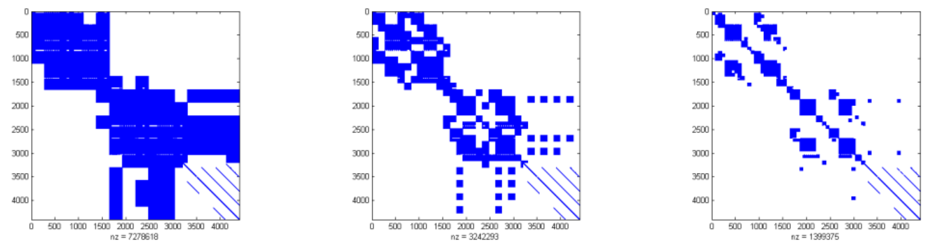


Figure 40 Sparsity of the inadmissible matrix N for recursion levels 4 (left), 5 (middle) and 6(right) - (Passcal, p=35, tol_hie=1e-4)

In general, the trend observed is that the optimal $levels$ is between 5 or 6 for all the test matrices.

6.3.1.3 *Effect of p on the performance of hierarchical-LU preconditioner*

The optimal value of p depends very much on the physics of the system, reflected through the inherent rank of the off diagonal blocks of the system matrix. It can be observed from Figure 41 that as p approaches a threshold value, the time required for hierarchical-LU decomposition drops drastically. As p increases beyond this threshold, the time required increase or decrease only slightly. The increase can be explained by the fact that as p increases, the work required for every low rank operation increases. In addition, it can also cause more approximation since more blocks become admissible, and therefore, possibly increasing the number of iterations required to solve. However, increase in p can also result in a more accurate low rank approximation to each low rank block. In this case, a slight drop in iteration can be observed, with a corresponding drop in time required for GMRES/IDR(s). A combination of these effects result in the unstructured behavior observed above the threshold p value.

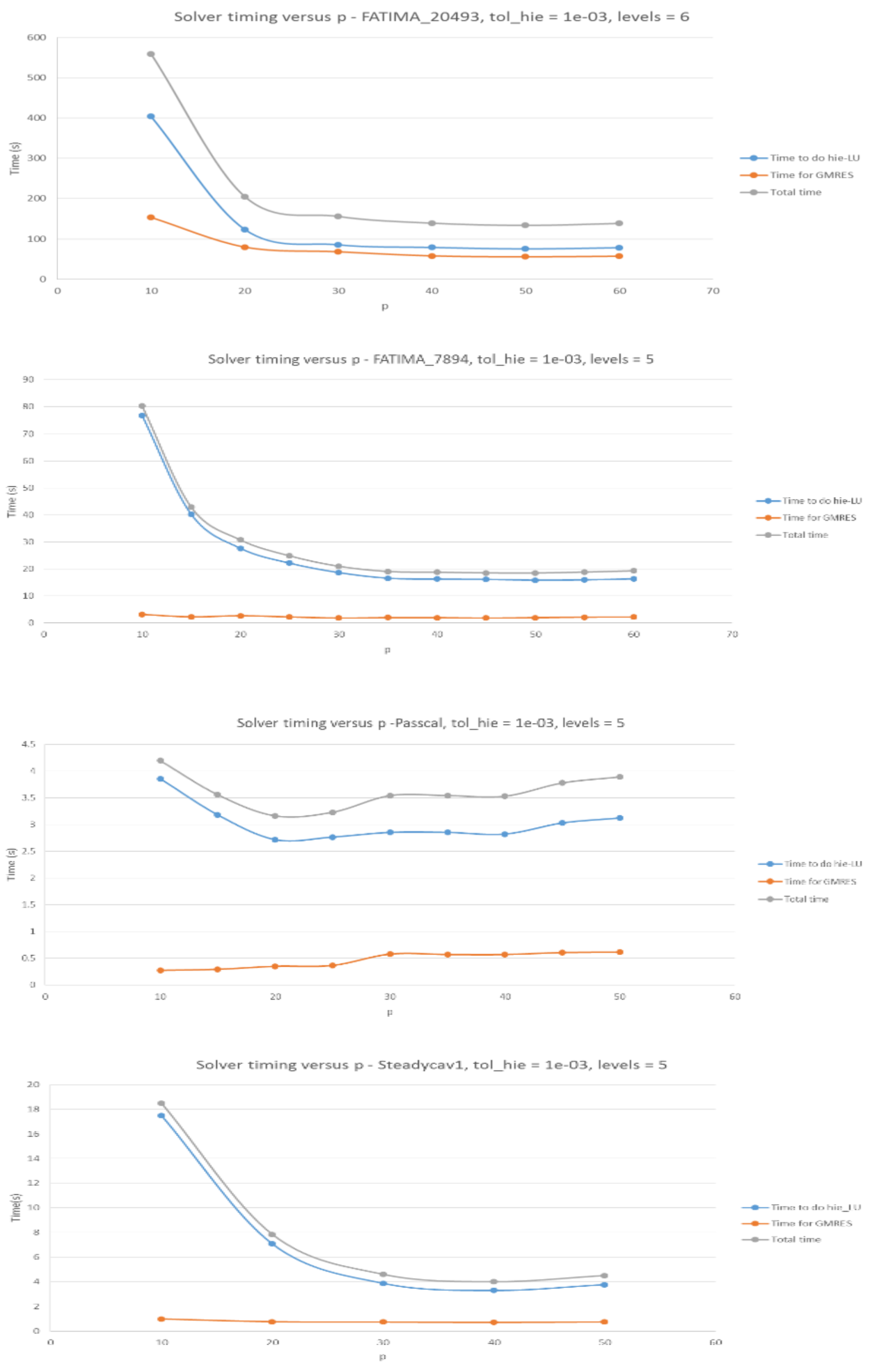


Figure 41 Solver timings versus rank p for the test matrices

This threshold value of p is observed to be about 30 for FATIMA_20493 and FATIMA_7894, 20 for PASSCAL, and 30 for Steadycav1.

6.3.2 Results based on parallel computations

The previous section has demonstrated the success of the hierarchical-LU preconditioner when operated sequentially. In this section, the performance of the hierarchical-LU preconditioner in parallel is evaluated. First, the baseline results for block Jacobi preconditioner when OpenMP is enabled is shown in Table 21.

Matrix	nrhs	Block-size for Block Jacobi Preconditioner	GMRES		IDR(s)	
			Wall clock time (s)	# iters	Wall clock time (s)	# iters
FATIMA_20493	1	4000 for GMRES 1708 for IDR(s)	87.62	103	80.45	260
	7	4000	211.49	103	220.48	112
FATIMA_7894	1	1000	6.36	121	6.55	133
	7	1000	25.74	121	28.18	136
PASSCAL	1	500	0.72	91	0.76	96
STEADYCAV1 (representing all steadycav matrices)	1	500	0.57	61	0.58	62

Table 21 Baseline results using block Jacobi preconditioner – Parallel. Results that are better than those obtained with hierarchical-LU preconditioner are highlighted in green

Matrix	Nrhs	Variables for hie_LU	GMRES		IDR(s)	
			Wall clock time (s)	# iters	Wall clock time (s)	# iters
FATIMA_20493	1	Tol: 1e-03 b = 200 p = 50	47.48	51	50.14	55
	7	Tol: 1e-04 b = 100 p = 30	108.96	30	116.20	32
FATIMA_7894	1	Tol: 1e-03 b = 200 p = 50	6.44	11	6.57	11
	7	Tol: 1e-03 b = 200 p = 40	9.17	11	9.74	11
PASSCAL	1	Tol: 1e-03 b = 100 p = 20	1.42	11	1.49	11
STEADYCAV1 (representing all steadycav matrices)	1	Tol: 1e-02 b = 100 p = 30	1.934	32	2.12	37

Table 22 Best results obtained using hierarchical-LU preconditioner - Parallel. Results that are better than those obtained with block Jacobi preconditioner are highlighted in green

The best solve time attained using hierarchical-LU decomposition with OpenMP is shown in the Table 22. For both Tables 21 and 22, the better solve times attained when comparing the block Jacobi preconditioner with the hierarchical-LU preconditioner, are highlighted in green.

In parallel, the hierarchical-LU preconditioner still outperforms the block Jacobi preconditioner for the large size matrix FATIMA_20493. The time required to solve the system using hierarchical-LU preconditioner is still about 40% lower than block Jacobi for the case with 1 RHS. The time taken to solve the FATIMA_7894 system in parallel with 1 RHS is comparable between the two methods.

With multiple RHS, the benefit of the hierarchical-LU preconditioner over block Jacobi is even more pronounce. For the FATIMA_20394, the improvement is about 48%, while that for FATIMA_7894 is about 65%. The reason for the sharp drop in time for FATIMA_7894 is due to the fact that the hierarchical-LU preconditioner conditions the system very well. The number of iterations required for convergence for FATIMA_7894 is only 11, as compared to the 121 iterations required for block Jacobi preconditioner. This results in significantly less dense matrix-matrix multiplication required, hence, the large improvement of 65%.

For the smaller matrices, just like in the sequential case, block Jacobi preconditioner performs better than hierarchical-LU preconditioner. The advantage of the block Jacobi preconditioner in parallel is more obvious due to its near-optimal speedup and the non-optimal speedup for the hierarchical-LU preconditioner. Even so, because of the much lower time required to solve the small system as compared to the large system, this slight setback on the hierarchical-LU preconditioner is not substantial.

The complete results recorded when the program is ran with OpenMP for different $tol\_hie, b\ and\ p$ are shown in Tables 23 to 28. The column **speedup** is measured in preconditioner construct time (time to construct the hierarchical LU preconditioner sequentially/time to construct the hierarchical-LU preconditioner in parallel). The same trends as the results computed sequentially can be observed. The sub-sections below discuss the results unique to parallel implementation, namely speedup and the results obtained for systems with multiple RHS.

6.3.2.1 *Comments on speedup*

The speedup attained for the construction of the hierarchical-LU preconditioner ranges from about 1.5 to about 3.6. The average speedup attained for FATIMA_20493, FATIMA_7894 and PASSCAL matrices ranges from about 2.7 to 3.0. This is as expected, since some parts of the hierarchical LU decomposition algorithm have to run sequentially. The lower end of the speedup occurs mostly in the test cases involving Steadycav1 matrices. This indicates that the load balancing with the recursive parallelism is not good with the Steadycav1 matrices. A closer look at the sparsity diagram of the final inadmissible matrix for Steadycav1 as compared to the other matrices explains this bad load balancing. An example with $b = 100$, $p = 30$ and $tol\_hie = 1e - 4$ is shown in Figure 42 below. Recall that the hie-LU code is parallelized such that the main diagonals have to be dealt with in a sequential way. With the main diagonal almost completely inadmissible, this means that more work have to be performed sequentially for Steadycav1 matrix. This explains the low speedup observed.

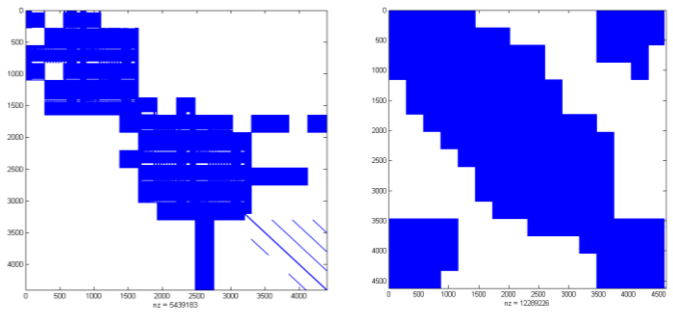


Figure 42 Sparsity diagram of the N_{levels} for Passcal (left) and Steadycav1(right) with b=100, p=30, tol_hie=1e-4.

6.3.2.2 *Comment on results for multiple RHS.*

An interesting observation can be made when looking at the results for multiple RHS. While higher tolerance ($tol_{hie} = 1e - 04$) does not seem to work well when there is only one RHS, this setting performs well when applied to a system with multiple RHS. The reason for this is that the matrix-matrix multiplication is significantly more expensive than matrix-vector multiplication. Therefore, lowering the number of iterations drives the time down by a big amount, which dominates over the time increase to construct the preconditioner due to the higher tolerance setting. It can therefore be concluded that with multiple RHS, one can afford a longer time to construct the preconditioner if the system turns out to be better conditioned such that the number of iterations is reduced by a significant amount.

To sum up Section 6, it has been shown that hierarchical-LU preconditioner works very well for the test matrices. By exploiting the hierarchical structure of these matrices, the construction of the hierarchical-LU preconditioner can be done relatively cheaply with almost linear complexity. Test results have shown that the hierarchical-LU preconditioner conditions the system very well, reducing the number of iterations significantly more than block Jacobi preconditioner with reasonable block size. Therefore, the use of hierarchical-LU preconditioner over block Jacobi preconditioner is definitely recommended.

FATIMA_20493

Tol_hie	b	p	GMRES							IDR(s)							Speedup
			Wall clock time (s)					#iter	Rel error	Wall clock time (s)					#iter	Rel error	
			Prec const	Prec apply	Matvec	Solve	Total			Prec const	Prec apply	Matvec	Solve	Total			
1.00E-02	100	10	10.13					>500		10.75	54.61	113.68	170.28	181.56	420	1.65E-05	2.73
		20	8.70					>500		9.76	39.76	98.21	139.71	149.92	371	8.07E-06	2.80
		30	9.05					>500		9.77	43.40	112.05	157.39	167.82	421	1.30E-05	2.81
		40	10.74					>500		10.67					>500		2.73
		50	22.73					>500		26.71					>500		1.74
	200	10	23.08	22.33	37.34	60.60	83.83	142	2.06E-05	23.58	25.47	43.02	69.28	93.31	162	8.03E-06	3.06
		20	15.60	25.40	47.69	74.59	90.40	181	6.12E-06	15.84	31.23	59.17	91.47	107.85	223	2.59E-06	2.75
		30	14.43	47.50	93.18	144.17	158.96	354	4.85E-06	14.67	33.74	67.07	101.99	117.05	252	4.00E-06	2.54
		40	14.31					>500		14.20	47.09	93.19	141.91	156.86	352	9.42E-07	2.46
		50	14.25					>500		14.04	55.68	114.29	171.95	186.55	431	3.43E-06	2.39
	600	10	48.93	27.56	27.74	55.82	105.14	105	6.54E-06	49.09	31.10	32.20	63.90	113.42	121	2.71E-06	2.51
		20	34.54	33.13	36.24	70.25	104.93	138	4.69E-06	34.45	38.07	42.47	81.32	115.89	160	1.24E-06	2.12
		30	31.88	29.69	34.57	65.05	97.11	130	1.32E-06	32.07	33.77	39.37	73.85	106.09	145	6.38E-07	1.91
		40	31.16	31.16	36.06	68.09	99.77	137	1.53E-06	31.23	35.71	41.61	78.08	109.55	156	7.10E-07	1.87
		50	29.86	31.58	36.97	69.45	99.60	140	4.49E-06	29.77	35.79	43.18	79.75	110.10	160	2.45E-06	1.79
		60	29.29	33.44	39.51	73.98	103.61	150	1.24E-06	29.37	39.98	48.41	89.27	118.96	179	1.77E-06	1.84
1.00E-03	100	10	42.01					>500		43.44					>500		3.25
		20	23.03	14.86	24.86	40.16	63.64	95	2.86E-06	23.35	16.92	29.10	46.56	70.39	109	7.43E-07	3.15
		30	20.84	9.01	17.86	27.10	48.88	68	3.27E-07	21.51	10.46	20.07	30.92	53.46	75	1.30E-07	3.16
		40	22.09	8.66	17.33	26.26	49.49	66	5.84E-07	22.03	9.83	19.95	30.17	53.10	73	2.21E-07	3.21
		50	45.08	7.92	28.73	39.16	85.68	61	5.52E-07	66.19	8.92	36.96	46.28	113.96	68	1.99E-07	1.72
	200	10	113.45	29.14	31.19	60.99	174.64	119	2.08E-05	113.78	32.77	36.29	69.73	183.73	137	5.86E-06	3.58
		20	37.86	12.81	18.29	31.34	69.47	70	1.99E-06	38.42	14.57	20.77	35.75	74.45	78	8.32E-07	3.33
		30	27.91	10.11	16.25	26.56	54.80	62	5.84E-07	28.05	11.37	18.79	30.53	58.92	69	5.20E-07	3.14
		40	26.33	8.34	13.88	22.37	49.19	53	1.84E-07	26.42	9.17	15.41	24.90	51.78	57	1.84E-07	3.09
		50	25.32	7.73	13.46	21.33	47.48	51	1.22E-07	23.86	8.59	14.96	25.44	50.14	55	1.80E-07	3.08
		60	26.85	8.14	13.82	22.10	49.65	52	2.21E-07	26.84	8.96	15.09	24.36	51.86	56	1.64E-07	3.03
	600	10	285.75	21.15	15.34	36.71	325.68	57	3.63E-06	285.20	23.48	16.58	40.41	328.96	62	1.40E-06	3.37
		20	88.83	13.86	12.30	26.27	115.28	47	6.47E-07	89.02	14.39	13.38	28.05	117.30	49	4.33E-07	3.11
		30	59.89	14.93	14.77	29.86	90.26	56	1.66E-06	59.57	16.69	16.72	33.75	93.86	62	4.70E-07	2.74
		40	47.73	13.94	13.92	28.01	76.01	53	7.22E-08	48.18	15.16	15.87	31.35	80.08	58	3.06E-08	2.54
		50	41.50	12.58	13.18	25.90	67.68	50	1.28E-07	40.97	13.72	14.87	28.89	70.18	54	6.58E-08	2.24

		60	41.54	12.34	13.18	25.65	67.83	50	1.03E-06	41.16	13.76	14.94	29.01	70.54	55	4.17E-07	2.23
1.00E -04	100	20	62.73	10.20	13.01	23.64	87.13	48	3.53E-07	63.87	11.15	14.20	25.63	90.31	52	8.84E-08	3.43
		30	58.21	5.89	7.88	13.83	73.00	30	1.34E-08	57.64	6.58	8.81	15.59	73.93	32	8.20E-09	3.34
		40	69.32	5.88	15.38	21.63	91.84	31	2.28E-08	70.48	6.66	15.67	22.56	94.27	33	9.93E-09	3.16
		50	134.27	6.31	26.39	32.81	168.21	33	1.29E-08	121.08	6.63	29.21	36.08	158.60	34	3.85E-08	1.97
	200	10	311.85	16.71	19.75	36.67	351.87	54	9.13E-07	304.73	18.52	17.70	36.54	344.73	58	1.34E-06	3.56
		20	104.28	10.40	11.55	22.05	126.94	44	2.72E-07	103.88	10.83	12.53	23.62	127.86	46	3.46E-07	3.56
		30	85.41	6.40	7.68	14.13	99.91	29	3.29E-08	85.34	7.18	8.58	15.96	101.74	31	3.00E-08	3.53
		40	79.72	5.96	7.30	13.31	93.48	28	1.75E-08	79.59	6.44	8.03	14.66	94.82	29	2.78E-08	3.51
		50	72.04	6.00	7.66	13.71	86.31	29	1.92E-08	71.71	6.80	8.88	15.88	88.19	32	6.09E-09	3.44
		60	72.58	6.11	7.91	14.07	87.61	30	2.47E-08	72.63	6.55	8.58	15.33	88.97	31	1.92E-08	3.37
	600	20	208.03	11.96	8.87	20.89	232.11	34	2.97E-07	207.93	12.52	9.58	22.32	230.47	35	2.52E-07	3.35
		30	121.90	9.46	7.89	17.39	139.81	30	1.62E-07	122.47	10.47	8.79	19.47	142.18	32	9.26E-08	3.25
		40	106.68	8.84	7.62	16.51	124.02	29	1.39E-08	106.12	9.52	8.26	17.98	124.37	30	3.22E-08	3.17
		50	91.59	10.48	9.12	19.66	111.88	35	9.54E-08	91.83	11.29	10.14	21.66	114.10	37	3.43E-08	3.08
		60	87.93	10.64	9.59	20.30	108.87	36	1.76E-07	87.92	11.47	10.32	22.03	110.35	38	1.97E-07	3.03

Table 23 Results using hierarchy-LU preconditioner for FATIMA_20493 with nrhs=1. The best results obtained for GMRES and IDR(s) are highlighted in green

FATIMA_20493																
Tol_hie	b	p	GMRES							IDR(s)						
			Wall clock time (s)					#iter	Rel error	Wall clock time (s)					#iter	Rel error
			Prec const	Prec apply	Matvec	Solve	Total			Prec const	Prec apply	Matvec	Solve	Total		
1.00E-02	200	10	23.46	75.80	154.43	236.55	260.16	142	2.06E-05	23.25	86.74	175.15	267.79	291.22	161	1.58E-05
		20	15.85	86.57	196.66	293.22	309.63	181	6.12E-06	15.98	103.19	235.95	346.95	363.18	217	4.31E-06
		30	14.52	168.71	385.26	573.05	588.00	354	4.85E-06	14.81	121.70	278.48	409.28	424.44	256	2.33E-06
	600	10	49.23	92.11	113.60	209.39	258.81	105	6.54E-06	49.38	92.53	114.49	211.00	260.52	105	6.54E-06
		20	34.57	109.45	150.42	265.88	300.95	138	4.69E-06	34.49	125.38	174.72	306.01	340.68	160	1.91E-06
		30	31.99	97.12	141.28	243.78	276.28	130	1.32E-06	32.05	110.62	158.89	274.91	307.18	145	3.87E-07
		40	30.94	102.06	148.61	256.58	287.80	137	1.53E-06	31.29	121.23	168.26	295.18	326.75	155	6.10E-07
		50	29.83	100.55	151.65	258.36	288.84	140	4.49E-06	29.62	124.85	180.35	311.26	341.19	165	2.35E-06
		60	29.58	110.67	163.02	280.70	310.70	150	1.24E-06	29.81	130.87	190.04	327.34	357.53	175	2.61E-06
1.00E-03	100	20	23.88	41.59	102.56	147.24	171.60	95	2.86E-06	22.77	46.83	115.75	166.63	189.88	106	1.19E-06
		30	21.72	29.04	73.27	104.37	126.81	68	3.27E-07	21.83	32.98	83.04	118.98	141.85	76	1.16E-07
		40	23.24	28.26	71.01	101.28	125.44	66	5.84E-07	27.38	31.26	79.01	113.09	141.34	72	2.63E-07
	200	20	38.66	43.91	76.84	122.60	161.54	70	1.99E-06	38.27	47.58	82.00	132.54	171.13	75	7.20E-07
		30	27.98	35.51	67.08	104.12	132.48	62	5.84E-07	28.00	40.23	76.27	119.26	147.93	69	8.58E-07
		40	26.13	29.98	57.08	88.26	114.84	53	1.84E-07	26.22	32.70	63.02	98.08	124.77	57	1.77E-07
		50	25.42	28.99	54.83	84.97	110.94	51	1.22E-07	25.25	32.74	62.59	97.59	123.69	57	3.04E-08
		60	26.66	29.98	56.35	87.51	114.86	52	2.21E-07	26.90	33.76	62.78	98.82	126.68	57	2.66E-07
	600	20	89.04	48.75	52.76	102.55	191.80	47	6.47E-07	89.75	50.58	52.91	105.59	195.51	48	5.13E-07
		30	59.31	51.89	60.55	113.76	173.31	56	1.66E-06	59.60	58.40	67.82	128.75	188.56	62	3.15E-07
		40	47.86	46.33	57.60	105.16	153.31	53	7.22E-08	48.19	52.71	63.44	118.55	166.99	58	8.97E-08
		50	41.19	42.69	54.31	98.13	139.63	50	1.28E-07	41.36	45.84	59.45	107.56	149.49	54	7.12E-08
		60	41.27	41.83	54.28	97.24	138.88	50	1.03E-06	41.48	48.84	62.30	113.47	155.60	56	8.15E-08
1.00E-04	100	20	62.16	29.92	51.81	83.10	145.80	48	3.53E-07	63.59	32.86	57.20	92.22	156.58	52	1.67E-07
		30	56.85	18.12	32.33	51.11	108.96	30	1.34E-08	57.95	20.08	35.68	57.23	116.20	32	2.15E-08
		40	75.91	19.02	39.49	59.31	136.42	31	2.28E-08	73.26	21.22	41.20	64.13	138.63	33	1.04E-08
		50	135.83	20.56	52.71	74.31	211.56	33	1.29E-08	122.88	22.16	48.41	72.40	196.39	34	3.08E-08
	200	20	103.81	36.05	47.54	84.53	188.65	44	2.72E-07	104.04	38.93	50.81	91.76	196.12	46	2.64E-07
		30	85.25	22.83	31.10	54.24	139.88	29	3.29E-08	84.87	24.46	33.56	59.46	145.06	30	5.13E-08
		40	79.58	21.66	30.13	52.08	132.16	28	1.75E-08	79.46	24.06	33.64	59.13	139.08	30	1.04E-08
		50	71.76	21.84	31.49	53.64	126.00	29	1.92E-08	72.08	24.41	34.58	60.44	133.43	31	3.58E-08
		60	72.48	22.67	32.25	55.56	129.06	30	2.47E-08	72.76	25.22	35.81	62.55	136.25	32	8.94E-09
	600	20	207.68	43.93	37.13	84.53	292.71	34	2.97E-07	207.65	49.03	41.14	91.86	299.70	37	1.74E-08

		30	122.24	33.98	32.87	67.19	189.66	30	1.62E-07	122.08	36.54	34.76	72.77	195.08	31	7.45E-08
		40	106.19	31.78	31.33	63.42	170.19	29	1.39E-08	106.36	35.19	34.80	71.46	178.08	31	9.68E-09
		50	91.77	37.29	37.64	75.37	167.47	35	9.54E-08	91.25	42.78	43.60	88.15	179.72	39	1.85E-08
		60	88.09	38.95	39.03	78.74	167.51	36	1.76E-07	88.19	42.01	42.23	85.97	174.56	38	8.54E-08

Table 24 Results using hierarchy-LU preconditioner for FATIMA_20493 with nrhs=7. The best results obtained for GMRES and IDR(s) are highlighted in green

FATIMA_7894																	
Tol_hie	b	p	GMRES							IDR(s)							Speedup
			Wall clock time (s)					#iter	Rel error	Wall clock time (s)					#iter	Rel error	
			Prec const	Prec apply	Matvec	Solve	Total			Prec const	Prec apply	Matvec	Solve	Total			
1.00E-02	100	10	6.30	9.84	7.04	17.38	23.99	173	9.19E-05	6.34	13.86	9.42	23.56	29.92	243	3.73E-05	3.09
		20	3.49	16.89	15.92	34.11	37.64	395	1.40E-05	3.43	13.17	11.57	25.09	28.56	305	7.94E-06	2.71
		30	3.06					>500		3.08							2.78
		40	3.40	12.90	14.46	28.42	31.90	355	2.22E-06	3.37	10.90	11.28	22.52	25.96	301	1.41E-06	2.76
		50	3.62	11.85	13.96	26.82	30.55	343	1.67E-06	3.69	9.91	10.92	21.14	24.93	278	1.25E-06	2.88
		60	4.09	11.82	13.39	26.15	30.36	328	4.64E-06	4.09	10.12	10.76	21.20	25.41	277	7.50E-07	2.87
	200	10	8.87	2.22	1.65	3.90	12.78	41	1.11E-07	8.90	2.49	1.86	4.41	13.32	44	6.89E-08	3.32
		20	4.33	2.29	2.15	4.50	8.85	53	1.16E-06	4.38	2.72	2.55	5.35	9.75	61	1.00E-07	2.89
		30	3.66	2.35	2.33	4.74	8.44	57	4.20E-07	3.65	2.87	2.91	5.86	9.53	68	2.93E-07	2.63
		40	3.56	2.58	2.54	5.19	8.79	65	5.41E-07	3.59	3.06	3.08	6.23	9.86	75	6.44E-07	2.66
		50	3.57	3.78	4.03	7.98	11.60	98	6.85E-07	3.52	4.19	4.69	9.02	12.59	109	4.87E-07	2.61
		60	3.77	3.95	4.34	8.48	12.30	105	5.23E-07	3.64	4.63	5.20	9.97	13.67	121	2.42E-07	2.62
	250	10	14.36	1.80	0.92	2.74	17.11	23	1.16E-08	14.11	1.70	1.05	2.78	16.90	24	1.12E-08	2.81
		20	8.67	2.09	1.20	3.31	11.99	30	1.92E-08	8.50	2.34	1.39	3.78	12.29	32	4.72E-08	2.27
		30	7.09	2.07	1.23	3.33	10.43	31	1.52E-07	7.13	1.95	1.41	3.41	10.55	34	8.04E-08	1.89
		40	6.98	2.17	1.36	3.56	10.55	33	8.79E-08	6.96	2.45	1.53	4.03	11.01	36	2.51E-08	1.88
		50	6.82	2.68	1.71	4.42	11.26	41	9.98E-08	6.84	2.37	1.79	4.22	11.08	43	9.29E-08	1.84
		60	6.72	2.66	1.62	4.31	11.06	40	1.30E-07	6.73	2.44	1.80	4.30	11.06	44	2.50E-08	1.84
1.00E-03	100	10	17.01	4.20	2.12	6.37	23.40	54	9.71E-08	17.09	4.64	2.25	6.96	24.07	58	1.62E-07	3.41
		20	7.41	1.41	0.97	2.39	9.84	25	2.59E-08	7.29	1.66	1.12	2.82	10.15	27	6.91E-09	3.19
		30	5.84	1.29	1.00	2.31	8.21	26	1.09E-08	5.79	1.28	1.00	2.29	8.14	26	1.09E-08	3.17
		40	5.55	1.17	0.92	2.10	7.73	24	9.54E-09	5.51	1.32	1.04	2.40	7.98	25	2.05E-08	3.19
		50	5.83	1.11	0.92	2.05	7.98	24	1.80E-08	5.88	1.29	1.06	2.39	8.37	26	8.13E-09	3.21
		60	6.35	1.15	0.92	2.09	8.56	24	8.27E-09	6.35	1.29	1.07	2.40	8.87	25	1.92E-08	3.28
	200	10	21.74	0.99	0.52	1.51	23.26	14	8.07E-08	21.85	1.19	0.65	1.86	23.73	15	6.92E-08	3.57
		20	8.43	0.76	0.53	1.29	9.74	14	3.47E-09	8.42	0.92	0.73	1.67	10.12	15	1.16E-09	3.34
		30	6.10	0.52	0.39	0.91	7.04	10	1.77E-08	6.13	0.64	0.52	1.18	7.33	11	3.49E-09	3.16
		40	5.53	0.54	0.42	0.96	6.53	11	9.19E-10	5.51	0.63	0.53	1.18	6.74	11	2.09E-09	3.06
		50	5.44	0.53	0.42	0.96	6.44	11	1.81E-09	5.39	0.60	0.51	1.13	6.57	11	3.61E-09	3.07
		60	5.65	0.56	0.47	1.04	6.75	12	5.70E-09	5.56	0.67	0.58	1.28	6.90	13	2.77E-09	3.06
	250	10	30.67	0.86	0.36	1.23	31.91	9	2.47E-10	30.74	1.01	0.45	1.48	32.22	9	2.31E-09	3.36
		20	12.95	0.62	0.28	0.90	13.86	7	1.10E-08	13.02	0.62	0.35	0.99	14.02	7	6.15E-09	2.93

		30	9.62	0.64	0.33	0.97	10.61	8	3.11E-09	9.57	0.75	0.41	1.18	10.76	8	1.23E-08	2.55
		40	8.80	0.69	0.39	1.08	9.90	9	1.36E-09	8.68	0.79	0.41	1.22	9.92	9	3.08E-09	2.27
		50	8.51	0.48	0.36	0.84	9.37	9	5.76E-09	8.48	0.61	0.45	1.08	9.58	10	1.38E-09	2.14
		60	8.46	0.74	0.41	1.15	9.64	10	3.29E-09	8.53	0.71	0.48	1.21	9.77	10	1.28E-08	2.23
1.00E -04	100	10	26.69	2.32	0.89	3.23	29.95	24	1.14E-08	26.72	2.64	0.99	3.67	30.42	25	9.39E-09	3.51
		20	14.56	2.16	1.19	3.36	17.97	30	2.39E-08	14.50	2.39	1.31	3.74	18.28	32	5.23E-08	3.41
		30	11.33	2.30	1.49	3.82	15.20	37	3.72E-08	11.40	2.54	1.65	4.24	15.70	40	2.31E-08	3.40
		40	10.98	1.52	1.01	2.54	13.60	26	2.86E-08	11.07	1.77	1.14	2.95	14.10	27	7.67E-08	3.37
		50	11.38	1.06	0.74	1.80	13.28	19	2.39E-08	11.54	1.25	0.86	2.15	13.78	21	9.76E-09	3.34
		60	12.01	0.55	0.39	0.94	13.06	10	3.43E-09	12.03	0.66	0.46	1.14	13.29	10	9.19E-09	3.41
	200	10	33.54	0.62	0.26	0.88	34.43	7	1.47E-09	33.44	0.74	0.37	1.12	34.57	7	2.96E-09	3.61
		20	15.25	0.83	0.52	1.36	16.62	12	1.65E-08	15.33	0.95	0.65	1.62	16.98	13	7.23E-09	3.51
		30	10.94	0.83	0.53	1.37	12.34	14	7.13E-09	10.82	0.82	0.69	1.54	12.39	15	3.24E-09	3.37
		40	9.81	0.72	0.50	1.23	11.08	13	3.25E-09	9.68	0.87	0.66	1.55	11.27	14	1.42E-09	3.34
		50	9.45	0.60	0.43	1.03	10.53	11	6.42E-08	9.45	0.73	0.56	1.31	10.81	12	1.38E-08	3.33
		60	9.44	0.45	0.31	0.76	10.26	8	4.31E-10	9.40	0.54	0.39	0.95	10.40	8	1.90E-09	3.31
	250	10	44.36	0.55	0.19	0.74	45.11	5	3.93E-10	44.43	0.63	0.27	0.92	45.35	5	1.89E-09	3.46
		20	21.25	0.74	0.32	1.05	22.31	8	3.65E-08	21.34	0.83	0.43	1.28	22.63	9	2.47E-09	3.22
		30	14.62	0.85	0.38	1.24	15.87	10	2.11E-09	14.86	0.84	0.50	1.36	16.24	10	3.20E-09	3.04
		40	13.61	0.61	0.28	0.89	14.52	7	3.41E-10	13.39	0.63	0.36	1.01	14.42	7	1.06E-09	2.87
		50	12.33	0.67	0.32	1.00	13.35	8	6.03E-09	12.24	0.74	0.42	1.18	13.44	9	4.87E-10	2.81
		60	11.75	0.44	0.20	0.64	12.41	5	1.13E-08	11.70	0.54	0.31	0.87	12.60	6	7.72E-10	2.75

Table 25 Results using hierarchy-LU preconditioner for FATIMA_7894 with nrhs=1. The best results obtained for GMRES and IDR(s) are highlighted in green

FATIMA_7894																
Tol_hie	B	p	GMRES							IDR(s)						
			Wall clock time (s)					#iter	Rel error	Wall clock time (s)					#iter	Rel error
			Prec const	Prec apply	Matvec	Solve	Total			Prec const	Prec apply	Matvec	Solve	Total		
1.00E-02	200	10	8.77	7.30	6.60	14.12	22.91	41	1.11E-07	8.82	8.25	7.42	16.07	24.91	45	4.62E-08
		20	4.29	7.58	8.49	16.43	20.74	53	1.16E-06	4.41	8.79	10.06	19.36	23.79	59	9.72E-07
		30	3.60	7.36	9.28	17.06	20.69	57	4.20E-07	3.68	9.63	11.25	21.43	25.17	67	1.76E-07
		40	3.51	9.09	10.77	20.39	23.95	65	5.41E-07	3.57	10.49	12.40	23.50	27.15	74	3.44E-07
		50	3.57	13.70	16.01	30.86	34.48	98	6.85E-07	3.51	15.81	18.62	35.31	38.87	112	5.86E-07
		60	3.71	14.71	17.40	33.45	37.22	105	5.23E-07	3.74	16.86	19.90	37.70	41.51	119	3.49E-07
	250	10	14.34	6.02	3.78	9.88	24.22	23	1.16E-08	14.48	6.88	4.41	11.55	26.04	25	3.35E-09
		20	8.58	6.91	4.98	12.02	20.60	30	1.92E-08	8.60	7.67	5.52	13.49	22.10	32	2.68E-08
		30	7.10	6.69	5.29	12.12	19.24	31	1.52E-07	7.13	7.55	5.90	13.77	20.91	34	1.21E-07
		40	7.00	7.09	5.45	12.69	19.71	33	8.79E-08	7.01	7.84	6.54	14.72	21.75	36	7.11E-08
		50	6.89	8.60	6.70	15.53	22.44	41	9.98E-08	6.85	9.41	7.46	17.27	24.15	44	7.50E-08
		60	6.75	7.98	6.55	14.75	21.52	40	1.30E-07	6.72	9.64	7.49	17.53	24.28	44	3.39E-08
1.00E-03	100	10	17.05	11.03	8.87	20.28	37.35	54	9.71E-08	17.14	12.13	9.58	22.20	39.36	58	7.35E-08
		20	7.35	3.70	4.11	7.90	15.30	25	2.59E-08	7.40	4.05	4.51	8.82	16.26	26	1.57E-08
		30	5.92	3.61	4.29	8.00	13.99	26	1.09E-08	5.86	4.15	4.81	9.23	15.15	28	5.60E-09
		40	5.58	3.35	3.98	7.41	13.07	24	9.54E-09	5.52	3.83	4.43	8.52	14.12	26	3.47E-09
		50	5.95	3.45	4.07	7.61	13.66	24	1.80E-08	5.85	3.99	4.47	8.72	14.67	26	3.49E-09
		60	6.42	3.57	3.99	7.65	14.19	24	8.27E-09	6.43	3.78	4.17	8.19	14.74	24	2.25E-08
	200	10	21.92	3.31	2.26	5.61	27.54	14	8.07E-08	21.88	3.90	2.59	6.67	28.56	15	2.32E-08
		20	8.47	2.57	2.31	4.92	13.40	14	3.47E-09	8.52	2.97	2.66	5.81	14.35	15	8.22E-10
		30	6.15	1.71	1.66	3.39	9.57	10	1.77E-08	6.13	2.09	2.01	4.25	10.41	11	5.80E-09
		40	5.46	1.81	1.84	3.68	9.17	11	9.19E-10	5.51	2.05	1.99	4.19	9.74	11	5.43E-09
		50	5.47	1.85	1.82	3.69	9.21	11	1.81E-09	5.46	2.09	2.01	4.25	9.75	11	3.82E-09
		60	5.60	2.03	1.98	4.04	9.69	12	5.70E-09	5.62	2.44	2.32	4.92	10.60	13	5.59E-09
	250	10	30.76	3.00	1.44	4.46	35.23	9	2.47E-10	30.76	3.40	1.65	5.18	35.95	9	1.06E-09
		20	13.11	2.03	1.14	3.19	16.31	7	1.10E-08	13.00	2.61	1.49	4.23	17.24	8	8.51E-10
		30	9.43	2.08	1.32	3.42	12.86	8	3.11E-09	9.58	2.39	1.51	4.02	13.61	8	9.38E-09
		40	8.88	2.26	1.48	3.77	12.66	9	1.36E-09	8.80	2.56	1.70	4.39	13.21	9	3.53E-09
		50	8.40	2.05	1.50	3.57	12.00	9	5.76E-09	8.42	2.74	1.87	4.74	13.19	10	9.40E-10
		60	8.53	2.45	1.63	4.10	12.66	10	3.29E-09	8.52	2.98	2.00	5.13	13.68	11	1.12E-09
1.00E-04	100	10	26.72	6.08	3.87	10.05	36.79	24	1.14E-08	26.89	6.58	4.17	11.01	37.92	25	9.89E-09
		20	14.65	5.60	4.91	10.64	25.33	30	2.39E-08	14.60	6.29	5.42	12.01	26.65	32	3.44E-08

		30	11.38	6.17	6.08	12.43	23.87	37	3.72E-08	11.36	7.00	6.95	14.32	25.74	41	1.05E-08
		40	10.91	4.28	4.35	8.73	19.72	26	2.86E-08	10.98	4.55	4.52	9.34	20.40	26	8.50E-08
		50	11.51	3.17	3.22	6.46	18.07	19	2.39E-08	11.30	3.71	3.58	7.51	18.90	21	3.30E-09
		60	12.02	1.77	1.63	3.42	15.56	10	3.43E-09	11.98	2.05	1.84	4.04	16.15	10	1.44E-08
	200	10	33.57	2.06	1.14	3.22	36.80	7	1.47E-09	33.48	2.44	1.31	3.87	37.37	7	3.02E-09
		20	15.42	2.64	1.93	4.60	20.04	12	1.65E-08	15.34	3.16	2.40	5.73	21.09	13	4.40E-09
		30	10.86	2.83	2.41	5.27	16.17	14	7.13E-09	10.90	3.23	2.67	6.07	17.01	15	4.82E-09
		40	9.83	2.51	2.23	4.78	14.65	13	3.25E-09	9.85	2.69	2.35	5.20	15.10	13	8.55E-09
		50	9.56	2.09	1.83	3.95	13.56	11	6.42E-08	9.47	2.51	2.16	4.82	14.34	12	4.94E-09
		60	9.47	1.55	1.32	2.89	12.42	8	4.31E-10	9.48	1.80	1.48	3.42	12.95	8	1.48E-09
	250	10	44.45	1.94	0.82	2.77	47.23	5	3.93E-10	44.63	2.37	1.00	3.48	48.11	5	2.16E-09
		20	21.45	2.52	1.28	3.82	25.29	8	3.65E-08	21.45	3.13	1.63	4.89	26.35	9	1.11E-09
		30	14.78	2.85	1.63	4.50	19.30	10	2.11E-09	14.75	3.23	1.92	5.29	20.05	10	3.65E-09
		40	13.60	2.08	1.17	3.26	16.88	7	3.41E-10	13.45	2.48	1.37	3.97	17.44	7	1.96E-09
		50	12.32	2.28	1.37	3.67	16.02	8	6.03E-09	12.39	2.84	1.73	4.71	17.12	9	1.16E-09
		60	11.71	1.48	0.88	2.37	14.11	5	1.13E-08	11.65	2.08	1.18	3.37	15.05	6	8.89E-10

Table 26 Results using hierarchy-LU preconditioner for FATIMA_7894 with nrhs=7. The best results obtained for GMRES and IDR(s) are highlighted in green

PASSCAL																	
Tol_hie	b	P	GMRES							IDR(s)							Speedup
			Wall clock time (s)					#iter	Rel error	Wall clock time (s)					#iter	Rel error	
			Prec const	Prec apply	Matvec	Solve	Total			Prec const	Prec apply	Matvec	Solve	Total			
1.00E-02	100	10	0.97	0.44	0.13	0.58	1.55	23	4.62E-08	0.92	0.49	0.16	0.66	1.58	24	1.33E-08	2.46
		20	0.64	0.62	0.28	0.91	1.55	44	3.34E-08	0.62	0.63	0.29	0.94	1.56	46	2.38E-07	2.15
		30	0.66	0.54	0.30	0.85	1.52	47	1.01E-06	0.65	0.62	0.36	1.00	1.66	51	1.69E-07	2.22
		40	0.72	0.58	0.30	0.89	1.62	49	7.71E-07	0.73	0.68	0.33	1.03	1.77	53	3.33E-07	2.36
		50	0.73	0.46	0.34	0.81	1.55	50	7.72E-07	0.72	0.53	0.35	0.90	1.64	54	3.60E-07	2.55
	200	10	1.74	0.32	0.12	0.44	2.19	20	1.68E-07	1.71	0.40	0.15	0.56	2.28	22	2.42E-08	3.00
		20	0.94	0.53	0.23	0.76	1.71	38	9.54E-08	0.93	0.57	0.26	0.86	1.79	42	3.45E-07	2.33
		30	0.80	0.55	0.29	0.85	1.66	46	7.41E-07	0.80	0.62	0.31	0.94	1.74	50	2.57E-07	2.17
		40	0.76	0.58	0.29	0.88	1.65	48	1.14E-06	0.77	0.65	0.34	1.01	1.79	53	2.27E-07	2.23
		50	0.83	0.54	0.31	0.86	1.70	50	9.29E-07	0.79	0.60	0.34	0.96	1.76	55	2.36E-06	2.22
1.00E-03	100	10	1.41	0.22	0.05	0.27	1.69	8	4.87E-09	1.43	0.27	0.07	0.35	1.78	9	3.60E-10	2.82
		20	1.10	0.24	0.08	0.32	1.42	11	1.01E-08	1.10	0.29	0.08	0.38	1.49	11	1.14E-08	2.56
		30	1.17	0.29	0.12	0.42	1.59	19	7.42E-09	1.12	0.38	0.14	0.53	1.66	21	1.34E-09	2.56
		40	1.15	0.27	0.11	0.38	1.54	18	1.78E-08	1.12	0.30	0.12	0.44	1.57	19	6.54E-09	2.62
		50	1.22	0.28	0.13	0.41	1.64	20	1.21E-08	1.21	0.30	0.14	0.46	1.68	21	9.70E-09	2.68
	200	10	2.28	0.10	0.03	0.13	2.41	5	3.62E-10	2.28	0.13	0.04	0.18	2.45	5	6.93E-09	3.27
		20	1.67	0.20	0.07	0.27	1.94	11	6.88E-10	1.66	0.22	0.09	0.31	1.97	11	2.65E-09	2.95
		30	1.30	0.26	0.12	0.37	1.68	18	2.50E-08	1.31	0.30	0.13	0.44	1.75	20	1.40E-08	2.75
		40	1.17	0.26	0.11	0.38	1.55	19	8.70E-09	1.19	0.27	0.12	0.40	1.59	19	2.41E-08	2.52
		50	1.16	0.25	0.11	0.37	1.53	19	2.29E-08	1.14	0.28	0.13	0.42	1.57	20	2.58E-08	2.51
1.00E-04	100	10	2.05	0.20	0.04	0.25	2.30	7	3.20E-09	2.11	0.23	0.05	0.29	2.40	7	5.80E-09	3.00
		20	1.40	0.17	0.05	0.22	1.62	7	4.46E-09	1.36	0.20	0.05	0.26	1.63	7	1.61E-09	2.84
		30	1.46	0.17	0.05	0.21	1.68	8	2.27E-09	1.40	0.20	0.06	0.27	1.68	8	8.58E-09	2.65
		40	1.61	0.15	0.05	0.20	1.82	7	1.78E-10	1.60	0.17	0.05	0.23	1.84	7	4.19E-10	2.99
		50	1.70	0.13	0.05	0.18	1.90	7	3.12E-10	1.73	0.16	0.06	0.23	1.98	7	7.20E-10	2.92
	200	10	3.71	0.11	0.03	0.14	3.85	5	4.08E-09	3.73	0.17	0.05	0.22	3.95	6	3.52E-11	3.47
		20	2.15	0.08	0.02	0.11	2.26	4	8.12E-12	2.17	0.11	0.04	0.16	2.33	4	9.60E-12	3.22
		30	1.60	0.12	0.05	0.17	1.78	7	3.87E-09	1.61	0.17	0.06	0.23	1.84	8	1.01E-10	2.95
		40	1.47	0.12	0.04	0.16	1.64	7	3.68E-09	1.45	0.15	0.05	0.21	1.66	7	6.46E-09	2.85
		50	1.50	0.10	0.04	0.14	1.65	6	1.56E-09	1.49	0.12	0.05	0.18	1.67	6	1.04E-08	2.94

Table 27 Results using hierarchy-LU preconditioner for PASSCAL with nrhs=1. The best results obtained for GMRES and IDR(s) are highlighted in green

STEADYCAV1																	
Tol_hie	b	p	GMRES							IDR(s)							Speedup
			Wall clock time (s)					#iter	Rel error	Wall clock time (s)					#iter	Rel error	
			Prec const	Prec apply	Matvec	Solve	Total			Prec const	Prec apply	Matvec	Solve	Total			
1.00E-02	100	20	1.17	0.39	0.74	1.13	2.31	32	5.76E-05	1.18	0.46	0.88	1.35	2.53	37	3.48E-05	1.69
		30	0.92	0.26	0.74	1.00	1.93	32	9.35E-05	0.92	0.31	0.87	1.19	2.12	37	1.43E-06	1.77
		40	0.90	0.26	0.78	1.04	1.95	34	1.58E-05	0.90	0.31	0.91	1.22	2.13	38	1.98E-05	1.79
		50	1.07	0.32	0.78	1.11	2.20	34	1.37E-05	1.06	0.27	0.88	1.16	2.24	37	7.82E-05	1.80
	200	10	4.25	0.60	0.73	1.34	5.59	32	2.52E-05	4.23	0.69	0.85	1.54	5.77	36	2.54E-06	1.80
		20	1.52	0.40	0.67	1.07	2.60	29	3.45E-05	1.51	0.44	0.72	1.16	2.67	30	5.53E-06	1.55
		30	1.20	0.40	0.73	1.13	2.33	32	8.87E-05	1.19	0.47	0.85	1.33	2.53	36	4.83E-05	1.54
		40	1.06	0.42	0.73	1.16	2.23	32	2.36E-05	1.07	0.50	0.85	1.36	2.44	36	3.86E-05	1.58
		50	1.12	0.28	0.74	1.02	2.15	32	5.50E-05	1.11	0.44	0.83	1.28	2.39	35	3.77E-05	1.59
1.00E-03	100	10	7.53	0.36	0.48	0.84	8.37	21	3.66E-05	7.53	0.46	0.58	1.04	8.57	24	2.53E-05	2.33
		20	3.24	0.26	0.46	0.71	3.96	20	4.02E-06	3.21	0.30	0.53	0.83	4.05	22	2.12E-06	2.19
		30	2.40	0.23	0.50	0.73	3.15	22	1.41E-05	2.43	0.27	0.59	0.86	3.30	24	2.75E-06	1.60
		40	2.01	0.20	0.50	0.71	2.73	22	3.18E-05	2.00	0.25	0.58	0.83	2.84	24	5.85E-05	1.64
		50	2.26	0.19	0.52	0.72	2.99	23	1.12E-05	2.26	0.26	0.60	0.87	3.15	25	1.92E-05	1.67
	200	10	7.74	0.33	0.34	0.67	8.42	15	8.97E-06	7.75	0.41	0.42	0.83	8.58	17	2.59E-06	2.33
		20	4.10	0.36	0.46	0.83	4.93	20	6.35E-05	4.10	0.46	0.56	1.02	5.13	23	2.92E-06	2.09
		30	2.53	0.30	0.46	0.76	3.30	20	1.82E-05	2.52	0.37	0.56	0.93	3.46	23	2.05E-06	1.54
		40	1.89	0.27	0.46	0.73	2.63	20	7.27E-06	1.89	0.23	0.53	0.77	2.67	22	6.69E-05	1.57
		50	1.93	0.29	0.46	0.75	2.69	20	4.04E-05	1.93	0.25	0.54	0.79	2.73	22	4.23E-05	1.58
1.00E-04	100	10	8.47	0.26	0.30	0.55	9.02	13	4.78E-07	8.46	0.28	0.35	0.64	9.10	14	3.24E-06	2.44
		20	6.31	0.36	0.50	0.87	7.18	22	1.56E-05	6.28	0.45	0.63	1.09	7.38	26	2.21E-06	2.18
		30	4.98	0.35	0.55	0.90	5.89	24	5.53E-05	4.98	0.43	0.67	1.11	6.10	28	1.43E-05	2.27
		40	5.08	0.29	0.48	0.77	5.86	21	1.68E-05	5.04	0.33	0.58	0.92	5.97	24	4.42E-06	1.89
		50	4.79	0.22	0.39	0.61	5.41	17	9.41E-06	4.78	0.26	0.46	0.72	5.53	19	6.34E-06	1.96
	200	10	8.19	0.29	0.28	0.56	8.76	12	3.53E-06	8.15	0.27	0.33	0.60	8.75	13	9.52E-06	2.38
		20	6.55	0.33	0.43	0.76	7.31	17	6.95E-06	6.56	0.41	0.44	0.85	7.42	18	7.93E-07	2.23
		30	5.19	0.41	0.48	0.90	6.09	21	2.27E-05	5.19	0.54	0.63	1.17	6.37	26	1.26E-06	2.14
		40	4.35	0.26	0.41	0.67	5.03	18	1.37E-05	4.37	0.36	0.47	0.83	5.21	19	1.55E-05	1.89
		50	3.58	0.27	0.37	0.63	4.22	16	7.98E-07	3.59	0.25	0.44	0.70	4.30	18	2.68E-06	2.01
		60	3.41	0.26	0.34	0.61	4.03	15	4.53E-06	3.40	0.29	0.40	0.70	4.10	16	6.84E-07	2.06

Table 28 Results using hierarchy-LU preconditioner for STEADYCAV1 with nrhs=1. The best results obtained for GMRES and IDR(s) are highlighted in green

7 CONCLUSION

In this report, four strategies to improve the efficiency of the dense linear solver used in panel codes were explored. These are

1. The use of the IDR(s) solver instead of GMRES
2. The choice to use variable size block Jacobi preconditioner in suitable scenarios
3. Replacing dense matvec in the solver with hierarchical matvec
4. The use of hierarchical-LU preconditioner instead of block Jacobi preconditioner

With these, many tests were conducted with the test matrices. The best strategy, together with the timing attained, are summarized in the table below:

Test Matrix	nrhs	Strategy	Time(s)
FATIMA_20493	1	GMRES with Hierarchical-LU preconditioner OpenMP enabled	47.48
	7	GMRES with Hierarchical-LU preconditioner OpenMP enabled	108.96
FATIMA_7894	1	GMRES with block Jacobi Preconditioner OpenMP enabled	6.36
	7	GMRES with Hierarchical-LU preconditioner OpenMP enabled	9.17
PASSCAL	1	GMRES with block Jacobi Preconditioner OpenMP enabled	0.718
Steadycav1	1	GMRES with block Jacobi Preconditioner OpenMP enabled	0.565
Steadycav2	1	GMRES with block Jacobi Preconditioner OpenMP enabled	0.599
Steadycav3	1	GMRES with block Jacobi Preconditioner OpenMP enabled	0.667
Steadycav4	1	GMRES with block Jacobi Preconditioner OpenMP enabled	0.665

Table 29 Final best timings attained in this Project

While it may seem at first glance that GMRES with block Jacobi appears to provide the best solution for many of the test matrices, the author proposes the use of strategies 1 and 4 over this solution.

The reason for the recommendation to use IDR(s) instead of GMRES stems from the fact that it generally outperforms GMRES significantly when restart is required. When restart is not required, IDR(s) usually requires only a few more iterations when compared with GMRES. This is evident from the results presented in Section 3.3. Thus, from a practical point of view, it may be more beneficial to use IDR(s).

The reason to propose the use of hierarchical-LU preconditioner over the block Jacobi preconditioner is due to its scalability. It has the major advantage of having almost linear complexity of $O(N(logN)^2)$. Thus, even though for smaller systems, block Jacobi may perform slightly better than the hierarchical-LU preconditioner, at large sizes or with multiple RHS, the performance of hierarchical-LU preconditioner outshines the block Jacobi preconditioner. In addition, it conditions the system very well, reducing the number of dense matvec required significantly. The reduction in time both sequentially and in parallel has been shown to be about 40-50% for the large FATIMA_20493 test matrix.

There are many improvements that can be made to the hierarchical-LU decomposition codes as introduced in this report. One major improvement that can be made is to improve its parallelization. The parallelization strategy used in this report was rudimentary at best. It was done to give an idea of how the hierarchical-LU preconditioner compares with the block Jacobi preconditioner in a parallel environment. To properly make the code efficient in parallel, much more work and time are required, and is not within the scope of this project. To this, the author would like to propose two approaches that could be taken to improve the parallelization.

In [1], M. de Jong had shown that the use of GPU with block Jacobi preconditioner was able to significantly lower the total time required to solve the largest test problem. In a similar way, the use of GPU with hierarchical-LU preconditioner can also reduce the total time required significantly. The same MAGMA library mentioned in [1] can be employed in this case.

Alternatively, there are available literatures which suggest that task based approach to parallelizing the hierarchical-LU preconditioner can provide almost optimal speedup and good scaling behavior up to many cores. One such literature is that written by Kriemann, R. in [14]. By using the task based approach with a directed acyclic graph for efficient scheduling, the hierarchical-LU decomposition algorithm can be redesigned to provide a speedup behavior as illustrated in Figure 43 below. In the figure, the blue line indicates the speedup behavior should the recursive approach be used. This recursive approach is much like the approach taken in this project. The task based approach is seen to provide a near optimal performance with many cores.

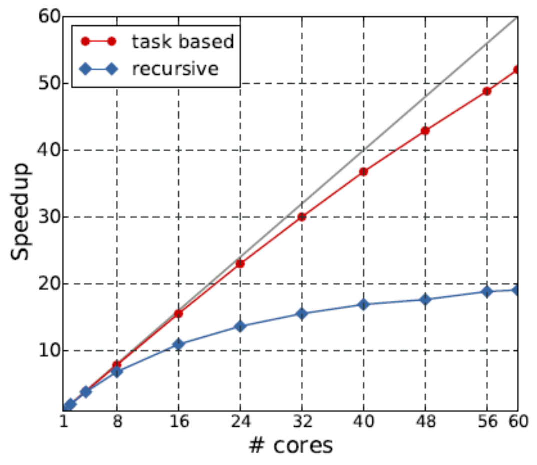


Figure 43 Speedup of task-based H-LU factorization obtained in [14] for Laplace kernel on a sphere

With proper parallelization in placed, the author believes that the efficiency of the panel codes can be significantly improve by the use of IDR(s) with hierarchical-LU preconditioner. The improvement will be especially significant for large matrices and for systems with multiple RHS. To improve the efficiency even more from this point on, the use of Fast Multipole Method to bring the complexity down to $O(N)$ can be explored.

8 References

1. de Jong, M. & van der Ploeg, A. 2012. *Efficient Solvers for Panel Codes.* MARIN Report No. 21750-2-RD.
2. Ang, Y.M.E. 2015. *Master Thesis Literature Review: Efficiency Improvement for Panel Codes*.
3. Sonneveld, P. & van Gijzen, M.B. 2008. *IDR(s): A Family of Simple and Fast Algorithms for Solving Large Nonsymmetric Stsrems of Linear Equations.* SIAM Journal for Scientific Computing Vol. 31, No. 2, pp. 1035-1062.
4. van Gijzen, M.B. & Sonneveld, P. 2011. *Algorithm 913: An Elegant IDR(s) Variant that Efficiently Exploits Biorthogonality Properties.* ACM Trans. Math. Softw. 38, 1, Article 5 (November 2011), 19 pages.
5. Greengard, L. & Rokhlin, V. 1988. *On the Efficient Implementation of the Fast Multipole Algorithm.* Research Report YALEU/DCS/RR-602.
6. Brunner, D. et al. 2010. Comparison of the Fast Multipole Method and Hierarchical Matrices for the Helmholtz-BEM. CMES, vol 58, no.2., pp. 131-158, 2010.
7. Bebendorf, M. 2008. Hierarchical Matrices – A Means to Efficiently Solve Elliptic Boundary Value Problems. Springer-Verlag Berlin Hedelberg.
8. Demmel, J. 2000. *Iterative Algorithms: Golub-Kahan-Lanczos Method*. In Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., & van der Vorst, H., editors. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide.* Philadelphia, SIAM.
9. Bebendorf, M. 2000. Approximation of Boundary Element Matrices. Numer. Math. (2000) 86: 565=589
10. Rjasanow, S. 2002. Adaptive Cross Approximation of Dense Matrices. International Association for Boundary Element Methods 2002.
11. Zhao K, Vouvakis, M.N. & Lee J. 2005. The Adaptive Cross Approximation Algorithm for Accelerated Method of Moments Computations of EMC problems. IEEE Transactions on Electromagnetic Compatibility, Vol. 47, No. 4, Nov 2005.
12. Börm, S., Grasedyck, L. & Hackbush, W. 2005. Hierarchical Matrices (Lecture Notes).

13. Ting, W., Zhao, N.J., & Yi, J.S. 2011. *Hierarchical Matrix Techniques Based on Matrix Decomposition Algorithm for the Fast Analysis of Planar Layered Structures.* IEEE Transactions on Antennas and Propagation, Vol 59, No.11.
14. Kriemaan, R. 2014. *H-LU Factorization on Many-Core Systems*. Computing and Visualization in Science, Vol 16, Issue 3.