Updating Incomplete Factorization Preconditioners for Model Order Reduction

Hartwig Anzt[a,*], Edmond Chow[b], Jens Saak[c], Jack Dongarra[a,d,e]

[a] *Innovative Computing Lab, University of Tennessee, Knoxville, USA*
[b] *School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, USA*
[c] *Max Planck Institute for Dynamics of Complex Technical Systems, Magdeburg, Germany*
[d] *Oak Ridge National Laboratory, USA*
[e] *University of Manchester, UK*

Abstract

When solving a sequence of related linear systems by iterative methods, it is common to reuse the preconditioner for several systems, and then to recompute the preconditioner when the matrix has changed significantly. Rather than recomputing the preconditioner from scratch, it is potentially more efficient to update the previous preconditioner. Unfortunately, it is not always known how to update a preconditioner, for example, when the preconditioner is an incomplete factorization. A recently proposed iterative algorithm for computing incomplete factorizations, however, is able to exploit an initial guess, unlike existing algorithms for incomplete factorizations. By treating a previous factorization as an initial guess to this algorithm, an incomplete factorization may thus be updated. We use a sequence of problems from model order reduction. Experimental results using an optimized GPU implementation show that updating a previous factorization can be inexpensive and effective, making solving sequences of linear systems a potential niche problem for the iterative incomplete factorization algorithm.

Keywords: sequence of linear systems, preconditioner update, incomplete factorization, fine-grained parallelism, model order reduction, GPU

1. Introduction

The need to solve sequences of related linear systems arises in several contexts, for example, in the solution of nonlinear systems and time-dependent problems. When the linear systems are large and sparse, preconditioned Krylov subspace solvers are preferred. In this case, it is possible to "recycle" subspace information from solve to solve. Another way to exploit the fact that we are solving a sequence of systems is to reuse the preconditioner for several solves. When the preconditioner becomes stale, it is computed from scratch for the current linear system.

For certain types of preconditioners, it may be possible to update the stale preconditioner, to use for the current linear system, which may be more efficient than computing the preconditioner from scratch. In this paper, we consider updating incomplete factorization preconditioners. These preconditioners are popular because they are effective on a wide range of problems and are problem-independent (they do not need information from the problem besides the matrix). There has been little work on how to update incomplete factorization preconditioners. However, a recently proposed iterative algorithm for computing incomplete factorizations [1] is able to exploit an initial guess, unlike existing algorithms for incomplete factorizations. By treating a previous factorization as an initial guess to this algorithm, an incomplete factorization may thus be updated.

The recently proposed iterative algorithm for computing incomplete factorizations is completely different from the standard algorithm and offers fine-grained parallelism suitable for current hardware architectures, including graphics processing units (GPUs). Instead of a Gaussian elimination process, the new incomplete factorization algorithm computes an approximation to the factorization via a fixed-point iteration. As mentioned, a major advantage is that

*Corresponding author
Email address: hanzt@icl.utk.edu (Hartwig Anzt)

it is possible to compute a factorization starting from an initial guess. However, this feature of the algorithm has not been studied before.

We note that the idea of updating an incomplete factorization was introduced in [2], but these authors use different, much more expensive and less parallel algorithms than the one used in this paper, depending on sparse-sparse triangular solves (sparse triangular solves with many sparse right-hand sides).

To test our ideas, we use a realistic sequence of linear systems arising from the computation of transformation matrices in model order reduction (MOR). The topic of iteratively solving this sequence of problems in MOR has received only limited attention so far. Authors have studied the reuse of subspace information for different shift parameters in the sequence of coefficient matrices [3, 4, 5]. In contrast to this previous work, we focus on the reuse of preconditioner information for these sequences. Regarding parallel computations in MOR, the main development has been on sparse direct solution of the linear systems [6, 7, 8, 9] and matrix sign function approaches using dense matrices. For a selection of papers that either use the same model that we use in the numerical experiments, or also employ GPUs, see, e.g., [10, 11, 12].

This paper is structured as follows. Section 2 introduces model order reduction as one possible origin of a sequence of related linear systems. Section 3 provides background on the iterative algorithm for computing incomplete factorizations. Section 4 reports on experimental tests for solving the linear systems using the iteratively computed incomplete factorizations as preconditioners. We compare the case of updating the preconditioner from a previous factorization, and computing the preconditioner from scratch. We also experiment with reusing such preconditioners for more than one linear system. Section 5 concludes the paper.

2. Model Order Reduction and Shifted Linear Systems

Model order reduction is a technique for replacing a large and usually sparse dynamical system

$$E\dot{x}(t) = Hx(t) + Bu(t), \qquad y(t) = Cx(t), \tag{1}$$

where $E,\ H \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times p}$, and $C \in \mathbb{R}^{m\times n}$ ($m, p \ll n$), by a substantially smaller one of the form

$$\hat{E}\dot{\hat{x}}(t) = \hat{H}\hat{x}(t) + \hat{B}u(t), \qquad \hat{y}(t) = \hat{C}\hat{x}(t), \tag{2}$$

in order to perform fast simulations of the input-output behavior of the system, e.g., in control and optimization tasks. This means one searches for matrices $\hat{E},\ \hat{H} \in \mathbb{R}^{r\times r}$, $\hat{B} \in \mathbb{R}^{r\times p}$, and $\hat{C} \in \mathbb{R}^{m\times r}$ with $r \ll n$, with the property that when using the same input u in both systems, the deviation of the outputs $\|y - \hat{y}\|$ can be bounded in some suitable norm. In projection based MOR, the reduced order matrices are formed by a Petrov-Galerkin-projection approach, i.e.,

$$\hat{E} = W^T EV, \qquad \hat{H} = W^T HV, \qquad \hat{B} = W^T B, \qquad \hat{C} = CV. \tag{3}$$

Several reduction techniques exist that only differ in the way how the tall and thin matrices $V,\ W \in \mathbb{R}^{n\times r}$ are computed. For a textbook discussion of the system theoretically motivated methods see, e.g., the book by Antoulas [13]. Many of the system theoretic methods are in the one way or the other related to rational Krylov subspaces. This is very obvious for moment matching and rational interpolation methods like the prominent iterative rational Krylov algorithm (IRKA) [14]. In each step of IRKA, the matrices V and W are chosen as orthogonal bases of $\mathrm{span}\{(\sigma_1 E - H)^{-1}B, \ldots, (\sigma_r E - H)^{-1}B\}$ and $\mathrm{span}\{(\sigma_1 E - H)^{-1}C^T, \ldots, (\sigma_r E - H)^{-1}C^T\}$, respectively, and the set $\Sigma = \{\sigma_1, \ldots, \sigma_r\}$ is adapted for the next step until the method converges.

Other methods like Balanced Truncation [15, 16] use the solutions of two Lyapunov equations, the so called Gramians, as the basis for computing V and W. For very large n and small m and p, these Gramians typically have low numerical rank. This property can be used by modern solvers like the low rank alternating directions implicit (LR-ADI) iteration [17], or the rational (block-)Krylov subspace method (RKSM) [18] by representing the low rank factors of the Gramians in a rational Krylov basis.

Whenever rational Krylov subspaces are involved, the main effort in computing the basis is the solution of linear systems of equations of the form

$$(\sigma E - H)\, X = F, \tag{4}$$

or, with the notation $A_\sigma = (\sigma E - H)$,

$$A_\sigma X = F, \tag{5}$$

for varying shifts $\sigma \in \mathbb{C}$. Here, the matrices H and E are the large and sparse coefficients in (1), and X, F may consist of multiple columns. In the LR-ADI and IRKA cases, for example, the dimensions coincide with either those of B or C^T.

We focus on the linear systems of equations arising in the LR-ADI, when E is symmetric and positive definite (SPD), and H is symmetric and negative definite. This implies that the system (1) is asymptotically stable. Also, all appearing σ are real and positive, such that the matrix in (5) is SPD, and a preconditioned conjugate gradient (PCG) method can be used for fast iterative solution. An efficient preconditioner allows for reusing information across the sequence of problems. The iterative algorithm for computing incomplete factorizations that we describe next provides this feature by updating a previously computed factorization for varying shifts σ.

3. Iterative Incomplete Factorization Algorithm

An incomplete factorization is the approximate factorization of a nonsingular sparse matrix A into the product of a sparse lower triangular matrix L and a sparse upper triangular matrix U, i.e.,

$$A \approx LU,$$

where nonzeros or fill-in computed by a Gaussian elimination process is somehow truncated to a sparsity pattern, S. For an overview, see [19]. We use incomplete factorizations to precondition the matrices A_σ described in the previous section. A limitation to the use of these preconditioners is the difficult to parallelize factorization in the preconditioner setup phase [20, 21, 22, 23]. In particular, strategies providing some parallelism fail to leverage the fine-grained parallelism of current HPC architectures [24].

The recently proposed iterative algorithm for computing incomplete factorizations is different from previous approaches, as it does not try to parallelize the Gaussian elimination process [1]. Instead, it iteratively approximates the incomplete factors using the property

$$(LU)_{ij} = a_{ij}, \quad (i, j) \in S, \tag{6}$$

where $(LU)_{ij}$ denotes the (i, j) entry of the product of the computed factors L and U, and a_{ij} is the corresponding entry in matrix A. In particular, the iterative incomplete factorization algorithm computes the unknowns

$$\begin{aligned} l_{ij}, &\quad i > j, \quad (i, j) \in S, \\ u_{ij}, &\quad i \leq j, \quad (i, j) \in S \end{aligned}$$

using the constraints

$$\sum_{k=1}^{\min(i,j)} l_{ik} u_{kj} = a_{ij}, \quad (i, j) \in S \tag{7}$$

which corresponds to enforcing the property (6). The formulation

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right), \quad i > j, \tag{8}$$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}, \quad i \leq j \tag{9}$$

suggests solving for the unknowns using a fixed-point iteration $x = G(x)$ where x is the vector containing the unknowns l_{ij} and u_{ij} for $(i, j) \in S$. It can be proven that the iteration is locally convergent for standard (synchronous) and asynchronous iterations [25]. However, convergence for an arbitrary initial $x^{(0)}$ cannot be guaranteed. See [1] for more details on the convergence of this method.

Previously it was shown that scaling A symmetrically (as a preprocessing step) to have a unit diagonal is an aid to converging the fixed-point iterations [1]. Also, the fixed-point iterations may be started from an initial guess for x

formed from the lower and upper triangular parts of A. This is called the "standard initial guess". We will later define an alternative initial guess when computing a sequence of factorizations of related matrices.

Algorithm 1 gives the pseudocode for the fixed-point iterations for solving the equations (7) in the case of a SPD factorization, $A \approx U^T U$, which is called the "iterative incomplete Cholesky" or "iterative IC" algorithm. In the remainder of the paper we use this algorithm as we address sequences of SPD systems. Each "sweep" (a term we will use repeatedly in this paper) corresponds to one fixed-point iteration updating all the unknowns in the factor U. In contrast to the iterative IC algorithm, we call the standard Gaussian elimination-based incomplete Cholesky algorithm the "exact IC" algorithm, as it computes the exact IC factors.

Algorithm 1: Iterative IC Algorithm

```
1  Set unknowns u_ij to initial values
2  for sweep = 1, 2, ... until convergence do
3      parallel for (i, j) ∈ S_U do
4          s = a_ij − Σ_{k=1}^{i−1} u_ki u_kj
5          if i ≠ j then
6              u_ij = s / u_ii
7          else
8              u_ii = √s
9          end
10     end
11 end
```

We use a GPU implementation of Algorithm 1 described in [26]. In this implementation, subsets of the components of x are assigned to GPU thread blocks that are scheduled onto the GPU multiprocessors. Each thread block updates the components of x assigned to it in parallel (in Jacobi-like fashion). Usually, there are more thread blocks than multiprocessors, which implies that some thread blocks are processed before others, and thus update their components of x before others. Every thread block uses the latest available values for components outside its own thread block. Hence, some updates within one fixed-point sweep may use newer data than others (in Gauss–Seidel-like fashion). As there is in general no pre-defined order in which thread blocks are scheduled onto multiprocessors, the iteration scheme may be considered as "block-asynchronous" [27]. In [26], several unconventional optimization strategies are proposed that allow for better reuse of GPU cache and control of update order, important for the overall efficiency of the fixed point iteration.

Based on this previous study, we use the following strategies as the default setup for the iterative IC algorithm applied in this paper:

- The CUDA kernel performs one sweep of the iterative IC algorithm. Additional sweeps are performed by launching this kernel multiple times.
- The inputs to the kernel are arrays corresponding to matrices A and U. The input value of U is the initial guess for the U factor. The output value of U is the computed incomplete factor.
- We use the reverse Cuthill-McKee (RCM) ordering for the system matrix, which is a preferred ordering for incomplete factorizations [28]. Further, RCM ordering reduces the bandwidth of the systems, which is beneficial for cache reuse in the iterative IC algorithm [26].
- The linear systems are scaled such that A has a unit diagonal, which improves the convergence of the iterative IC algorithm.

4. Experimental Results

In Section 4.1, we describe a sequence of linear systems arising from a model order reduction problem that serves as our benchmark application. In Section 4.2, we show the performance of the iterative IC preconditioner applied to solving these linear systems. Sections 4.3 and 4.4 are the main contribution of this paper, showing results for the case

of computing a factorization by updating a previous factorization. Section 4.5 demonstrates how sparse triangular solves can also be performed efficiently in parallel for these problems.

The experimental platform is a two socket Intel Xeon E5-2670 (Sandy Bridge) system accelerated by an NVIDIA Tesla K40c GPU. The host system has a theoretical peak of 333 GFLOP/s and the GPU has a theoretical peak of 1,682 GFLOP/s (double precision). The main memory size is 64 GB with a theoretical peak memory bandwidth of 51 GB/s. On the K40 GPU, 12 GB of main memory is available at a theoretical peak bandwidth of 288 GB/s. The implementation of the iterative incomplete factorization GPU kernels are implemented in CUDA [29] version 6.0 [30] using a default thread block size of 128. The PCG linear solver is from the MAGMA open-source software library, version 1.6.0 [31]. It is able to solve with multiple right-hand sides simultaneously. The sparse triangular solve routines are taken from the NVIDIA cuSPARSE library [32]. All computations use double precision arithmetic.

4.1. Benchmark application

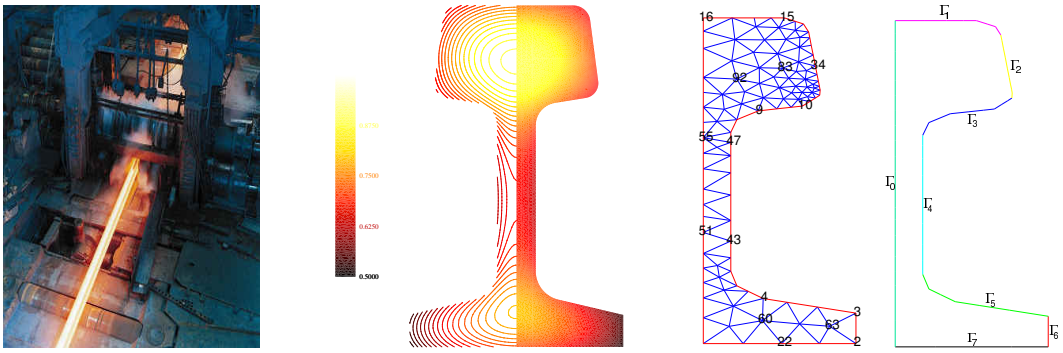

Figure 1: The steel cooling example: process picture (left), sample heat distribution (center), initial grid and boundaries (right).

The benchmark application is the cooling process in rolling mills for steel rail production. A uniform temperature distribution across cross-sections is needed before applying the final rolling step, to avoid degrading material properties. A uniform temperature distribution can be generated via applying active cooling to the surface of the rail. To this end, the boundary of the cross-section is split into seven disjoint parts, where different controls can be applied. These controls are realized as coolant sprays with different temperatures or intensities. Key for a dynamic cooling process is the use of real-time temperature estimators. For this purpose, model order reduction is applied to a spatial discretization using finite elements (see Figure 1) of the partial differential equation model. In this model, both the stiffness and mass matrices are symmetric, so the PCG solver can be employed to solve equation (5). The shift parameters are chosen according to the optimal procedure suggested for the ADI method by Wachspress (see, e.g., [33]).

The discretization we use is a re-implementation of the steel profile example [34] from the Oberwolfach benchmark collection for model order reduction using the FEniCS [35] finite element software package in version 1.2.0. Like in the original benchmark, we discretize using increasing mesh granularities via global bisection refinement. For every granularity, we obtain a set of matrices H, E and F for equation (4). In Table 1 we list some key properties for these matrices.

The main effort (usually more than 90% of the execution time in the case of the LR-ADI iteration mentioned in Section 2) towards the computation of the projection matrices V and W (see equation (3)) is to solve equation (5) for different shifts σ and block right-hand sides, each containing 7 vectors. We attempt to accelerate this stage of the MOR process by providing a fast preconditioning strategy for the sequence of shifted linear systems. The length of the sequence of shifts, as well as the shift values themselves, depend on the discretization, the accuracy bound we demand for the model order reduction, and the method we employ for computing V and W. In this paper, we use shifts according to a bound of 10^{-5} for the error $\|y - \hat{y}\|$ when using the same input u in the original and the reduced system and applying LR-ADI based Balanced Truncation. Figure 2 (left) shows the shift values for the different discretizations. The variation of the matrix condition numbers along the sequence of shifts shows a characteristic pattern which is similar for the different discretizations. The condition numbers are plotted for the example problem DISC_5 in Figure 2 (right).

problem	size n	$nnz(A-\sigma E)$	# shifts σ_i	$\max_i(\sigma_i)$	$\min_i(\sigma_i)$	$\min_i(\kappa(\sigma_i E-A))$	$\max_i(\kappa(\sigma_i E-A))$
DISC_1	371	2,343	22	1.0233e+00	7.9033e-05	3.57e+01	5.58e+02
DISC_2	1,357	8,997	25	4.5907e+00	7.9050e-05	4.67e+01	2.46e+03
DISC_3	5,177	35,241	28	1.9973e+01	7.9046e-05	5.26e+01	1.05e+04
DISC_4	20,209	139,473	30	8.3472e+01	7.9037e-05	5.47e+01	4.01e+04
DISC_5	79,841	554,913	33	3.4157e+02	7.9016e-05	5.53e+01	1.77e+05
DISC_6	317,377	2,213,697	35	1.3896e+03	7.9814e-05	5.56e+01	7.10e+05
DISC_7	1,265,537	8,842,881	39	5.6426e+03	1.2248e-05	5.58e+01	2.44e+06

Table 1: Properties of the discretizations of the rail problem, showing matrix dimension n, number of nonzeros nnz, number of shifts or matrices in the sequence, the maximum and minimum shift values, and the minimum and maximum condition numbers of matrices in the sequence.

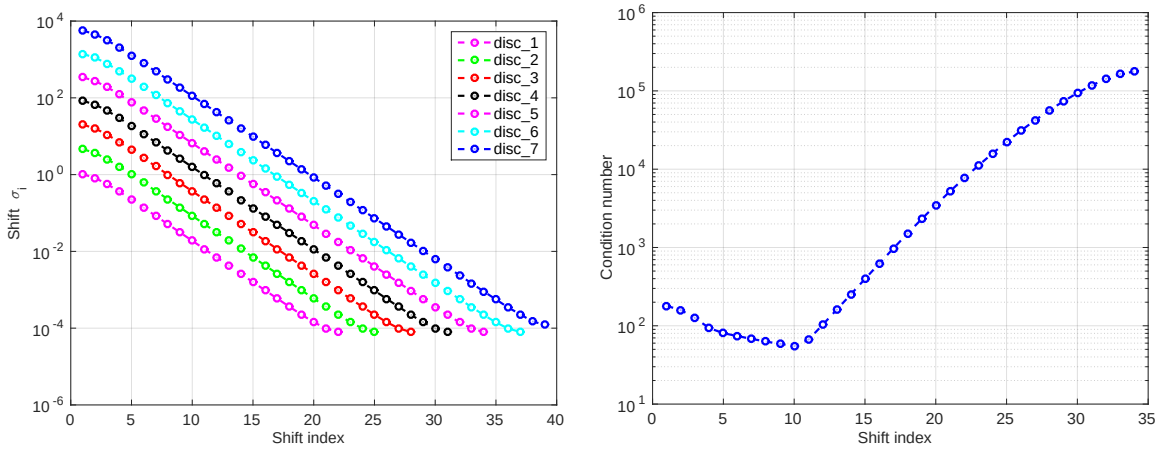

Figure 2: Plot of the sequence of shifts σ_i for the distinct discretizations (left) and the condition numbers for the matrices $(\sigma_i E - A)$ for the DISC_5 problem (right).

4.2. Results on iterative incomplete Cholesky factorization

In this section, we first consider the entire set of shifted matrices for DISC_5 and show the PCG iteration counts when using IC preconditioners computed approximately using the iterative IC algorithm. The right-hand side of each system is the block of 7 vectors denoted as matrix F in equation (5). The systems were diagonally scaled on the left and right such that the matrices remain symmetric but have unit diagonal. The iterations start with a zero initial guess, and they are stopped when the residual norm relative to the initial residual norm has been reduced below 10^{-6} for the first vector. We always use the level 0 incomplete Cholesky factorization in this paper, but this is not a limitation of the iterative IC algorithm. We recall that the standard initial guess is used for computing the iterative IC factorization, which is the upper triangular part of the scaled coefficient matrix.

Table 2 shows the PCG iteration counts when the iteratively computed IC factorization, using 0 to 5 sweeps, is used as a preconditioner. (Zero sweeps corresponds to only using the initial guess, and which corresponds to using symmetric Gauss–Seidel preconditioning.) Solver iteration counts are also shown for the incomplete Cholesky preconditioner computed exactly using a Gaussian elimination process (labelled "Exact IC"), and for no preconditioning (labelled "No Prec"). The results for the iterative IC algorithm are averaged over 10 runs. Figure 3 plots the PCG iteration count data for up to 3 sweeps of the iterative IC algorithm. The most important observation is that very few sweeps are needed to produce preconditioners that yield PCG iteration counts close to those yielded by exact IC factorization.

To understand the behavior of the iterative IC algorithm, Figure 4 plots the relative nonlinear residual norm,

$$\left\| \left(A - \left(U^{(k)}\right)^T U^{(k)} \right)_S \right\|_F \Big/ \left\| \left(A - \left(U^{(0)}\right)^T U^{(0)} \right)_S \right\|_F ,$$

and the relative IC residual norm,

$$\left\| A - \left(U^{(k)}\right)^T U^{(k)} \right\|_F \Big/ \left\| A - \left(U^{(0)}\right)^T U^{(0)} \right\|_F ,$$

shift σ_i	No Prec	Exact IC	Number of sweeps 0	1	2	3	4	5
-3.42e+02	13	5	5	5	5	5	5	5
-2.74e+02	13	4	5	5	4	4	4	4
-1.90e+02	13	4	5	4	4	4	4	4
-1.22e+02	12	4	5	4	4	4	4	4
-7.60e+01	12	4	5	4	4	4	4	4
-4.68e+01	11	4	4	4	4	4	4	4
-2.87e+01	13	4	5	4	4	4	4	4
-1.76e+01	17	5	6	5	5	5	5	5
-1.08e+01	22	6	8	7	6	6	6	6
-6.58e+00	27	8	10	8	8	8	8	8
-4.02e+00	35	10	12	10	10	10	10	10
-2.46e+00	44	12	15	13	12	12	12	12
-1.50e+00	55	15	19	16	15	15	15	15
-9.19e-01	69	18	24	20	19	18	18	18
-5.62e-01	85	22	30	24	23	22	22	22
-3.44e-01	105	27	36	29	28	27	27	27
-2.10e-01	129	33	45	36	34	33	33	33
-1.28e-01	161	39	56	44	41	40	40	39
-7.86e-02	195	47	67	54	50	48	47	47
-4.80e-02	233	56	80	65	60	57	57	56
-2.94e-02	280	66	96	79	72	69	67	67
-1.80e-02	330	78	113	93	83	81	80	80
-1.10e-02	391	89	134	104	100	97	94	92
-6.72e-03	440	109	151	130	119	108	107	107
-4.11e-03	528	128	180	147	138	133	131	130
-2.51e-03	630	150	215	178	162	156	154	153
-1.54e-03	707	166	241	199	181	173	170	169
-9.40e-04	796	190	271	226	206	198	195	193
-5.77e-04	903	216	305	252	230	224	221	220
-3.55e-04	1000+	244	349	290	264	254	250	248
-2.21e-04	1000+	256	381	315	283	270	264	261
-1.42e-04	1000+	279	438	363	329	301	292	289
-9.85e-05	1000+	313	451	378	341	327	322	320
-7.90e-05	1000+	319	464	382	345	334	329	326
$\sum_{\sigma_i}$:	11269+	2930	4231	3500	3193	3055	3004	2982
relative to IC:	3.85+	1.00	1.44	1.19	1.09	1.04	1.03	1.02

Table 2: Solver iteration counts using the preconditioner generated via an exact IC factorization or with up to 5 sweeps of the iterative counterpart. The linear systems are the set of shifted matrices for the DISC_5 discretization.

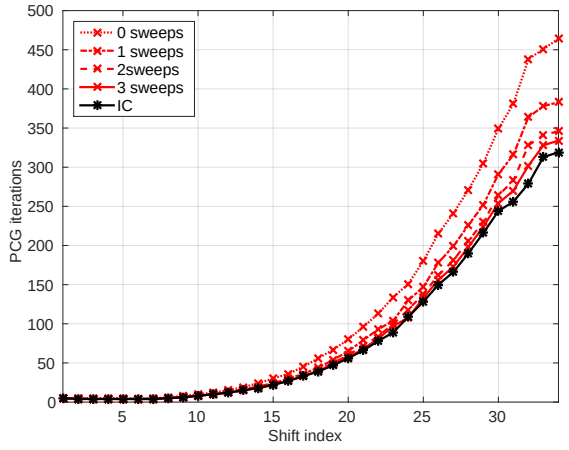

Figure 3: PCG iteration counts for DISC_5 using an exact IC factorization or up to 3 sweeps of the iterative IC algorithm starting with the standard initial guess.

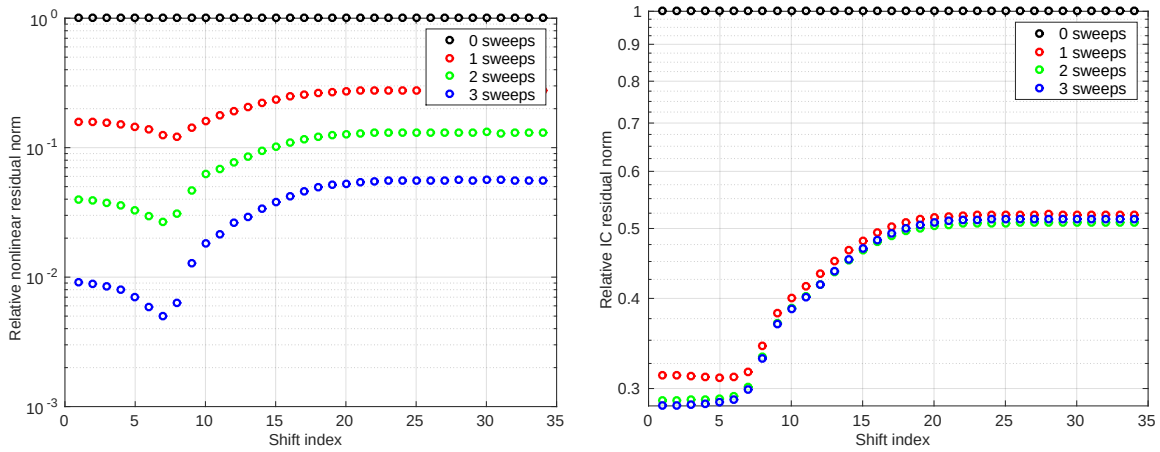

Figure 4: Relative nonlinear residual norm (left) and the relative IC residual norm (right) for the DISC_5 problems for different shifts σ_i after applying 1, 2, and 3 sweeps of the iterative IC algorithm.

corresponding to Figure 3, where $U^{(k)}$ denotes the IC approximation after k sweeps and $U^{(0)}$ is the initial value for this approximation. Note that the Frobenius norm in the relative nonlinear residual norm is only over the sparsity pattern S; values outside this sparsity pattern are considered to be zero for the Frobenius norm. Note also that the relative IC residual norm is generally nonzero for an incomplete factorization (it is zero for a full factorization).

Figure 4 (left) shows that after three sweeps the nonlinear residual norm has only been reduced 1–2 orders of magnitude. Since we saw in Figure 3 that using three sweeps was sufficient for good preconditioning, we conclude that it is not necessary to converge the nonlinear residual norm to a high tolerance in order to produce useful preconditioners. Figure 4 (right) shows that the IC residual norm, however, converges after a very small number of sweeps.

Referring back to Figure 3, it can be observed the factorizations converge faster or slower for different values of the shift. More precisely, fewer sweeps of the iterative IC algorithm are needed to match the exact IC results for low shift indices than for high shift indices. Generally, matrices earlier in the sequence are easier to solve than matrices later in the sequence. This roughly corresponds to the condition numbers of the matrices shown in Figure 2 (right). Although the convergence of the iterative IC algorithm is related to the norm of the Jacobian of the fixed-point iteration matrix, these Jacobians have smaller norms if the system matrix have large diagonal elements relative to off-diagonal elements [1], which in turn could be related to the condition number. In any case, connections between matrix properties and the convergence of the iterative IC algorithm are important to make.

Although we have only shown results DISC_5, the results are qualitatively similar for other discretizations. Figure 5 shows the convergence of the relative nonlinear residual norm and the relative IC residual norm for one system for each problem size. Each system corresponds to the last shift of each sequence, which corresponds to the hardest problem of that sequence. The results show only negligible differences in the convergence of the two different residual norms for different problem sizes.

The iterative IC algorithm is much faster than the conventional IC algorithm on highly parallel architectures such as GPUs. Table 3 lists the runtimes for the NVIDIA cuSPARSE [36] implementation of IC, and for 1, 3, and 5 sweeps of the iterative IC algorithm, for different sizes of our MOR problem (the shift used is immaterial here). The results show very significant speedups over the cuSPARSE factorization code, even when the iterative IC algorithm uses 5 sweeps. Note that 3 sweeps typically generate a preconditioner of comparable quality as the exact IC factorization, as shown earlier. We also note that for the iterative IC kernel for smaller problems, the GPU kernel launch overhead is a significant fraction of the total time.

4.3. Results on updating the factorization in the MOR problem sequence

The tests shown so far have solved each shifted system independently, using the standard initial guess (which we now abbreviate as "SIG") for the iterative IC factorization. We now address solving each shifted system in sequence. For simplicity, we use the same set of shifted systems for the DISC_5 problem presented in Section 4.1. As shown in

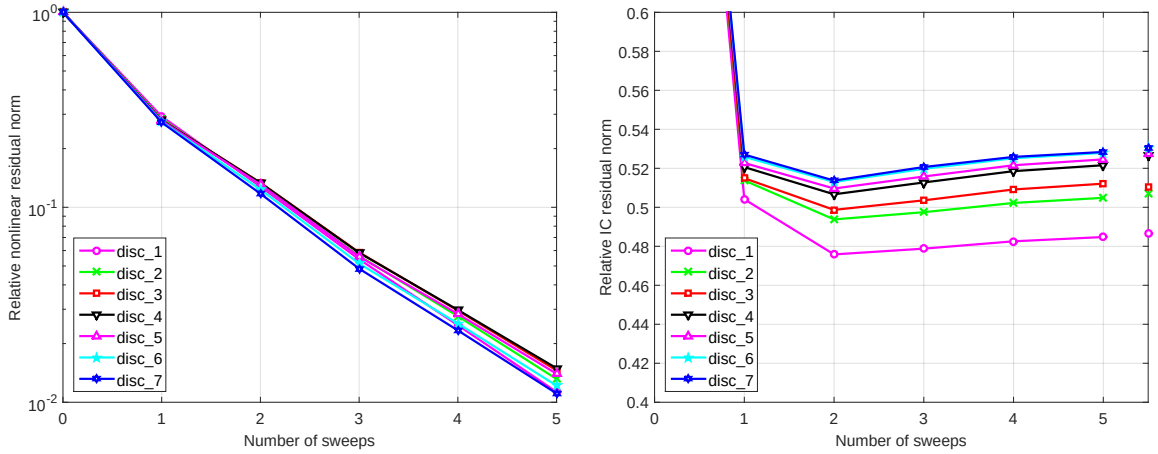

Figure 5: Convergence for the last shifted matrix in each sequence (different mesh sizes) of the relative nonlinear residual norm (left) and the relative IC residual norm (right). The relative IC residual norms of the exact IC factorizations are shown in the right margin of the graph.

	Time [ms]				Speedup		
		Number of sweeps			Number of sweeps		
	IC	1	3	5	1	3	5
DISC_1	4.88	0.03	0.06	0.08	167.70	87.14	57.48
DISC_2	9.01	0.03	0.06	0.08	309.62	158.07	107.39
DISC_3	15.60	0.04	0.08	0.12	421.62	200.00	134.48
DISC_4	32.00	0.08	0.19	0.30	405.58	167.54	105.26
DISC_5	66.40	0.23	0.62	1.01	286.21	106.58	65.74
DISC_6	142.00	0.78	2.25	3.73	182.75	63.11	38.07
DISC_7	323.00	2.94	8.77	14.50	109.86	36.83	22.28

Table 3: Time [ms] for the exact incomplete Cholesky factorization using NVIDIAs cuSPARSE library [32] and 1, 3, and 5 sweeps of the iterative counterpart – and the respective speedups.

Figure 2, the systems generally increase in difficulty for increasing shift index, but the shifts also decrease in size, making the systems with high shift index more similar to each other than the systems with low shift index.

To solve the shifted matrices in sequence, we begin by assuming that we have a factorization of the first matrix. We use an exact IC factorization for this, although an inexact factorization computed using the iterative IC procedure can also be used. Then, each subsequent matrix is factorized by updating the factorization of the previous matrix using some number of sweeps.

Figure 6 shows the PCG iteration counts when the iterative IC factorization was computed using this procedure (which we call using the previous factorization initial guess, or "PFIG"). The results using SIG and exact IC are also shown for comparison. In the Figure, each graph shows the results using 0, 1, or 2 sweeps. PFIG using 0 sweeps corresponds to using the exact IC factorization for the first system to precondition the entire sequence of problems. As can be observed, these PCG iteration counts depart rapidly from those for exact IC.

PFIG with 1 sweep gives results very close to those of exact IC computed for every shift. SIG with 1 sweep does not match exact IC for the harder problems with high shift indices. Assuming 1 sweep, the total cost for using PFIG is one exact factorization for the first matrix, and one sweep per each additional matrix. In summary, PFIG using 1 sweep appears sufficient for good performance when an update is performed for every shift. SIG using even 2 sweeps does not match this level of performance.

4.4. Reusing the factorization for many systems

An incomplete factorization preconditioner may be reused for more than one system in a sequence, which can save computations. In this section, we show results for DISC_5 from computing an incomplete factorization for every 3rd, 5th, and 7th system, i.e., the factorization is reused for 3, 5, or 7 consecutive systems. Below, we use ℓ to denote the recomputation interval. In practice, degradation in the PCG iteration count would be used for triggering recomputation of the factorization.

Exact and iterative IC factorizations were computed every ℓ systems, and were used to precondition ℓ consecutive systems. The iterative IC factorizations were computed using a single sweep applied to either the standard or previous factorization initial guesses (SIG or PFIG). In the case of PFIG, larger ℓ means that the factorization used as initial guess for the current factorization is more "stale" or numerically farther. It could be expected that for larger ℓ, the PFIG updating strategy may be worse than the SIG strategy without updating.

Figure 7 shows the PCG iteration counts due to these preconditioners, for $\ell = 3, 5, 7$. The right side graphs zoom in on the left side graphs. A typical sawtooth pattern can be observed, as the PCG iteration count degrades as a factorization is reused, and then improves again when it is recomputed.

From the graphs on the left side of Figure 7, fewer total number of PCG iterations are needed when factorizations are updated using PFIG, compared to when factorizations are computed from scratch using SIG. When the recomputation interval is larger, e.g., $\ell = 7$, the overall cost decreases (due to requiring fewer total number of sweeps) but the benefit of PFIG also decreases compared to SIG. However, even for $\ell = 7$, the PFIG strategy is better than the SIG strategy.

It can also be observed that PFIG is better when the difference between shifts is small (high shift indices), since previous factorizations are better guesses when the difference is small, but SIG is *slightly* better when the difference between shifts is large (low shift indices; see zoomed graphs on right side). The cutoff between SIG being better and PFIG being better is about 15 for $\ell = 3$, about 20 for $\ell = 5$, and about 28 for $\ell = 7$. The cutoff increases for larger ℓ which could be expected.

A subtle but interesting feature that can also be noticed is that the degradation in PCG iteration counts when the iterative IC preconditioners are reused is slightly worse than when exact IC preconditioners are reused. This is a very small effect, but can be observed, for example, in the $\ell = 7$ case between indices 8 and 14.

4.5. Approximate sparse triangular solves with block right-hand sides

Although not the main focus of this paper, in this section we address the problem of applying the preconditioner in parallel, i.e., the problem of parallel sparse triangular solves. The MOR application requires solving systems with a block of 7 right-hand sides. Significant research has focused on level scheduling and other parallelization approaches for single right-hand sides [37, 38, 39, 40] that could easily be extended to block problems. On GPUs, it is natural to use NVIDIA's cuSPARSE function for level-scheduled triangular solves for a block of vectors.

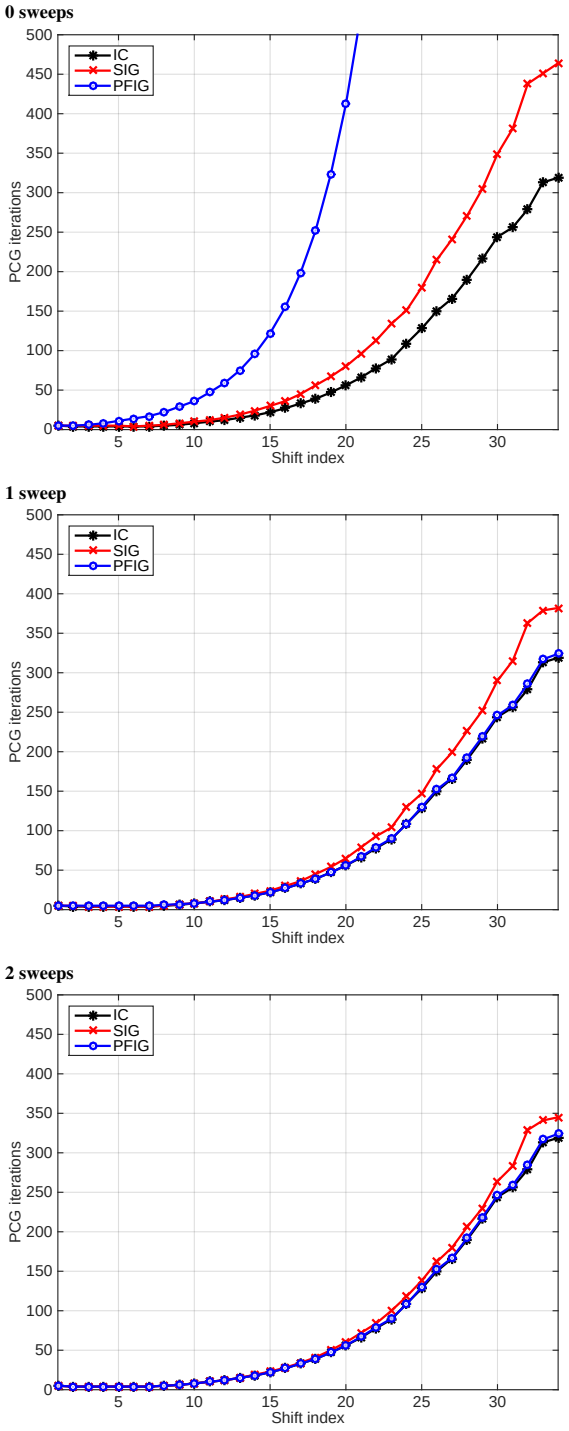

Figure 6: PCG iteration counts using a preconditioner generated after 0, 1, and 2 sweeps of the iterative IC algorithm. The 'IC' plot uses the exact IC factorization for every system of the sequence. The 'SIG' plot computes the factorization using a given number of sweeps on the standard initial guess. The 'PFIG' plot computes the factorization using a given number of sweeps on the previous factorization. In this case, the IC factorization of the first matrix of the sequence was computed exactly.

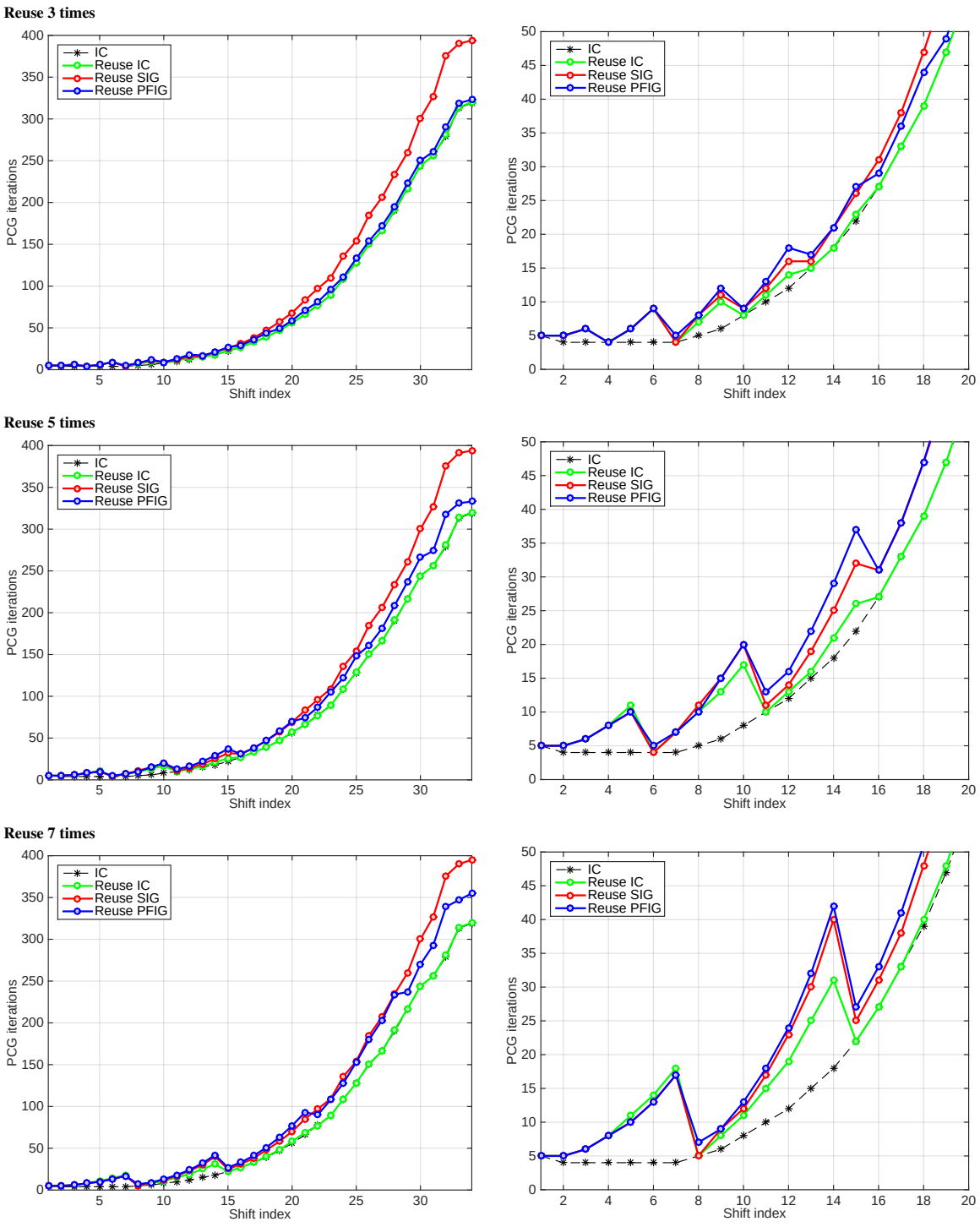

Figure 7: PCG iteration counts when each factorization is computed every ℓ systems and reused for ℓ consecutive systems, for $\ell = 3, 5, 7$. Right side graphs zoom in on the left side graphs. A single sweep was used in the SIG and PFIG cases. "IC" denotes the case where the exact IC factorization is used for every system.

		Exact	Number of Jacobi iterations for approximate triangular solves							
		triangular solves	1	2	3	4	5	6	7	max. speedup
DISC_1	$\sum_{\sigma_i}$ iterations:	216	404	273	239	227	198	200	218	
	$\sum_{\sigma_i}$ time [s]:	0.72	0.59	0.44	0.39	0.39	0.38	0.40	0.42	1.89
DISC_2	$\sum_{\sigma_i}$ iterations:	405	844	578	479	440	361	409	372	
	$\sum_{\sigma_i}$ time [s]:	2.27	1.28	0.92	0.81	0.76	0.75	0.77	0.80	3.02
DISC_3	$\sum_{\sigma_i}$ iterations:	751	1688	1185	954	842	794	742	765	
	$\sum_{\sigma_i}$ time [s]:	8.13	2.72	2.03	1.72	1.59	1.60	1.62	1.68	5.11
DISC_4	$\sum_{\sigma_i}$ iterations:	1323	3058	1946	1836	1536	1356	1409	1398	
	$\sum_{\sigma_i}$ time [s]:	30.85	6.28	5.04	4.79	4.48	4.50	4.84	5.23	6.89
DISC_5	$\sum_{\sigma_i}$ iterations:	2935	7169	5443	4162	3612	3483	3200	3226	
	$\sum_{\sigma_i}$ time [s]:	158.70	27.97	26.49	24.09	24.41	26.87	27.49	30.73	6.59
DISC_6	$\sum_{\sigma_i}$ iterations:	5904	15005	11353	9016	7469	7139	7457	6282	
	$\sum_{\sigma_i}$ time [s]:	785.55	156.36	159.24	158.56	158.48	177.82	211.90	201.32	4.96
DISC_7	$\sum_{\sigma_i}$ iterations:	10501	26803	19625	16523	14030	12358	12132	12381	
	$\sum_{\sigma_i}$ time [s]:	3207.04	982.42	995.78	1074.86	1116.08	1149.81	1306.34	1502.27	3.26

Table 4: Comparison of iteration counts (first line) and timings [s] (second line) of PCG solving the complete sequence of problems and using different methods of solving with the IC factors: exact triangular solves with cuSPARSE or approximate triangular solves using a fixed number of Jacobi steps.

However, following [1], which experimented on Intel's MIC architecture, we replace exact sparse triangular solves used as the preconditioning operation, by approximate triangular solves using a small number of Jacobi iterations. For our purpose, we extended the Jacobi method to a block version that is able to iterate multiple vectors simultaneously. In Table 4, we sum the number of PCG iterations needed to solve the sequence of linear systems for each discretization. We compare the use of approximate triangular solves with the cuSPARSE block triangular solves mentioned above. The factorizations were constructed using 1 update sweep of the iterative IC algorithm started with the previous factorization initial guess, where the updates are performed for every shift.

Using only a few Jacobi iterations results in a less accurate preconditioner, and generally requires a larger number of PCG iterations. However, the total time for the iterations is generally improved. The tradeoff between the speed and accuracy of these solves is problem-dependent in general. The iterative triangular solves, however, accelerate the solution process for all considered cases. The optimal choice of Jacobi iterations results in up to 7× speedup. Speedup would be less in a method like GMRES compared to PCG since the former has more work per iteration outside the preconditioner.

5. Conclusions

This paper demonstrated the use of an iterative algorithm for computing incomplete factorizations as part of solving a sequence of linear systems from model order reduction. The iterative algorithm can exploit an initial guess for the factorization. We tested the use of a previous factorization as the initial guess. While this initial guess is generally effective, it is not always better than using a standard initial guess when the factors to be updated correspond to a matrix that is numerically far from the current matrix to be factored.

For the MOR benchmark problem, we have only addressed solving one set of linear systems. This reflects, e.g., the case of the LRADI iteration (Section 2) with a pre-computed set of shifts. In the LRADI with an adaptive shift procedure, or the IRKA approach (Section 2), a much smaller batch of shifts is available at once, but the values of the shifts change as they are optimized. In the case of IRKA that means that systems for different shifts can be solved simultaneously, while additional systems for other shifts only become available after solves are performed. For the LRADI the systems for one batch still need to be solved sequentially, but still preconditioners could be computed or updated as soon as the batch becomes available. In this bigger picture, different possibilities for parallelism exist, depending on the available parallel resources.

A general protocol for solving sequences of linear systems involves reuse of the preconditioner and occasional updates or recomputation, the schedule for which will depend on how quickly the matrices change from one to the next. Reuse of the preconditioner may also be combined with other techniques, such as recycling of Krylov subspaces. This paper shows that an iterative algorithm for computing incomplete factorizations can be an important part of solving sequences of linear systems on highly parallel computers.

Acknowledgments

The authors thank Maximilian Behr for providing the FEniCS based matrices that served as the benchmark application. This material is based upon work supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Numbers DE-SC-0012538 and DE-SC-0010042, the Ministry of Education and Science of the Russian Federation (Agreement N 14.607.21.0006, unique identifier RFMEFI57714X0020), and NVIDIA.

[1] E. Chow, A. Patel, Fine-grained parallel incomplete LU factorization, SIAM Journal on Scientific Computing 37 (2015) C169–C193.

[2] C. Calgaro, J.-P. Chehab, Y. Saad, Incremental incomplete LU factorizations with applications, Numerical Linear Algebra with Applications 17 (5) (2010) 811–837. doi:10.1002/nla.756.
URL `http://dx.doi.org/10.1002/nla.756`

[3] C. A. Beattie, S. Gugercin, S. Wyatt, Inexact solves in interpolatory model reduction, Linear Algebra Appl. 436 (8) (2012) 2916 – 2943. doi:10.1016/j.laa.2011.07.015.

[4] M. I. Ahmad, D. B. Szyld, M. B. van Gijzen, Preconditioned multishift BiCG for H2-optimal model reduction, Research Report 12-06-15, Department of Mathematics, Temple University (2012).

[5] M. Baumann, M. B. van Gijzen, Nested Krylov methods for shifted linear systems, Report 14-01, Delft Institute of Applied Mathematics, Delft University of Technology (2014).

[6] J. M. Badía, P. Benner, R. Mayo, E. S. Quintana-Ortí, Parallel algorithms for balanced truncation model reduction of sparse systems, in: J. J. Dongarra, K. Madsen, J. Wasniewski (Eds.), Applied Parallel Computing: 7th International Conference, PARA 2004, Lyngby, Denmark, June 20-23, 2004, no. 3732 in Lecture Notes in Comput. Sci., Springer-Verlag, Berlin, Heidelberg, New York, 2006, pp. 267–275.

[7] J. M. Badía, P. Benner, R. Mayo, E. S. Quintana-Ortí, G. Quintana-Ortí, A. Remón, Balanced truncation model reduction of large and sparse generalized linear systems, Technical report Chemnitz Scientific Computing Preprints 06-04, Fakultät für Mathematik, TU Chemnitz (Nov. 2006).

[8] M. Köhler, J. Saak, Efficiency Improving Implementation Techniques for Large Scale Matrix Equation Solvers, Chemnitz Scientific Computing Prep. CSC 09-10, TU Chemnitz (2009).

[9] M. Köhler, J. Saak, A shared memory parallel implementation of the IRKA algorithm for $\mathcal{H}_2$ model order reduction, in: P. Manninen, P. Öster (Eds.), Applied Parallel and Scientific Computing, Vol. 7782 of Lecture Notes in Comput. Sci., Springer-Verlag, Berlin/Heidelberg, 2013, pp. 541–544. doi:10.1007/978-3-642-36803-5_42.

[10] J. M. Badía, P. Benner, R. Mayo, E. S. Quintana-Ortí, G. Quintana-Ortí, J. Saak, Parallel order reduction via balanced truncation for optimal cooling of steel profiles, in: J. C. Cunha, P. D. Medeiros (Eds.), Euro-Par 2005 Parallel Processing, Vol. 3648 of Lecture Notes in Comput. Sci., Springer-Verlag, Berlin, Heidelberg, New York, 2005, pp. 857–866. doi:10.1007/11549468_93.

[11] P. Benner, P. Ezzatti, E. S. Quintana-Ortí, A. Remón, Using hybrid CPU-GPU platforms to accelerate the computation of the matrix sign function, in: H.-X. Lin, M. Alexander, M. Forsell, A. Knüpfer, R. Prodan, L. Sousa, A. Streit (Eds.), Euro-Par 2009 – Parallel Processing Workshops, Vol. 6043 of Lecture Notes in Comput. Sci., Springer-Verlag, Berlin/Heidelberg, 2010, pp. 132–139. doi:10.1007/978-3-642-14122-5_17.

[12] P. Benner, P. Ezzatti, D. Kressner, E. S. Quintana-Ortí, A. Remón, Accelerating model reduction of large linear systems with graphics processors, in: K. Jónasson (Ed.), Applied Parallel and Scientific Computing, Vol. 7134 of Lecture Notes in Comput. Sci., Springer-Verlag, Berlin/Heidelberg, 2012, pp. 88–97. doi:10.1007/978-3-642-28145-7_9.

[13] A. C. Antoulas, Approximation of Large-Scale Dynamical Systems, SIAM Publications, Philadelphia, PA, 2005.

[14] S. Gugercin, A. C. Antoulas, C. Beattie, $\mathcal{H}_2$ model reduction for large-scale dynamical systems, SIAM Journal on Matrix Analysis and Applications 30 (2) (2008) 609–638.

[15] M. S. Tombs, I. Postlethwaite, Truncated balanced realization of a stable nonminimal state-space system, Internat. Journal of Control 46 (4) (1987) 1319–1330.

[16] A. J. Laub, M. T. Heath, C. C. Paige, R. C. Ward, Computation of system balancing transformations and other applications of simultaneous diagonalization algorithms, IEEE Trans. Automat. Control 32 (2) (1987) 115–122.

[17] P. Benner, J.-R. Li, T. Penzl, Numerical solution of large Lyapunov equations, Riccati equations, and linear-quadratic control problems, Numer. Lin. Alg. Appl. 15 (9) (2008) 755–777.

[18] V. Druskin, L. Knizhnerman, V. Simoncini, Analysis of the rational Krylov subspace and ADI methods for solving the Lyapunov equation, SIAM Journal on Numerical Analysis 49 (2011) 1875–1898.

[19] Y. Saad, Iterative Methods for Sparse Linear Systems, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[20] E. L. Poole, J. M. Ortega, Multicolor ICCG methods for vector computers, SIAM Journal on Numerical Analysis 24 (1987) 1394–1417.

[21] D. Lukarski, Parallel sparse linear algebra for multi-core and many-core platforms - parallel solvers and preconditioners, Ph.D. thesis, Karlsruhe Institute of Technology (KIT), Germany (2012).

[22] M. Benzi, W. Joubert, M. G., Numerical experiments with parallel orderings for ILU preconditioners, Electronic Transactions on Numerical Analysis 8 (1999) 88–114.

[23] S. Doi, On parallelism and convergence of incomplete LU factorizations, Applied Numerical Mathematics 7 (5) (1991) 417–436. doi:10.1016/0168-9274(91)90011-N.

[24] K. Bergman *et al.*, Exascale computing study: Technology challenges in achieving exascale systems, DARPA IPTO ExaScale Computing Study (2008).

[25] A. Frommer, D. B. Szyld, On asynchronous iterations, Journal of Computational and Applied Mathematics 123 (2000) 201–216.

[26] E. Chow, H. Anzt, J. Dongarra, Asynchronous Iterative Algorithm for Computing Incomplete Factorizations on GPUs, in: International Supercomputing Conference (ISC 2015), Accepted, 2015.

[27] H. Anzt, Asynchronous and Multiprecision Linear Solvers - Scalable and Fault-Tolerant Numerics for Energy Efficient High Performance Computing, Ph.D. thesis, Karlsruhe Institute of Technology, Institute for Applied and Numerical Mathematics (Nov. 2012).

[28] I. S. Duff, G. A. Meurant, The effect of ordering on preconditioned conjugate gradients, BIT 29 (4) (1989) 635–657.
[29] NVIDIA Corporation, NVIDIA CUDA Compute Unified Device Architecture Programming Guide, 2nd Edition (August 2009).
[30] NVIDIA Corporation, NVIDIA CUDA TOOLKIT V6.0 (July 2013).
[31] Innovative Computing Lab, Software distribution of MAGMA version 1.6, `http://icl.cs.utk.edu/magma/` (2014).
[32] NVIDIA Corporation, CUSPARSE LIBRARY (August 2014).
[33] E. L. Wachspress, The ADI Model Problem, Springer New York, 2013. doi:10.1007/978-1-4614-5122-8.
[34] P. Benner, J. Saak, A semi-discretized heat transfer model for optimal cooling of steel profiles, in: P. Benner, V. Mehrmann, D. Sorensen (Eds.), Dimension Reduction of Large-Scale Systems, Vol. 45 of Lect. Notes Comput. Sci. Eng., Springer-Verlag, Berlin/Heidelberg, Germany, 2005, pp. 353–356.
[35] A. Logg, K.-A. Mardal, G. N. Wells, et al., Automated Solution of Differential Equations by the Finite Element Method, Springer, 2012. doi:10.1007/978-3-642-23099-8.
[36] NVIDIA Corporation, CUBLAS Library User Guide, du-06702-001_v6.5 Edition (August 2014).
[37] M. Wolf, M. Heroux, E. Boman, Factors impacting performance of multithreaded sparse triangular solve, in: J. Palma, M. Daydé, O. Marques, J. Lopes (Eds.), High Performance Computing for Computational Science – VECPAR 2010, Vol. 6449 of Lecture Notes in Comput. Sci., Springer Berlin Heidelberg, 2011, pp. 32–44. doi:10.1007/978-3-642-19328-6_6.
[38] M. Naumov, Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU, Tech. Rep. NVR-2011-001, NVIDIA (2011).
[39] F. L. Alvarado, R. Schreiber, Optimal parallel solution of sparse triangular systems, SIAM Journal on Scientific Computing 14 (1993) 446–460.
[40] A. Pothen, F. Alvarado, A fast reordering algorithm for parallel sparse triangular solution, SIAM Journal on Scientific and Statistical Computing 13 (2) (1992) 645–653. doi:10.1137/0913036.