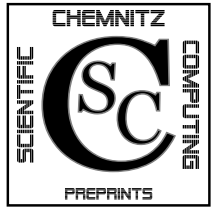
TECHNISCHE UNIVERSITÄT CHEMNITZ

Martin Köhler Jens Saak

Efficiency improving implementation techniques for large scale matrix equation solvers

CSC/09-10



Chemnitz Scientific Computing Preprints

Chemnitz Scientific Computing Preprints – ISSN 1864-0087

Impressum:

Chemnitz Scientific Computing Preprints — ISSN 1864-0087
(1995–2005: Preprintreihe des Chemnitzer SFB393)

Herausgeber:
Professuren für
Numerische und Angewandte Mathematik
an der Fakultät für Mathematik
der Technischen Universität Chemnitz

Postanschrift:
TU Chemnitz, Fakultät für Mathematik
09107 Chemnitz
Sitz:
Reichenhainer Str. 41, 09126 Chemnitz

http://www.tu-chemnitz.de/mathematik/csc/

[27] J.-R. Li and J. White, *Low rank solution of Lyapunov equations*, SIAM J. Matrix Anal. Appl., 24 (2002), pp. 260–280. 2

[28] K. Morris and C. Navasca, *Solution of algebraic Riccati equations arising in control of partial differential equations.*, in Control and Boundary Analysis., J. P. Zolesio and J. Cagnol, eds., vol. 240 of Lecture Notes in Pure Appl. Math., CRC Press, 2005. 1

[29] D. Peaceman and H. Rachford, *The numerical solution of elliptic and parabolic differential equations*, J. Soc. Indust. Appl. Math., 3 (1955), pp. 28–41. 1

[30] T. Penzl, *A cyclic low rank Smith method for large sparse Lyapunov equations*, SIAM J. Sci. Comput., 21 (2000), pp. 1401–1418. 2

[31] ——, *Algorithms for model reduction of large dynamical systems*, Linear Algebra Appl., 415 (2006), pp. 322–343. (Reprint of Technical Report SFB393/99-40, TU Chemnitz, 1999.). 1

[32] Y. Saad, *Numerical solution of large Lyapunov equation*, in Signal Processing, Scattering, Operator Theory and Numerical Methods, M. A. Kaashoek, J. H. van Schuppen, and A. C. M. Ran, eds., Birkhäuser, 1990, pp. 503–511. 1

[33] V. Simoncini, *A new iterative method for solving large-scale Lyapunov matrix equations*, SIAM J. Sci. Comput., 29 (2007), pp. 1268–1288. 1

[34] V. Simoncini and V. Druskin, *Convergence analysis of projection methods for the numerical solution of large Lyapunov equations*, SIAM J. Numer. Anal., 47 (2009), pp. 828–843. 1

[35] E. Wachspress, *The ADI model problem*, 1995. Available from the author. 1

TECHNISCHE UNIVERSITÄT CHEMNITZ

Chemnitz Scientific Computing Preprints

Martin Köhler Jens Saak

Efficiency improving implementation techniques for large scale matrix equation solvers

CSC/09-10

Abstract

We address the important field of large scale matrix based algorithms in control and model order reduction. Many important tools from theory and applications in systems theory have been widely ignored during the recent decades in the context of PDE constraint optimal control problems and simulation of electric circuits. Often this is due to the fact that large scale matrices are suspected to be unsolvable in large scale applications. Since around 2000 efficient low rank theory for matrix equation solvers exists for sparse and also data sparse systems. Unfortunately upto now only incomplete or experimental MATLAB® implementations of most of these solvers have existed. Here we aim on the implementation of these algorithms in a higher programming language (in our case C) that allows for a high performance solver for many matrix equations arising in the context of large scale standard and generalized state space systems. We especially focus on efficient memory saving data structures and implementation techniques as well as the shared memory parallelization of the underlying algorithms.

CSC/09-10 ISSN 1864-0087 December 2009

Contents

1. Introduction 1

2. Low Rank ADI for the Lyapunov Equation 2

3. Efficient Implementation of LRCF-ADI in C 4
   3.1. A Single-Pattern-Multi-Value-LU Decomposition 5
   3.2. A Memory Efficient Cache to Disk Technique 9

4. OpenMP Parallelization 12

5. Numerical Verification 16

6. Future Perspectives 19

A. Listings 20

Author's addresses:

Martin Köhler (komart@hrz.tu-chemnitz.de)
Jens Saak (jens.saak@mathematik.tu-chemnitz.de)
TU Chemnitz
Fakultät für Mathematik
Professur Mathematik in Industrie und Technik
D-09107 Chemnitz

http://www.tu-chemnitz.de/mathematik/industrie_technik

http://www.am.uni-erlangen.de/home/spp1253/wiki/index.php/Preprints, 2010. 1

[13] A. Bensoussan, G. Da Prato, M. C. Delfour, and S. K. Mitter, *Representation and control of infinite dimensional systems*, Systems & Control: Foundations & Applications, Birkhäuser Boston Inc., Boston, MA, second ed., 2007. 1

[14] T. O. A. R. Board, *OpenMP Application Program Interface, Version 3.0*, 2008. 12

[15] T. A. Davis, *Algorithm 832: UMFPACK V4.3 – An unsymmetric-pattern multifrontal method*, ACM Trans. Math. Softw., 30 (2004), pp. 196–199. 8, 19

[16] T. A. Davis, *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006. 8, 16, 19

[17] J. R. Gilbert, *Predicting structure in sparse matrix computations*, SIAM J. Matrix Anal. Appl, 15 (1994), pp. 62–79. 6

[18] J. R. Gilbert and T. Peierls, *Sparse partial pivoting in time proportional to arithmetic operations*, SIAM Journal on Scientific and Statistical Computing, 9 (1988), pp. 862–874. 19

[19] G. Golub and C. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, third ed., 1996. 3

[20] L. Grasedyck, *Existence of a low rank or $\mathcal{H}$-matrix approximant to the solution of a Sylvester equation*, Numer. Lin. Alg. Appl., 11 (2004), pp. 371–389. 1

[21] M. Heyouni and K. Jbilou, *An extended block Arnoldi algorithm for large-scale solutions of the continuous-time algebraic Riccati equation*, Electr. Trans. Num. Anal., 33 (2009), pp. 53–62. 1

[22] I. Jaimoukha and E. Kasenally, *Krylov subspace methods for solving large Lyapunov equations*, SIAM J. Numer. Anal., 31 (1994), pp. 227–251. 1

[23] K. Jbilou, *ADI preconditioned Krylov methods for large Lyapunov matrix equations.*, Linear Algebra Appl., 432 (2010), pp. 2473–2485. 1

[24] K. Jbilou and A. Riquet, *Projection methods for large Lyapunov matrix equations*, Linear Algebra Appl., 415 (2006), pp. 344–358. 1

[25] D. Kleinman, *On an iterative technique for Riccati equation computations*, IEEE Trans. Automat. Control, AC-13 (1968), pp. 114–115. 2

[26] I. Lasiecka and R. Triggiani, *Control Theory for Partial Differential Equations: Continuous and Approximation Theories I. Abstract Parabolic Systems*, Cambridge University Press, Cambridge, UK, 2000. 1

References

[1] P. R. Amestoy, Enseeiht-Irit, T. A. Davis, and I. S. Duff, *Algorithm 837: AMD, an approximate minimum degree ordering algorithm*, ACM Trans. Math. Softw., 30 (2004), pp. 381–388. 4, 16

[2] A. Antoulas, *Approximation of Large-Scale Dynamical Systems*, SIAM Publications, Philadelphia, PA, 2005. 1

[3] A. Antoulas, D. Sorensen, and Y. Zhou, *On the decay rate of Hankel singular values and related issues*, Sys. Control Lett., 46 (2002), pp. 323–342. 1

[4] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*, SIAM, Philadelphia, PA, 1994. 6

[5] N. Bell and M. Garland, *Efficient sparse matrix-vector multiplication on CUDA*, NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008. 20

[6] N. Bell and M. Garland, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, in SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, New York, NY, USA, 2009, ACM, pp. 1–11. 20

[7] P. Benner, *Solving large-scale control problems*, IEEE Control Systems Magazine, 14 (2004), pp. 44–59. 1

[8] P. Benner, J.-R. Li, and T. Penzl, *Numerical solution of large Lyapunov equations, Riccati equations, and linear-quadratic control problems*, Numer. Linear Algebra Appl., 15 (2008), pp. 755–777. 2, 11

[9] P. Benner, V. Mehrmann, and D. Sorensen, eds., *Dimension Reduction of Large-Scale Systems*, vol. 45 of Lecture Notes in Computational Science and Engineering, Springer-Verlag, Berlin/Heidelberg, Germany, 2005. 1

[10] P. Benner, H. Mena, and J. Saak, *On the parameter selection problem in the Newton-ADI iteration for large-scale Riccati equations*, Electr. Trans. Num. Anal., 29 (2008). 4

[11] P. Benner and J. Saak, *Efficient balancing based mor for second order systems arising in control of machine tools*, in Proceedings of the MathMod 2009, I. Troch and F. Breitenecker, eds., no. 35 in ARGESIM-Reports, Vienna, Austria, January 2009, Vienna Univ. of Technology, ARGE Simulation News, pp. 1232–1243. ISBN/ISSN: 978-3-901608-35-3. 4

[12] ——, *A Galerkin-Newton-ADI method for solving large-scale algebraic Riccati equations.* submitted to SIAMMatrix; also available as SPP1253 Preprint

1. Introduction

One of the key ingredients of many linear-quadratic optimal control problems for parabolic PDEs [7, 13, 26, 28], as well as the balancing based model order reduction of large linear systems [2, 9] is the numerical treatment of large scale sparse algebraic matrix equations

$$0 = \mathfrak{R}(X) := CC^T + XA + A^TX - XBB^TX \tag{ARE}$$

and

$$FX + XF^T = -GG^T. \tag{LYAP}$$

In those applications, the coefficient matrices $A, F \in \mathbb{R}^{n\times n}$ are normally sparse (or data sparse) such that multiplications and linear system solves with A and F are comparably cheap. The matrices C, B and G are rectangular, i.e., $C \in \mathbb{R}^{n\times r_C}$, $B \in \mathbb{R}^{n\times r_B}$, $G \in \mathbb{R}^{n\times r_G}$ with r_C, r_G, $r_B \ll n$ (usually). As a rule of thumb recall, the smaller r_C, r_G and r_B are, the more efficient the algorithms we focus on can be. In the case of generalized state space representations the above equations become

$$0 = \mathfrak{R}(X) := CC^T + E^TXA + A^TXE - E^TXBB^TXE \tag{G-ARE}$$

and

$$FXE^T + EXF^T = -GG^T, \tag{G-LYAP}$$

for $E \in \mathbb{R}^{n\times n}$ invertible. The case of rank deficient E matrices, i.e., descriptor systems, will be treated in future versions of our codes. In any case the (numerical) rank of the solution matrices is usually small compared to its actual dimension (see [31, 3, 20]) and therefore the solutions allow for good approximation via low rank solution factors. In the literature one finds two classes of algorithms for the computation of these low rank solution factors. The one class follows an idea by Saad [32]. These methods project the matrix equations onto certain subspaces and solve the much smaller projected equations using classic dense solvers. The main concern then is to find the right subspaces to project on. Starting with the idea to use Krylov subspaces [22] these methods have lead to a class of extended Krylov subspace based projection methods. The Krylov subspace generated by an Arnoldi process for the matrix F here is extended by the basis generated with respect to F^{-1} [21, 23, 24, 33]. A convergence analysis for these methods has recently been presented in [34].

The second path, that we are following here, is based on the observation by Wachspress (see, e.g. [35]) that vectorized Lyapunov equations, i.e., the equivalent representation as linear system of equations of squared size, have the same structure as finite difference discretized Poisson equations (see,e.g. [12]). Therefore he proposed that they are an alternating directions implicit (ADI) [29] model problem.

Algorithm 1 Newton's Method for Algebraic Riccati Equations – Kleinman Iteration

Input: A,B,C as in (ARE) and an initial guess $K^{(0)}$ for the feedback.
Output: X^{k_0} solving (ARE) and the optimal state feedback K^{k_0} (or approximations when stopped before convergence).

1: **for** $k = 1, 2, \ldots, k_0$ **do**
2: Determine the solution $X^{(k)}$ of $(A - BK^{(k-1)^T})^T X^{(k)} + X^{(k)}(A - BK^{(k-1)^T}) = -CC^T - K^{(k-1)}K^{(k-1)^T}$.
3: $K^{(k)} = X^{(k)}B$.
4: **end for**

The applicability of the ADI in low rank fashion has been proven in [8, 30, 27]. We will briefly review the ideas in Section 2.

The algebraic Riccati equation can be tackled with the ADI in combination with a Newton-Kleinman [25] approach. The key observation here is that the Frechét derivative of $\mathfrak{R}(\cdot)$ is a Lyapunov operator which can be handled with low rank ADI (see, e.g., [8]).

The remainder of this paper is structured as follows. In Section 2 we present the basic algorithms we are interested in. Especially we point out the fundamental features and properties of the representation we use in the codes. Thereafter we point out how these are implemented in Section 3. We introduce the single pattern multi value (spmv) LU-decomposition, that allows us to reduce the memory requirements drastically in sparse computations. We further present variants of the algorithms implementing a minimum runtime, or a minimum memory requirement approach. Section 4 then focuses on the shared memory parallelization of the algorithms using OpenMP. It is followed by a section on numerical verification of the ideas. The paper is finished with some conclusions and an outlook on future research perspectives.

2. Low Rank ADI for the Lyapunov Equation

Before we start the discussion of the low rank ADI for the Lyapunov equation, we introduce the important notion of low rank updated sparse matrices.

Definition 2.1 (splr)**.** A matrix $F \in \mathbb{R}^{n\times n}$ is called *sparse plus low rank* or simply *splr* if we can find matrices $A \in \mathbb{R}^{n\times n}$ and $U, V \in \mathbb{R}^{n\times p}$ such that

$$F = A + UV^T.$$

We will require the splr structure to be able to apply the low rank solver to the Lyapunov equation resulting in the Newton step (step 2 in Algorithm 1) when


```
void mvp_csr(long n, long *rowptr, long *colptr,
             double *x, double *y){
 double t;
 long i,j;
 #pragma omp parallel for private(i,j) default(shared).
 for (i = 0;i < n;i++){
  y[i] = 0;
  for (j = rowptr[i]; j < rowptr[i+1];j++){
        y[i] += values[j]*x[colptr[j]];
  }
 }
}
```

Listing 2: OpenMP parallel matrix-vector-product for CSR

```
void mvp_csrt(long n, long *rowptr, long *colptr,
              double *x, double *y){
 double *tmp, *ev;
 double r, t;
 long id, pos, one =1;
 long worldsize = omp_get_max_threads();
 tmp = (double*) malloc(sizeof(double) * n * worldsize);
 for ( i = 0; i < n*worldsize; i++) { tmp[i] = 0.0;}
 ev = (double*) malloc(sizeof(double) * worldsize);
 for ( i = 0 ; i < worldsize ; i++ ) { ev[i] = 1.0;}
 #pragma omp parallel for private(i,j,t,pos,id)
 for (i=0;i<n;i++){
  id = (long) omp_get_thread_num();
  pos = id * matrix->rows;
  t = x->values[i];
  for ( j = rowptr[i] ; j < rowptr[i+1] ; j++ ) {
   tmp [ pos + colptr[j] ] += values[j] * t;
  }
 }
 r = 1.0; t = 0.0;
 // dgemv_ - BLAS r*Ax+t*y -> y
 dgemv_("N",&n,&worldsize,&r,tmp,&n,ev,&one,&t,y,&one);
 free(tmp);
 free(ev);
}
```

Listing 3: OpenMP parallel transposed matrix-vector-product for CSR

```c
void lureuse(long rows, long cols,
             double *values, long *colptr, long *rowptr,
             long *lcolptr, long *lrowptr,
             long *ucolptr, long *urowptr,
             double *lvalues, double *uvalues)
{
 double *w;
 long i, j, k, col;
 long row;
 double alpha;
 w = (double *) malloc (sizeof(double) * cols);
 for ( i = rowptr[0];  i < rowptr[1]; i++) {
  w [colptr[i]] = values[i];
 }
 for ( i = urowptr[0];  i < urowptr[1]; i ++ ){
  uvalues[i] = w[ucolptr[i]];
 }
 for ( i = 0; i < cols; i++) { w[i] = 0.0; }
 lvalues[0] = 1.0;
 for ( i = 1; i < rows; i++) {
  for (j=rowptr[i]; j < rowptr[i+1]; j++){
   w[ colptr[j] ] = values[j];
  }
  for (j=lrowptr[i]; j < lrowptr[i+1]-1; j++){
   col = lcolptr[j];
   alpha = w[col] / uvalues[urowptr[col]];
   for ( k = urowptr[col]; k < urowptr[col+1]; k++){
    w[ucolptr[k]] -= alpha * uvalues[k];}
   w[col] = 0;
   lvalues[j] = alpha;
  }
  lvalues[j] = 1.0;
  for (j=urowptr[i]; j < urowptr[i+1]; j++){
   uvalues[j] = w[ucolptr[j]];
   w[ucolptr[j]] = 0;
  }
 }
 free (w);
}
```

Listing 1: pattern reuse algorithm in C

Algorithm 2 (Generalized Low-rank Cholesky factor ADI iteration (G-LRCF-ADI))

Input: E,F,G defining (LYAP), or (G-LYAP) respectively and shift parameters $\{p_1, \ldots, p_{i_{max}}\}$

Output: $Z = Z_{imax} \in \mathbb{C}^{n \times ti_{max}}$, such that $ZZ^H \approx X$

1: Solve $(F + p_1 E)V_1 = \sqrt{-2\,\mathrm{Re}\,(p_1)}G$ for V_1
2: $Z_1 = V_1$
3: **for** $i = 2, 3, \ldots, i_{max}$ **do**
4: Solve $(F + p_i E)\tilde{V} = EV_{i-1}$ for $\tilde{V}$
5: $V_i = \sqrt{\mathrm{Re}\,(p_i)/\,\mathrm{Re}\,(p_{i-1})}(V_{i-1} - (p_i + \overline{p_{i-1}})\tilde{V})$
6: $Z_i = [Z_{i-1}\ \ V_i]$
7: **end for**

solving the Riccati equation (ARE).

For an splr matrix we can employ the Sherman-Morrison-Woodburry formula[1] (e.g. [19])

$$(A + UV^T)^{-1} = A^{-1} - A^{-1}U(I + V^T A^{-1} U)^{-1} V^T A^{-1}, \tag{1}$$

whenever its inversion is required.

We start our discussion of the low rank ADI from the two step ADI introduced by Wachspress:

$$(F + p_i I)\, X_{i+\frac{1}{2}} = GG^T - \left(F^T - p_i I\right) X_i, \tag{2}$$

$$(F + p_i I)\, X_{i+1} = GG^T - \left(F^T - p_i I\right) X_{i+\frac{1}{2}}. \tag{3}$$

The basic ideas towards a low rank version of the iteration are now

- represent the iterates by low rank decompositions $X_i = Z_i Z_i^T$,
- assume $Z_0 = 0$,
- insert the one equation into the other to end up with a single step iteration.

The main trick to result in

$$Z_1 = \sqrt{-2p_1}(F + p_1 I)^{-1} G, \tag{4}$$

$$Z_j = \left[\sqrt{-2p_1}(F + p_j I)^{-1} G,\ (F + p_j I)^{-1}(F - p_j I) Z_{j-1}\right] \tag{5}$$

is the observation, that

$$(F - p_i I) = (F + p_i I - 2p_i I)$$

[1] This formula is often referred to as *matrix inversion lemma* in the engineering literature, as well.

dimension of the system	size of L+U in MB	size of 16 LUs in MB
100	0.02	0.35
2 500	1.16	18.59
10 000	6.45	103.20
40 000	33.62	537.92
90 000	90.75	1 452.00
250 000	285.10	4 561.30
562 500	718.00	11 488.00
1 000 000	1 379.00	22 064.00

Table 1: memory usage of one factor and the complete set of 16 factors on a 64-bit architecture

and thus

$$(F + p_i I)^{-1}(F - p_i I) = I - 2p_i(F + p_i I)^{-1}.$$

Note that the assumption $Z_0 = 0$ is not necessary but a non-zero initial matrix increases the computational cost notably, since it has to be processed in every iteration step in addition to the new columns. Equation (4) gives rise to Algorithm 2 in the case $E = I$. In [11] we gave a detailed derivation for the case $E \neq I$ invertible, motivating the generalization for Equation (G-LYAP). The choice of the shift parameters is crucial to get good performance of the solver. Several methods for their computation have been discussed in [10] and references therein.

3. Efficient Implementation of LRCF-ADI in C

We discussed the fundamental ideas of the Low Rank ADI in Section 2. In this section the focus lies on how to factorize a set of shifted sparse matrices, like we use them in the ADI iteration. The following example will show, why it might not be a good idea to factorize all matrices $A_i := A - p_i I$ or $A_i := A - p_i E$ with standard techniques at the beginning of the iteration.

Example 3.1 (demo_l1 from LyaPack[2])**.** Consider a simple FDM-semidiscretized PDE problem (an instationary convection-diffusion equation on the unit square with homogeneous first kind boundary conditions) with different discretization levels n. To reduce the fill-in of the L and U factors, we reorder the rows and columns using approximate minimum degree reordering [1]. Assume we use 16 shift parameters in the ADI iteration. We intend to use the shifts cyclicly, i.e., restart with the first shift once we have applied all of them and the iteration has not converged. Since we do not want to compute the same LU factorizations

[2] http://netlib.org/lyapack

CSR transposed matrix vector product. After we discussed a workaround solution for the missing vector reduce operation in OpenMP, Listing 3 presents the resulting C code. Computing partial results and merging them together in another step is one way to deal with the missing vector reduce operation. The transposed matrix vector product can be implemented with synchronization statements too, but then for every access on y we have to synchronize which slows down the whole algorithm. In practice that means that the actual speed up can be smaller than 1.

```
#pragma omp parallel for private(i)
for (i=0;i<n;i++){
 umfpack_dl_numeric (rowptr[i], colptr[i], values[i],
                     Symbolic, &Numeric[i],
                     data->Control[i], data->Info[i]) ;
}
```

the computed decomposition may be incorrect. Depending on the size of the matrix the probability for this to happen increases. Another problem is the relatively high memory usage of UMFPACK. The question arises whether it is possible to combine the advantages of the spmv-LU and the multifrontal ideas in UMFPACK.

Upcomming Parallelization techniques. Since NVIDIA introduced the Compute Unified Device Architecture (CUDA), the usage of GPUs for scientific computing has become more and more popular. It has to be proofen whether we can use the parallel performance and computation power (up to 1 Tflops in single precission) to fasten the ADI process or at least some parts of it, especially for sparse matrix computations. Some results for the sparse matrix vector product have already been presented by NVIDIA [6, 5]. Another interessting question is, if complex algorithms like eigenvalue computations or graph reordering can be implemented on GPUs, too.

A. Listings

Pattern-Reuse-LU. Listing 1 shows a C implementation of the pattern-reuse-LU like it is used in C.M.E.S.S. for the spmv. It differs from Algorithm 5 by dealing with the worker vector w. The implemented version performs the $w = \tilde{A}(i,:)$ operation in a sparse way to avoid reseting or reintializing the whole vector. The sparse AXPY-operation is done as an inline operation in the inner `for`-loop. After copying the required values from w to $\tilde{U}$ we reset only the used values in w.

CSR matrix vector product. In Listing 2 we present a small but efficient implementation of a sparse matrix vector product for matrices stored in Compressed Sparse Row Storage. Because of disjoint write access to y and read-only use of A and x each process only needs i and j as private variables. All other data is shared between all processes.

repeatedly, we have to store 16 decomposition in memory at a time. Due to the property of 64-bit architectures that `long` and `double` both occupy 8 bytes, this leads to enormous memory requirements, as shown in table 1.

Example 3.1 shows that it is nearly impossible to store all factors in memory using standard desktop computers. In Section 3.1 we derive a variant of the LU-decompostion, exploiting the special structure of our systems to reduce the memory usage by almost 50%. A technique to solve Lyapunov equations with even less main memory is presented in Section 3.2. To handle the pattern of a sparse matrix easily, we use the following definition.

Definition 3.2. Let $A \in \mathbb{R}^{n\times m}$ be a matrix. We call the set

$$\mathcal{P}(A) = \{(i,j) \mid A_{i,j} \neq 0\}$$

pattern of A. Furthermore we define

$$\mathcal{P}_R(A,i) = \{j \mid A_{i,j} \neq 0\}$$

as the *pattern of the i-th row* of A.

3.1. A Single-Pattern-Multi-Value-LU Decomposition

The following example motivates how we can reduce the memory requirements for the multiple shifted linear systems.

Example 3.3. Consider the small 4×4 matrix

$$A := \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 2 & 1 & 0 \\ 3 & 0 & 3 & 0 \\ 0 & -2 & 0 & 4 \end{pmatrix},$$

with its LU-decomposition

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 3 & 0 & 1 & 0 \\ 0 & -1 & \frac{1}{3} & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 2 & 1 & 0 \\ 0 & 0 & 3 & 3 \\ 0 & 0 & 0 & 3 \end{pmatrix},$$

and the pattern of L and U:

$$L = \begin{pmatrix} * & & & \\ & * & & \\ * & & * & \\ & * & * & * \end{pmatrix} \quad U = \begin{pmatrix} * & & & * \\ & * & * & \\ & & * & * \\ & & & * \end{pmatrix}.$$

Examining the shifted matrix $A + I_4$ and its LU decomposition (with the same pivoting strategy)

$$A + I_4 = \begin{pmatrix} 2 & 0 & 0 & -1 \\ 0 & 3 & 1 & 0 \\ 3 & 0 & 4 & 0 \\ 0 & -2 & 0 & 5 \end{pmatrix} = \overbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \frac{3}{2} & 0 & 1 & 0 \\ 0 & -\frac{2}{3} & \frac{1}{6} & 1 \end{pmatrix}}^{\tilde{L}} \cdot \overbrace{\begin{pmatrix} 2 & 0 & 0 & -1 \\ 0 & 3 & 1 & 0 \\ 0 & 0 & 4 & \frac{3}{2} \\ 0 & 0 & 0 & \frac{19}{4} \end{pmatrix}}^{\tilde{U}},$$

we find that the patterns for $\tilde{L}$ and $\tilde{U}$ coincide with those of L and U.

We observe in Example 3.3, that the position and the number of non zero elements is exactly the same (except from eventually computed exact zero entries resulting from certain value distributions in the original matrix) when we factorize a matrix shifted by multiples of the identity matrix. The systems in the ADI iteration all are of the same structure. The question arises, whether it is possible to exploit this property of $LU := A$ and $\tilde{L}\tilde{U} := A + pI$ to preserve the pattern in order to save memory. The answer is, yes.

Another question is, whether we can use information from an LU decomposition of A to factorize systems $A + pI$ with certain shifts p. Again the answer to this question is yes. To develop an algorithm which solves both problems we use Definition 3.2 to access the pattern structure independent from the values.

Reuse of information from LU = A In the process of shift parameter computation for the ADI we have to solve systems with A, so we can assume, that $LU = A$ has been computed before the iteration begins. With the knowledge of L and U the sets $\mathcal{P}(L)$, $\mathcal{P}(U)$ and the sizes of these sets are given. The fact that we know the sizes of the sets is not interessting in a mathematical point of view, but for the efficient implementation it is important, because the memory can be allocated in one step and no reallocations are necessary. Note that reallocations can be super-linear complexity algorithms and always require context changes to the operating system when called.

Another important property is that from $\mathcal{P}(A) = \mathcal{P}(\tilde{A})$ it follows $\mathcal{P}(L) = \mathcal{P}(\tilde{L})$ and $\mathcal{P}(U) = \mathcal{P}(\tilde{U})$ with $\tilde{L}\tilde{U} = \tilde{A}$, e.g. [17]. The main idea is, that the pattern sets depend only on the position of the entries not on their actual values. We assume that numerical cancelation is neglected. The sets $\mathcal{P}_R(A, i)$ are easily accessible because we store the matrix in Compressed Sparse Row Storage (CSR) [4, p. 57]. If the matrix is given in Compressed Sparse Column Storage (CSC), we factorize A^T instead of A.

To derive the algorithm we analyze a dense row-wise LU-decomposition presented in Algorithm 3. The first step is to put in the information we have when we

two norm in every step. If we do this every 5-th step we have to compute at most five steps more, but we easily compensate this by the time saved due to the dropped 2-norm evaluations. To accelerate the computation of the pattern, the first decomposition of A is done with UMFPACK. Reasons for not using it for every LU in the algorithm are shown in Section 6.

6. Future Perspectives

Left-Looking LU decomposition. Like it is implement in CSparse[16], a left looking LU decomposition is significantly faster then a naive implementation. It is derived from a block partioning

$$LU = A$$
$$\begin{pmatrix} L_{11} & & \\ l_{21} & 1 & \\ L_{31} & l_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & u_{12} & U_{13} \\ & u_{22} & u_{23} \\ & & U_{33} \end{pmatrix} = \begin{pmatrix} A_{11} & a_{12} & A_{13} \\ a_{21} & a_{22} & a_{23} \\ A_{31} & a_{32} & A_{33} \end{pmatrix}. \tag{7}$$

The middle row and column of each matrix is the kth row and column of L, U and A. If the first $k-1$ columns of L and U are known, we get three equations to compute the k-th columns of L and U: $L_{11}u_{12} = a_{12}$ is a triangular system that can be solved for u_{12}, $l_{21}u_{12} + u_{22} = a_{22}$ can be solved for the entry u_{22}, and $L_{31}u_{12} + l_{32}u_{22} = a_{32}$ can then be solved for l_{32}. The rearrangement of these equations leads to

$$\begin{pmatrix} L_{11} & & \\ l_{21} & 1 & \\ L_{31} & 0 & I \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} a_{12} \\ a_{22} \\ a_{32} \end{pmatrix}. \tag{8}$$

The solution of this system gives $u_{12} = x_1$, $u_{22} = x_2$ and $l_{32} = \frac{x_3}{u_{22}}$. In our case L, y and x are sparse. For this case there exist sparse triangular solvers, which are accelerated with graph-theoretical results like using the reachability set of underlying directed graph of L [18]. An interessting question is, if we can find a reuse algorithm which based on the left looking LU and which information are useful to derive fast parallel algorithms.

Using UMFPACK. UMFPACK [15] is a set of functions to solve sparse linear systems of equations of the form $Ax = b$ or $A^Tx = b$ with the non-symmetric multifrontal method. The factorization of a matrix is divided in a symbolic and a numerical phase. Because it uses level 3 BLAS it is much faster than standard sparse LU decompositions. But several problems prohibit the usage for the single-pattern-multi value idea. The first problem is that UMFPACK appears not to be thread safe, i.e. when running a construct like

	with 2-norm	without 2-norm	cache to disk
iterations	78	99	99
final 2-norm residual	5.77e-07	5.87e-07	5.87e-07
final $\frac{\|V_i\|_F}{\|Z_i\|_F}$	2.86e-11	8.47e-13	8.47e-13
time all (in seconds)	2346.21	2119.76	3168.41
first LU	1017.44	1056.66	1021.02
spmv-LU	854.28	885.77	2040.08
ADI	474.49	177.33	107.31

Table 5: demo_l1 with one rhs and different parameters

	2-norm every step	2-norm every 5th step
iterations	107	109
2-norm residual	6.29e-07	6.29e-07
last $\frac{\|V_i\|_F}{\|Z_i\|_F}$	2.69e-11	9.53e-13
time all	5063.01	1985.44
first LU[8]	19.14	18.82
spmv-LU	1096.33	1098.45
ADI	3947.54	868.17

Table 6: demo_l1 with 4 rhs

memory usage is as small as possibile. After the iteration the spmv is cleared and the solution factor is loaded from file to compute the 2-norm residual. Table 5 shows the runtimes for these three configurations. We can see that the naive LU decomposition costs more time than the reuse computations all together. In the ADI iteration the evaluation of the 2-norm residual is the most expensive operation. The speed of the cache-to-disk solution extremely depends on the storage, where the solvers are read from. Our test system has 4 SATA hard disk in a RAID-0 array, which allows transfer rates of up to 150MB/s.

In the case of a multiple input system, we have to solve with multiple right hand sides. To test the implementation we use a rank four randomly generated right hand side for the Lyapunov equation. Table 6 shows the results with computing the 2-norm residual in every step and every 5-th step. The stopping criteria are the same as in the single right hand side case. We observe that the main iteration takes more than eight times the execution time (compared to that for a rank-1 right hand side) if we evaluate the 2-norm in every step. In the case we compute the 2-norm residual in every 5-th step, we only need about double the time. The largest problem is, that the size of the solution factor Z_i increases more rapidly. The data in the CPU cache can not be used again because we have to transfer to much data from CPU to memory for every basic operation while evaluating the 2-norm residual. The example shows that it is not nessecary to evaluate the

Algorithm 3 rowise LU-decomposition

Input: $A \in \mathbb{R}^{n\times n}$

Output: $L \in \mathbb{R}^{n\times n}$ lower triangular, $U \in \mathbb{R}^{n\times n}$ upper triangular with $LU = A$

for $i = 1, \dots, n$ **do**
  $U(i,:) = A(i,:)$
  $L(i,i) = 1$
  for all $j = 1, \dots, i-1$ **do**
    $L(i,j) = \alpha = \frac{U(i,j)}{U(j,j)}$
    $U(i,:) = U(i,:) - \alpha \cdot U(j,:)$
  end for
end for

Algorithm 4 sparse axpy - SpAXPY(α, v, w)

Input: $w \in \mathbb{R}^n$, $v \in \mathbb{R}^n$ as v_{val} and v_{ind}, $\alpha \in \mathbb{R}$

Output: $w = w + \alpha v$

for $i = 1, \dots, |v_{ind}|$ **do**
  $w(v_{ind}(i)) = w(v_{ind}(i)) + \alpha v_{val}(i)$
end for

know $\mathcal{P}(L)$ and $\mathcal{P}(U)$. The for-loop with respect to the index j needs only access elements which exist in L. After putting in the information we have already from $\mathcal{P}(L)$, we can turn the inner continous for-loop into a loop over all elements $j \in \mathcal{P}_R(L,i)$. Next we have to avoid working on a full row in U. The best way to do this is to copy the i-th row of A into a temporary vector w and perform all operations done on the i-th row of U on this vector. After finishing the j-loop we move only the in $\mathcal{P}_R(U,i)$ existing columns from w to $U(i,:)$. Thefore the operation

$$U(i,:) = U(i,:) - \alpha \cdot U(j,:)$$

turns into

$$w = w - \alpha \cdot U(j,:),$$

which can be implemented by scattering the j-th row of U and using BLAS, or by using the sparsity of U. We use a sparse axpy operation. Assuming $U(j,:)$ is a sparse vector, we only have to modify w at position where $U(i,:)$ is non zero, which will be done by Algorithm 4.

This enables us to compute the LU-factors $\tilde{L}$, $\tilde{U}$ of $\tilde{A}$ with $\mathcal{P}(\tilde{A}) = \mathcal{P}(A)$ reusing the knowledge of the patterns as is shown in Algorithm 5. Thus Algorithm 5 is the answer to our second question. With a given factorization of A we can compute the factorization of a matrix with the same pattern (as, e.g., $\tilde{A} = A + pI$) faster than starting a new decomposition of $\tilde{A}$ from scratch. Numerical results and timings that show the adavantage of this method will be presented in Section 5.

Algorithm 5 Pattern-reuse-LU

Input: $\tilde{A} \in \mathbb{R}^{n\times n}$, $\mathcal{P}(L)$ and $\mathcal{P}(U)$ with $LU = A$ and $\mathcal{P}(A) = \mathcal{P}(\tilde{A})$
Output: $\tilde{L}$, $\tilde{U}$ with $\tilde{L}\tilde{U} = \tilde{A}$

```
Ũ(1,:) = Ã(1,:)
for i = 2,...,n do
    w = Ã(i,:)
    for all j ∈ P_R(L,i) ordered ascending do
        L̃(i,j) = α = w(j)/Ũ(j,j)
        SpAXPY(−α, U(j,:), w)
    end for
    for all j ∈ P_R(U,i) do
        Ũ(i,j) = w(j)
    end for
end for
```

We present a C implementation in Appendix A. If we do not know the patterns of L and U beforehand, we can compute them using existing software packages, like CSparse [16] or UMFPACK [15].

Single Pattern Idea. To derive an algorithm decreasing the memory usage, we have to analyse the pattern reuse idea again. We observe that the access to the given pattern structure is read-only.

The patterns of the new factors are the same as those of the previously computed ones, as we discussed in the previous section. The systems $A + p_i I$ have the same pattern as A for all i, then the computed LU-factors have the same structure. Note that we assume A to be Hurwitz, i.e., all diagonal entries need to be non-zero anyway and we cannot add elements to the sparsity pattern shifting with multiples of the identity.

The answer to our first question turns out to be: factorize the the system $A + pI$ or a system with the same pattern and use the pattern sets $\mathcal{P}(L)$ and $\mathcal{P}(U)$ for all subsequent factorizations of shifted matrices. Only the values of the new factorizations are saved. In case of CSR storage this means that we keep the *rowptr*-array and the *colptr*-array for all systems in main memory and only the *values*-array differs for each shift. Table 2 shows the reduced memory usage and the savings in comparison to the naive memory usage from example 3.1. It shows that we can save nearly half of the memory on a 64bit architecture. The maximum savings depend on the size of `double` and `long` data types on the architecture used. On 32bit architectures we only save 30% up to 40%, because there `long` variables, which used for the pattern sets, are only 4 Byte large instead of 8 byte on 64bit.

dimension	time CSparse	time reuse	savings
2 500	0.0062	0.0022	63.85%
10 000	0.0539	0.0235	56.32%
40 000	0.4147	0.2120	48.89%
90 000	1.7549	1.1670	33.50%
250 000	8.8364	6.9544	21.30%
562 500	34.2441	28.6534	16.33%
1 000 000	89.1395	75.8866	14.87%

Table 3: pattern reuse: normal LU with CSparse, reuse with alogrithm 5

dim.	naive LU	spmv LU			max.
		1 core	2 cores	4 cores	savings
90 000	98.506	19.594	10.479	10.054	89.79%
160 000	299.400	60.046	34.161	33.481	88.81%
250 000	686.910	121.172	69.666	66.768	90.28%
562 500	4 069.529	506.111	296.987	269.753	93.37%
1 000 000	18 526.310	1 318.375	762.947	707.374	96.18%

Table 4: spmv vs. naiv sparse LU, 16 LU decompositions

implemented rowwise LU decomposition and 1 up to 4 cores for the spmv algorithm. Table 4 show the computation time for a system with 16 shift parameters, like we would typically employ it inside an ADI process.

Basic example. As test example, we use a simple FDM-semidiscretized PDE problem from Example 3.1. The unit square is discretized with an aperture size of 1000 points in each direction. The resulting system matrix has the dimension 10^6. The solution of the Lyapunov eqaution has about 10^{12} degrees of freedom. They are approximated by a low rank factor, computed by the ADI. The configuration parameters of the ADI are:

- maximum number of iterations: 500
- 2-norm residual truncation: 1e-8
- relative change truncation ($\frac{\|V_i\|_F}{\|Z_i\|_F}$): 1e-12

The solution is computed in three different ways. The first one computes a naive LU decomposition to get the patterns of L and U and store the complete spmv-LU in the main memory. The 2-norm residual is evaluate in every step. The second one stores everything in memory but the 2-norm residual is only computed at the end of the iteration. The third one uses the cache-to-disk technique presented in Section 3.2. In this case, the spmv does not work in parallel to ensure that the

loaded. The *wait-until-loaded* stops the execution of the current program until a specified solver is loaded. Since OpenMP does not provide easy facilities to implement such behavior, we use pthreads, the thread environment existing on nearly all POSIX compatible operating systems. Our thread pool only has one worker thread because parallel I/O to hard disk should be avoided to ensure short reading of the solvers from hard disk to memory.

Dense linear algebra operations. Many operations in the ADI process, like scaling vectors or adding matrices, are dense operation. The Basic Linear Algebra Subroutines (BLAS) are used in this case. To accelerate them we choose a multithreaded implementation. This can be GotoBlas[6], ATLAS[7] or a vendor specific one, provided by CPU or server vendors.

5. Numerical Verification

The numerical test have been performed on a dual Intel®Xeon®5160 computer with 64GB DDR2-667ECC memory. Every CPU has two cores and 4MB L2 cache. Thus we can use up to 4 cores in parallel jobs. The operating system used is OpenSuse 11.1 x86-64. The four SATA harddisks are accessed via RAID-0 delivering high performance. The following numerical test were carried out on this machine with the matrices generated for example 3.1. The BLAS implementation used is GotoBlas2 1.08 with OpenMP thread support.

Pattern reuse. The pattern reuse idea, described in Section 3.1 is tested with matrices from Example 3.1. The matrices are reordered using the approximate minimum degree (AMD) [1] reordering strategie to reduce fill-in. The patterns $\mathcal{P}(L)$ and $\mathcal{P}(U)$ have been computed with CSparse [16]. CSparse implements an easy to use left looking LU decomposition for sparse matrices. We see in table 3, that the reuse idea can be used to save up to the half of the computation time for an LU decomposition, if we know the structure of the factors beforehand. For larger systems the CPU-cache limitation introduces additional difficulties. Ideas to prevent this will be noted in Section 6.

Single-Pattern-Multi-Value LU decomposition. The extension of the pattern reuse idea to save memory was presented in Section 3.1 and its parallelization in Section 4. The test uses one core for the initial LU decomposition with a naiv

[6] http://www.tacc.utexas.edu/resources/software/#blas
[7] http://math-atlas.sourceforge.net/

N	16 LUs in MB	spmv[3] LU	savings
100	0.35	0.17	51.70%
625	3.34	1.66	50.28%
2 500	18.59	9.35	49.69%
10 000	103.20	53.68	47.99%
40 000	537.92	281.26	47.71%
90 000	1 452.00	760.91	47.60%
160 000	2 804.50	1 471.50	47.53%
250 000	4 561.30	2 394.50	47.50%
562 500	11 488.00	6 038.00	47.44%
1 000 000	22 064.00	11 604.00	47.41%

Table 2: memory usage of the single pattern multi value idea and savings on a 64-bit architecture

In case of generalized systems, the patterns of stiffness matrix A and mass-matrix F may differ. That means we can not directly use $\mathcal{P}(L)$ and $\mathcal{P}(U)$ from A. Nevertheless we can use the idea to compute $A + pE$. Instead of $\mathcal{P}(L)$ and $\mathcal{P}(U)$ for $LU = A$ we construct $\mathcal{P}(\tilde{L})$ and $\mathcal{P}(\tilde{U})$ by decomposing $A + E$. The factorization of $(A+E)$ can be done in a symbolic way on the pattern only, because we do not need the values if we compute the shifted systems with algorithm 5. If we want to skip the symbolic factorization of $A + E$, we may factorize $A + p_1E$ with a standard LU solver and and reuse its factors patterns to compute $A + p_iE$ $(i > 1)$, with Algorithm 5.

3.2. A Memory Efficient Cache to Disk Technique

The technique in Section 3.1 can reduce the memory usage dramatically. But on standard workstations it is impossible to store these in memory for large systems ($n \geq 250\,000$). To discover a way to be able to solve very large problems on standard workstations, we analyse the ADI algorithm again. The analysis focuses on which data is needed per iteration step and which data does not effect the current iteration step.

First we take a look at the data dependency between two iteration steps in algorithm 2. Line 4 and 5 refer to the previously computed V_i.

$$\text{Solve}\,(F + p_iI)\tilde{V} = V_{i-1}$$
$$V_i = \sqrt{\text{Re}\,(p_i)/\,\text{Re}\,(p_{i-1})}(V_{i-1} - (p_i + \overline{p_{i-1}})\tilde{V})$$

and line 6 refers to the previously computed Z_i. But the already computed columns of Z_i are not needed to compute the next iterate. Because Z_i is a dense

matrix, it is stored in the Fortran matrix format. That means the data is stored in one large one dimensional array, where the columns are contained one after the other. The update

$$Z_i = [Z_{i-1}\ V_i]$$

can easily be performed by resizing Z_i to $\dim([Z_{i-1}\ V_i])$ and concatenating V_i to the end of Z_{i-1}. For $V_i \in \mathbb{R}^{n\times m}$, Z_i in every step uses $n \cdot m \cdot$ `sizeof`(double) of additional memory. In the case that Z_i can not be resized directly. This happens if there is not enough free memory behind the current Z_i. Then all data is copied to a new memory location, which has sufficient size. Therefore the system temporarily needs nearly twice the memory of the factor. The worst case would be if this happened in every iteration step. To prevent this, we want to write Z to external storage like a hard disk drive. If we realize the update of Z_i by writing Z_i to disk adding of new columns at the end is equivalent to appending the new data at the end of the file. The replacement of $Z_i = [Z_{i-1}\ V_i]$ by

```
fwrite(Vi, sizeof(double), n, m, filepointer);
```

where `filepointer` is the output file opened for the Z factor, eliminates the need to store Z_i in memory. The only data we have to keep in memory are the current V_i and $\tilde{V}$, whose memory locations can be overwritten in every iteration step. The size of V_i and $\tilde{V}$ is fixed. Reallocation is not needed and the complete memory can be allocated en block at the beginning of the iteration. After finishing the iteration, the matrix Z can be loaded into the memory as one block again.

We also keep the previously computed solvers for $(A+p_iI)$ in the system memory. From Example 3.1 we know that this is an undesirable amount of memory. But we only have to solve with one of these solvers in every iteration step. The idea is to dump all the other solvers to external storage after the solvers are computed and load them on demand once the iteration step needs them. After solving we overwrite the memory location with the next solver to avoid `malloc` and `free` calls. Depending on the available memory, we can pre-load the next k solvers to accelerate the iteration. The complete cache-to-disk solution is shown in Algorithm 6.

Even if we can compute the low rank solution of a Lyapunov equation with less memory, we have some new problems. The evaluation of the stopping criteria has to be compatible with our data storage. The most accurate criterion, the residual, is taken in the 2 norm in our case. In this norm it can be computed via a power iteration applied to the symmetric residual operator. In that case we have to compute matrix vector products with the system matrix F and the current solution factor Z_i. Because F is sparse, we store it in the main memory, but Z only exists in a file and not in memory. If we load the complete Z_i into the memory to compute the residual in every step, the iteration is becomes much

have to compute

$$y = \sum_{k=1}^{n} x_k (A_{k,\cdot})^T.$$

If we parallelize this like $y = Ax$ we have to deal with the problem, that an entry in y has to be written by different computational threads. Therefore we need synchronization statements, before every access on y. This will potentially slow down the computation significantly. Alternatively every job computes its own part of the linear combination and all parts are summed up at the end. This is done by a reduction operation over all local result. Unfortunately the reduction operation, which is provided by OpenMP, only works on scalars and not on vectors. The only way to accelerate the computation without using synchronization thus is to implement a workaround for the missing vector-reduction. The idea is first to compute the local parts of the linear combination via

$$y^{(i)} = \sum_{\substack{k=1,\dots,n \\ \text{row } k \text{ belongs to job } i}} x_k (A_{k,\cdot})^T$$

and afterward merge them by the product

$$y = \left[y^{(1)}, \dots, y^{(\#jobs)}\right] \cdot e \quad \text{with } e = [1, \dots, 1]^T \in \mathbb{R}^{\#jobs}.$$

The merge step can be computed with a dense BLAS matrix vector product again. The complete approach is shown in Listing 3. If the matrices are stored in compressed sparse column storage (CSC), the resulting algorithm will be the same, but they work on the transposed matrix. That means `rowptr` and `colptr` change their roles and Listing 2 computes $y = A^T x$ and Listing 3 computes $y = Ax$.

I/O pool for the cache to disk technique. Certain kinds of parallelization can not be implemented using OpenMP easily, for example an implementation of a thread pool. When using the cache to disk technique to save memory we have to load the solvers from disk and have to wait until this is done. We accelerate this by loading the solver for iteration step $i+1$ in the background, while working with solver i. Depending on the amount of free memory we may also pre-load more than one solver to minimize delays caused by waiting times. Therefore we implement a thread pool, where we register the loading tasks for the solvers. The jobs in the pool are served in first-in-first-out order, i.e., the order they were inserted. The pool is created in the background, such that the program execution continues unharmed. To ensure that the solver is completly loaded, before we use it, all solvers have to provide an *is-loaded* property and a *wait-until-loaded* function. The *is-loaded* property is used to check whether the solver is already

Algorithm 9 Solution of multiple right hand sides in parallel.

Input: $L \in \mathbb{R}^{n\times n}$, $U \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$
Output: $X \in \mathbb{R}^{n\times m}$ with $LUX = B$

```
#prama omp parallel for shared(L,U,X,B) default(private)
for i = 1,...,m do
    solve Ly = B(:,i)
    solve UX(:,i) = y
end for
```

parallel techniques. On ther other hand, the columnwise variant solves systems $LUX(:,j) = V_i(:,j)$ for every j. Looking at how data depend on each other, we can see that L and U are read-only and the columns of the right hand side are independend from each other. Considering the options, OpenMP provides, the easiest way is to solve the systems for the different columns j in parallel. This can be done by inserting only one OpenMP directive, as shown in Algorithm 9.

Matrix-Vector-Products. Another basic operation is the matrix vector product. In the ADI iteration it is only needed if we solve the generalized equations. In the computation of the shift parameters and in the evaluation of the 2-norm residual by applying the power iteration, the fast computation of the matrix vector product is a key ingredient. For dense matrix calculation of $y = Ax$ is computed applying (multi-threaded)BLAS [5]. For sparse matrices it strongly depends on the storage format, how an efficient parallelization can be achieved. In the case, that the matrix is stored in compressed sparse row storage (CSR) format, the product is computed by evaluating scalar products

$$y_i = \sum_{k=1}^{n} A_{ik} b_k$$

of the rows of the matrix and the vector. Since the rows are data-independend, the computation can be distributed across the CPUs, like it is done for the spmv-LU. Listing 2 shows a short parallel C implementation.

The matrix vector product with a transpose matrix is another issue. The explicit transposition of the matrix should be avoided and the effect of the transpose should be exploited implicitly while computing $y = A^T x$. For matrices in CSR Storage this is a linear combination of the rows of A with coefficients from x. We

[5] provided by the CPU Vendor, the ATLAS/GotoBlas projects or via in the reference implementation at NetLib.

Algorithm 6 (Low-rank Cholesky factor ADI iteration (LRCF-ADI))

Input: F,G defining $FX + XF^T = -GG^T$ and shift parameters $\{p_1, \ldots, p_{i_{max}}\}$
Output: $Z = Z_{imax} \in \mathbb{C}^{n\times ti_{max}}$, such that $ZZ^H \approx X$

```
load solver for p_1
Solve (F + p_1 I)V_1 = sqrt(-2 Re(p_1)) G for V_1
free solver for p_1
write V_1 to a new file Z
for i = 2,3,...,i_max do
    load solver for p_i
    Solve (F + p_i I)Ṽ = V_{i-1} for Ṽ
    free solver for p_i
    V_i = sqrt(Re(p_i)/Re(p_{i-1})) (V_{i-1} - (p_i + conj(p_{i-1}))Ṽ)
    append V_i to file Z
end for
```

to slow. Another stopping criterion suggested in [8] is the relative change of the factor

$$\frac{||V_i||_F}{||Z_i||_F} \leq \varepsilon.$$

The computation of $||V_i||_F$ can be performed easily. To compute $||Z_i||_F$ we employ the definition of the Forbenius norm

$$||Z_i||_F = \sqrt{\sum_{k=1}^{n}\sum_{l=1}^{p} Z_{kl}^{(i)}}$$

and exploit $Z_i = [Z_{i-1}\ V_i]$:

$$\begin{aligned} ||Z_i||_F^2 &= \sum_{k=1}^{n}\sum_{l=1}^{p} Z_{kl}^{(i)} \\ &= \sum_{k=1}^{n}\sum_{l=1}^{p-m} Z_{kl}^{(i-1)} + \sum_{k=1}^{n}\sum_{l=1}^{m} V_{kl}^{(i)} \\ &= ||Z_{i-1}||_F^2 + ||V_i||_F^2 \end{aligned} \tag{6}$$

Thus we can easily compute this criterion by updating $||Z_i||_F^2$ without accessing Z_i. Column compression is another technique to save memory, but as for the 2-norm residual, we have to load the complete Z_i factor from disk to perform the column compression. Therefore column compression has to be postponed for post processing in this case. The solution of generalized systems can be done the same way. We need some additional memory to evaluate EV_{i-1}, though.

Algorithm 7 Single-Pattern-Multi-Value LU decomposition (sequential)

Input: $A \in \mathbb{R}^{n \times n}$, $p_i \in \mathbb{R}$ $i = 1, \ldots, n_p$
Output: L_i and U_i for $i = 1, \ldots, n_p$

1: compute $\mathcal{P}(L)$ and $\mathcal{P}(U)$ with $LU = A$ or $LU = A + E$
2: **for** $i = 1, \ldots, n_p$ **do**
3: $\quad \tilde{A} = A + p_i I$ or $\tilde{A} = A + p_i E$
4: $\quad$ compute $L_i U_i = \tilde{A}$ with Algorithm 5 using $\mathcal{P}(L)$ and $\mathcal{P}(U)$ as pattern.
5: **end for**

4. OpenMP Parallelization

Parallelization of algorithms has become an increasingly important topic since standard desktop and server CPUs have more than one core, support multi-threading and similar technologies to work in parallel. To accelerate an algorithm it is no longer feasible to use faster CPUs the way it has been done over the recent decades. We have to search for ways existing algorithms can exploit the new capabilities of modern CPUs or we have to develop new strategies to use them. A compiler add-on and runtime library which supports the programmer in solving this problem is OpenMP[4]. OpenMP defines a standard how compilers have to behave, when compiling your C or Fortran code. The actual parallelization is performed by inserting pre-processor statements into existing code together with a small set of library functions. It is also possible to design software in a way, that it can be compiled on systems, which do not support OpenMP without loss of functionality (besides the parallelization). Another advantage of OpenMP is, that we can parallelize code step by step and there is no problem using non parallel code, as it is the case for other parallel computation paradigms. The list of available compiler directives and library functions is described in the OpenMP Standard [14].

Parallelization of the SPMV LU. The first step to parallelize an algorithm is to check the data dependencies. The problem with the parallelization of an LU decomposition is, that every step depends on all previously computed values. So a direct parallelization of the reuse algorithm is nearly impossible or too expensive because we have to use graph theoretical algorithms to compute the elemination tree and to find subtree in this. In our case we have to compute, for a given set of shift parameters p_i, the factors of the systems $(A + p_i I)$ or $(A + p_i E)$. If we do this in a sequential way, this will look like in Algorithm 7. Looking on how $\mathcal{P}(L)$ and $\mathcal{P}(U)$ are accessed in Algorithm 5 (or the C variant in Listing 1), it is easy to see, that these sets are used read-only. That means the computations for

[4] http://www.openmp.org

Algorithm 8 Single-Pattern-Multi-Value LU decomposition (parallel)

Input: $A \in \mathbb{R}^{n \times n}$, $p_i \in \mathbb{R}$ $i = 1, \ldots, n_p$
Output: L_i and U_i for $i = 1, \ldots, n_p$

1: compute $\mathcal{P}(L)$ and $\mathcal{P}(U)$ with $LU = A$ or $LU = A + E$
2: #pragma omp parallel for shared($\mathcal{P}(L)$, $\mathcal{P}(U)$, A, E) default(private)
3: **for** $i = 1, \ldots, n_p$ **do**
4: $\quad \tilde{A} = A + p_i I$ or $\tilde{A} = A + p_i E$
5: $\quad$ compute $L_i U_i = \tilde{A}$ with Algorithm 5 using $\mathcal{P}(L)$ and $\mathcal{P}(U)$ as pattern.
6: **end for**

$(A + p_i I)$ and $(A + p_j I)$ $(i \neq j)$ are independent of each other. Therefore we can perform them in parallel. The parallelization idea is: Distribute the computation of L_i and U_i over all available computational units and work on the same $\mathcal{P}(L)$ and $\mathcal{P}(U)$ read-only. A `for`-loop with no data dependency can be parallelized in OpenMP easily with the `#pragma omp parallel for` statement. It realizes a workshare on all CPUs by splitting the iteration intervall in a number of disjoint sub-intervals equal to the number of available processing units. If the CPU supports HyperThreading®or similar techniques, these should be turned off, because half of the cores only exist in a virtual way and the memory bandwidth is insufficient to provide all required data on time. For example, if we have $n_p = 16$ and compute on a four-core CPU, every core has to factorize four systems. In practice, the splitting of the intervall is controlled by a load balancer, which can be configured, as well, but the default settings are usually sufficient. To specify that $\mathcal{P}(L)$ and $\mathcal{P}(U)$ should be the same for all jobs, we define `shared(`$\mathcal{P}(L)$`,` $\mathcal{P}(U)$`)` in the OpenMP directive. All other data is private for each job, so the default for the OpenMP block is set to `private`. The preparation of $\tilde{A} = A + p_i I$ is performed by each processor in its own memory to get independent of the other processors. The resulting parallel variant is shown in Algorithm 8.

Dealing with multiple right hand sides. Depending on the number of inputs and outputs of the system, we have to solve with $(A + p_i I)$ on multiple right hand sides. If the right hand side of the Lyapunov equation is a rank one dyad (i.e. spanned by only one column), all V_i are column vectors. In any other case we have to solve for multiple right hand sides in every step. To do this, we have two choices. Either we process the right hand side rowwise or columnwise. In the rowwise case, we compute for every value in L and U the corresponding values in the right hand side. The advantage of this solution is, that L and U are only loaded one time from memory. But depending on the size of the right hand side, the distance between two entries in the same row in the memory can get too large. In this case the computation runs out of cache permanently, which slows down the whole algorithm. Another problem is, that there is no easy way to apply