July 28, 2019

A STRATEGY FOR PARALLEL ILU PRECONDITIONING

..

Abstract. In this paper, we propose a sparse approximate inverse of triangular matrices based on Jacobi iteration. The algorithm is easy to implement in parallel, as the main operation of the algorithm is sparse matrix-matrix multiplication. Then the two triangular solvers in the ILU preconditioning can be replaced by only two matrix-vector products, which make the iterative solvers completely parallel. We apply the new preconditioners to solving linear systems and eigenvalue problems with iterative methods.

Keywords: parallel preconditioner, iterative ILU factorization, preconditioned method, linear system, eigenvalue problem

1. Introduction. The ILU factorization is a type of general-purpose preconditioning for sparse linear systems. There are mainly two obstacles in its parallelization. The first is the parallel generation of ILU factors. In our previous paper [XX], we propose an iterative ILU factorization in matrix form, Algorithm 1. The factors can be generated by several matrix-matrix multiplications and some dropping rules, which are of fine-grained parallelism and easy to implement on various parallel computing platforms, and they have the similar preconditioning effect to the standard ILU factorization in solving many problems. The second obstacle is the solutions

Algorithm 1 Iterative ILU factorization (IterILU)

1: Set the initial data $L_0 = D = U_0 = 0$.
2: **for** $k = 1, 2, \cdots, p$ **do**
3: $B = A - L_0 U_0$
4: $D = diag(diag(B))$
5: $U_0 = triu(B, 1)$
6: $L_0 = tril(B, -1)D^{-1}$
7: execute dropping strategies on L_0 and U_0.
8: **end for**
9: Output: $L = L_0 + I$ and $U = U_0 + D$.

Notice: $diag(diag(B))$, $tril(B, -1)$ and $triu(B, 1)$ are Matlab-like notations and denote the diagonal, strictly lower triangular and strictly upper triangular part of B, respectively.

of triangular systems in the preconditioning procedure. It is easy to solve exactly a triangular system using the forward or backward substitution method. However, this is a highly sequential process, and it may be executed many times in solving a system with an iterative method, which cannot exploit the performance of a parallel computing platform sufficiently. This is the main problem we study in this paper.

The simplest parallelization strategy in solving triangular systems is to compute the summations of the substitution methods in parallel:

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{k=1}^{i-1} a_{ik}x_k\right) \quad \text{or} \quad x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{k=i+1}^{n} a_{ik}x_k\right).$$

This is a low-level concurrent method, especially for sparse triangular matrices. Its corresponding blocking method can improve the performance. The level-scheduling

method with reordering techniques improve the parallelism further [2]. However, these methods still have some restrictions in taking full use of manycores CPUs.

Another strategy is to replace the exact solution of a triangular systems by an approximate one. This ideal is based on the fact that the incomplete LU factorization is incomplete. The solutions of the incomplete factors should tolerate some errors. The (block) Jacobi iteration method is an easy way to obtain approximate solutions [7, 8], and it only need matrix-vector products and vector additions, which are of fine-grained parallelism. However, too many matrix and vector operations may increase the runtime. An alternative method is to use sparse approximate inverses (SAI), which is based on the decay of inverses of sparse matrices [11, 10, 14]. When using SAI in the ILU preconditioning, the two triangular solvers can be replaced by only two matrix-vector multiplications. There are several ways to compute the SAI of a matrix, for example Frobenius norm minimization and incomplete inverse factorization [4, 5, 6, 15]. Also, there are some SAIs specially designed for triangular systems [3, 13, 17]. In [12, 16], the truncated Neumann expansions play a similar role to the SAIs of ILU factors. Some recent processes of this topic are introduced in [1].

In this paper, we propose a Sparse Approximate Inverse for Triangular systems (SAIT) based on Jacobi iteration. The SAIT algorithm only involves some sparse matrix-matrix multiplications and dropping rules, which are very easy to implement on both shared-memory and distributed-memory platforms. Associated with Algorithm 1, the two algorithms IterILU-SAIT can be regarded as an entire preconditioner.

This paper is organized as follows. In Section 2, we describe the basic ideal of the SAIT algorithm. And we propose two dropping strategies to specify the SAIT algorithm in Section 3. In Section 4, we apply the new preconditioners to solving linear systems and a eigenvalue problems with iterative methods. In Section 5, it is the conclusion.

2. The basic idea of the SAIT algorithm. We solve the triangular system

$$Tx = b$$

by Jacobi iteration method

$$x^k = D^{-1}(D - T)x^{k-1} + D^{-1}b, \tag{2.1}$$

where D is the diagonal matrix of T. As T is triangular, the iteration solution converges to the exact solution within at most n steps.

For the sake of simplicity, we denote $\widetilde{T} = D^{-1}(D - T)$. Then, (2.1) becomes

$$x^k = \widetilde{T}x^{k-1} + D^{-1}b.$$

Letting the initial datum $x^0 = 0$, we write the solution of each iteration explicitly:

$$\begin{aligned}
x^1 &= D^{-1}b \\
x^2 &= \widetilde{T}x^1 + D^{-1}b = (\widetilde{T} + I)D^{-1}b \\
x^3 &= \widetilde{T}x^2 + D^{-1}b = (\widetilde{T}^2 + \widetilde{T} + I)D^{-1}b \\
&\cdots \\
x^k &= \widetilde{T}x^{k-1} + D^{-1}b = \sum_{i=0}^{k-1} \widetilde{T}^i D^{-1}b.
\end{aligned}$$

Denoting

$$M_k = \sum_{i=0}^{k-1} \widetilde{T}^i D^{-1}, \tag{2.2}$$

then we have

$$x^k = M_k b.$$

As $\widetilde{T}$ is a strictly triangular matrix, we have $\widetilde{T}^k = 0$ when $k \geq n$. Consistent with the convergence property of Jacobi method for triangular systems, the exact inverse of T is a finite series of (2.2), i.e.

$$T^{-1} = M_n \equiv \sum_{i=0}^{n-1} \widetilde{T}^i D^{-1}. \tag{2.3}$$

In the ILU preconditioning, the triangular factors are not the exact factorization of the original matrix. There are some errors contained in them. We denote the error of T by T_ϵ. Then the ideal solution should be

$$x_{ideal} = (T + T_\epsilon)^{-1} b.$$

Even though we solve the exact solution of the system $Tx = b$ ($x = T^{-1}b \equiv M_n b$), there is still a gap between x and x_{ideal}. Based on this fact, if we impose an appropriate perturbation M_ϵ on the exact inverse of T and get an approximate solution of x by $x_\epsilon = (T^{-1} + M_\epsilon)b \equiv (M_n + M_\epsilon)b$, the error magnitude of $\|x_\epsilon - x_{ideal}\|$ can be similar to $\|x - x_{ideal}\|$. Then x can be replaced by x_ϵ in the preconditioning procedure.

The perturbation M_ϵ can be some dropping and truncation rules. However, if we perform these rules on the matrix polynomial

$$M_k = D^{-1} + \widetilde{T} D^{-1} + \widetilde{T}^2 D^{-1} + \cdots + \widetilde{T}^k D^{-1},$$

it involves computing the high-order terms $\widetilde{T}^i$ $(i = 1, \cdots, k)$, which is consuming and possibly runs out of memory. In order to avoid such problem, we rewrite M_k in (2.2) as an equivalent form:

$$\begin{aligned} M_0 D &= I, \\ M_1 D &= \widetilde{T} + I, \\ M_2 D &= \widetilde{T}(\widetilde{T} + I) + I, \\ M_3 D &= \widetilde{T}(\widetilde{T}(\widetilde{T} + I) + I) + I, \\ &\cdots \end{aligned} \tag{2.4}$$

This is the Horner's method in calculating values of polynomial functions. The same method was also proposed by a Chinese mathematician *Qin Jiushao* in 13^{th} century

Song dynasty [18]. Furthermore, we present (2.4) in a recursive formula:

$$\begin{aligned} M_0 D &= I, \\ M_1 D &= \widetilde{T}(M_0 D) + I, \\ M_2 D &= \widetilde{T}(M_1 D) + I, \\ M_3 D &= \widetilde{T}(M_2 D) + I, \\ &\cdots \\ M_k D &= \widetilde{T}(M_{k-1} D) + I, \\ &\cdots \end{aligned} \tag{2.5}$$

Then, we can perform dropping rules after each multiplication by $\widetilde{T}$ in (2.5) and stop the iteration with an appropriate number. This strategy is summarized in Algorithm 2.

Algorithm 2 Sparse Approximate Inverse for Triangular systems base on Jacobi Iteration (SAIT)

1: Set the initial data $M = I$, let D be the diagonal matrix of T and let $\widetilde{T} = I - D^{-1}T$.
2: **for** $k = 1, 2, \cdots, m$ **do**
3: $M = \widetilde{T}M + I$
4: execute dropping rules on M
5: **end for**
6: $M = MD^{-1}$

3. Dropping strategies. Similar to the iterative ILU factorizations in [XX], we use two basic dropping strategies, threshold and position, to concretize Algorithm 2.

Threshold. In step 3 of Algorithm 2, the diagonal entries of M are all units. We drop the entries whose magnitudes are small than τ ($\tau < 1$) in M. By this setting, we can avoid dropping the diagonal entries, such that M can be alway full rank during the iterations. This strategy is summarized in Algorithm 3. We implement this algorithm in Matlab, pasted in Appendix A.

Algorithm 3 Threshold-based Sparse Approximate Inverse of Triangular systems SAIT_Thr(τ, m)

1: Set the initial data $M = I$, let D be the diagonal matrix of T and let $\widetilde{T} = I - D^{-1}T$.
2: **for** $k = 1, 2, \cdots, m$ **do**
3: $M = \widetilde{T}M + I$
4: drop the entries in M whose magnitudes are small than τ ($\tau < 1$)
5: **end for**
6: $M = MD^{-1}$

Sparsity pattern. We set a fixed sparsity pattern $\boldsymbol{S}$ in advance, and drop the entries out of $\boldsymbol{S}$ during the computations. The pattern of the power function of the original matrix is often used in computing a sparse approximate inverse. Letting T_0

be the strictly triangular part of T, then

$$T^p = (D + T_0)^p = \sum_{i=0}^{p} C_p^i D^{p-i} T_0^i.$$

Since T_0 shares the same pattern with $\widetilde{T}$, M_{p+1} has the same pattern with T^p. As M_p tends to the exact inverse with $p \to n$, this pattern can be regarded as a truncated pattern of the exact inverse. When $p = 1$, it is the Jacobi preconditioner, while when $p = 2$, it is pattern with the original matrix T. The M_p can be obtained by Algorithm 2 without dropping any entries. Then, in the following iterations, we drop the entries out of the pattern of M_p. The number p should not be too large, as the number of nonzeros in the pattern increases rapidly with p increasing. This dropping strategy is summarized in Algorithm 4.

Algorithm 4 Pattern-based Sparse Approximate Inverse of Triangular systems SAIT_Pat(p, m)

1: Set the initial data $M = I$, let D be the diagonal matrix of T and let $\widetilde{T} = I - D^{-1}T$.
2: **for** $k = 1, 2, \cdots, p$ **do**
3: $M = \widetilde{T}M + I$
4: **end for**
5: get the sparsity pattern $\boldsymbol{S}$ of M
6: **for** $k = 1, 2, \cdots, m$ **do**
7: $M = \widetilde{T}M + I$
8: drop the entries in M out of the pattern $\boldsymbol{S}$
9: **end for**
10: $M = MD^{-1}$

3.1. Balance between nonzeros and iteration counts. In the Algorithm 3 and 4, if we use smaller threshold τ and larger p, we can obtain more accurate approximate inverses, as we keep more nonzeros in the SAIT matrices. However, more accurate SAIT matrices do not mean shorter runtime of a iterative solver with them as the preconditioner.

We list the following terms related to an iterative solver with the SAIT preconditioners:

- $iter(M_L, M_U)$: M_L and M_U are the SAIT matrices of a ILU factors L and U, respectively. This term is the iteration count of a iterative solver with the pair of matrices in the preconditioning procedure.
- $comp(M_L, M_U)$: the total time of computing matrix-vector multiplications with M_L and M_U.
- *othercomp*: the time of computing other terms in each iteration except the preconditioning procedure.

Then, the runtime of a preconditioned iterative method can be presented roughly as the following formula:

$$runtime = iter(M_L, M_U) \times \Big(comp(M_L, M_U) + othercomp \Big). \tag{3.1}$$

For a certain iterative method and a certain computer environment, the tern *othercomp* is usually fixed. If we use more accurate SAIT matrices, the iteration count $iter(M_L, M_U)$

decrease. However, we can not expect that the SAIT preconditioners can reduce the iteration count less than the exact solver. Possibly, after exceeding some point, the $iter(M_L, M_U)$ does not continuously decrease with more nonzeros in M_L and M_U, which causes the term $comp(M_L, M_U)$ increasing, sequentially, the eventual *runtime* increasing. In order to reduce the runtime, it should make a balance between the accuracy of the SAIT matrix and the iteration count. As the accuracy is mainly decided by the threshold τ and the number p in the Algorithm 3 and 4, respectively, there should be some τ or p that makes the runtime of an iterative solver with SAIT shortest. We call such parameters *Optimal*. The optimal parameters may vary with different problems and different computer environments.

In the preconditioning procedure, the preconditioned solution is $M_U(M_L y)$ for a vector y. This involves two matrix-vector multiplications. We can compute the multiplication of M_U and M_L in advance, and get a matrix M_A, which can be regarded as an approximate inverse of A:

$$M_A = M_U M_L \approx U^{-1} L^{-1} \approx A^{-1}.$$

Then, $M_U(M_L y)$ can be replaced by only one multiplication $M_A y$. This can be a choice. However, for most cases, the number of nonzeros in M_A is larger the sum of nonzeros in M_U and M_L, and the runtime of $M_A y$ is probably longer than that of $M_U(M_L y)$.

4. Numerical experiments. Algorithm 1 and 2 can be regarded as an entire preconditioner — *IterILU-SAIT*. The dropping strategies in the two algorithms remain to be specified. In [XX], the numerical experiments illustrate that the pattern-based IterILU has the similar preconditioning effect to the standard ILU factors. In this section, we take the level-0 and level-1 IterILU factors as the original triangular systems and use the SAIT algorithm to approach their inverses. And then we apply the SAIT matrices in in preconditioned iterative methods for both linear systems and eigenvalue problems. As the promise of the SAIT algorithm is that the IterILU works, most the numerical examples we use here are the same with those in [XX]. The code is implemented in Matlab and run on a laptop with an Intel i7-6700HQ CPU with 16 GB RAM and GTX970M GPU with 3 GB memory. We set the stopping criteria to 10^{-10} for all the numerical experiments uniformly.

The main test model is a square matrix with 10^6 rows/columns from the finite difference discretization of three-dimensional Laplace equation with homogeneous Dirichlet boundary condition on a $102 \times 102 \times 102$ mesh. In fact, the ILU preconditioning may not be a very good choice for this mode problem. Generally, there are two situations that computations can benefit from preconditioning. One is that original solvers for some systems are not stable and even do not converge, and appropriate preconditioners can make them robust. The other is that the preconditioning can reduce the computing time essentially. This model is a quite well-posed problem, and many solvers are stable in solving it. Although the ILU preconditioning can reduce the iteration count, the preconditioning procedure involves solving two triangular systems $L(Uy) = b$ in each iteration, which may compensate the runtime The reduction of total computing time is not obvious, eventually, shown in Table 4.1. Usually, if one wants to observe an obvious time reduction, the iteration count should be reduced at least two thirds by the ILU preconditioners. Furthermore, the main operations of the primal conjugate gradient method are vector-inner products and matrix-vector products, which are of high parallelism. In the ILU preconditioned methods, it needs extra efforts to accomplish parallelism for solving the triangular systems.

	PCG iter	time (s)	improve
no p.c.	423	10.39	-
level-0 ILU	144	9.85	5.29%
level-1 ILU	97	8.31	20.02%

Table 4.1: The comparison of the iteration counts and runtime of solving the 3D FDM system with 10^6 row using PCG method without preconditioner and with level-0 and level-1 ILU preconditioners.

The reason that we still take use of this model problem is that it can act as a comparison of the exact solver for triangular systems and the SAIT algorithms in PCG method. Later, we will provide some examples that can obtain enough benefit from the ILU-SAIT preconditioning.

4.1. SAIT_Thr(τ, m) and SAIT_Pat(p, m). We approximate the triangular matrices of level-0 and level-1 ILU factorizations by the SAIT_Thr(τ, m) in Algorithm 3 and SAIT_Pat(p, m) in Algorithm 4 with different parameters, respectively. We test and compare the sizes and the preconditioning effects of these SAIT preconditioners.

To show its storage, we use the ratio of nonzeros in a SAIT matrix compared with its corresponding original matrix:

$$ratio\ /\ r = \frac{nnz(M_T)}{nnz(T)},$$

Figure 4.1 shows the tendencies of the ratios of nonzeros in SAIT matrices generated by SAIT_Thr(τ, m) with different threshold $\tau = 0,\ 0.05,\ 0.02,\ 0.01$ and increasing iteration number m. The setting $\tau = 0$ means that there is no entry dropped in each iteration, and its numbers of nonzeros under this situation increase rapidly. For other nonzero τ, the numbers of nonzeros reach different stable states after several iterations, the smaller τ the more nonzeros in its corresponding SAIT matrix. For SAIT_Pat(p, m), a fixed p leads to a fixed pattern, i.e. the number of nonzeros of its SAIT matrix is fixed. We put their ratios of the SAIT matrices of this algorithm in the legends in Figure 4.3.

Next, we test effects of SAIT matrices as preconditioners in PCG method. For the matrices generated by SAIT_Thr(τ, m), the iteration counts of PCG decrease with increasing m until reaching different stable states, shown in Figure 4.2. For the SAIT_Pat(p, m), the iteration counts keep stable with m varying, shown in Figure 4.3. For the both algorithms, the small threshold τ and larger p (more nonzeros) means less iteration count. However, there is no case that the iteration count of the SAIT preconditioner is less than that of the exact triangular solver.

Finially, we compare the two algorithms. From tendencies in Figure 4.4, the PCG iteration count of SAIT_Thr(τ, m) is less than that of SAIT_Pat(p, m) with the same nonzeros in the SAIT matrices for the both level ILU factorizations.

4.2. Runtime and Jacobi iteration. Since the SAIT algorithms are designed based on Jacobi iteration method, if we use the Jacobi method in the preconditioning procedure directly, it is equivalent to the SAIT matrices generated by SAIT_Thr$(0, m)$, which are more accurate than the cases with thresholds $\tau > 0$. In addition, it requires less memory, since a SAIT matrix usually has more nonzeros than its original matrix. We list the iteration counts and runtime of the SAIT algorithms in Table 4.2 and the

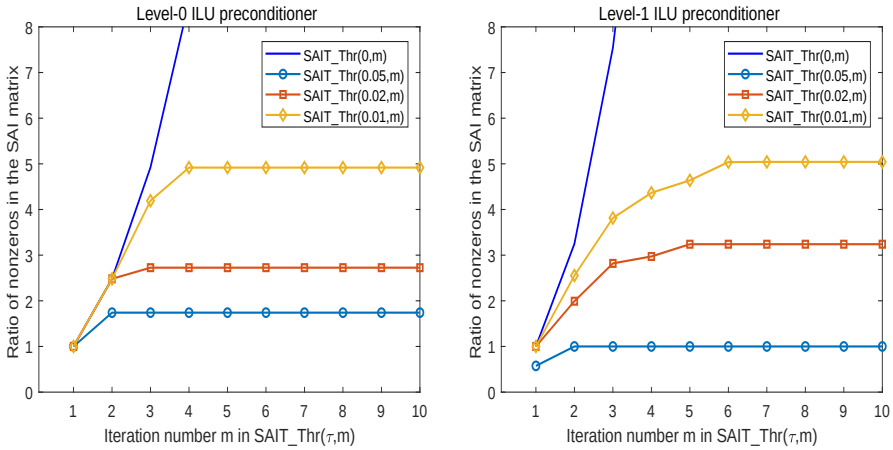

Fig. 4.1: The ratios of the nonzeros of the SAIT matrix generated by SAIT_Thr(τ, m) with different τ and m for level-0 (Left) and level-1 (Right) ILU factors.

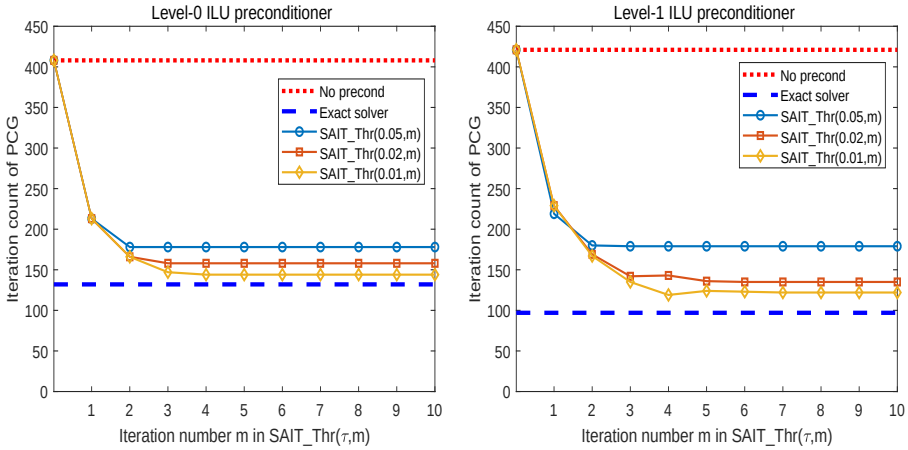

Fig. 4.2: The iteration counts of PCG with SAIT preconditioners generated by SAIT_Thr(τ, m) with different τ and m for level-0 (left) and level-1 (right) ILU factors.

Jacobi method in Table 4.3, respectively, in the same computer environment. Compared with the result in Table 4.1, the PCG iteration counts of Jacobi method can be almost the same as the exact solver after several iteration. However, for most cases, the runtime of SAIT preconditioners is much shorter than that of Jacobi method. The reason is that Jacobi method involves several matrix-vector multiplications and vector-vector additions, while there is only one matrix-vector multiplication when using the SAIT preconditioners, although its matrices have more nonzeros than the matrix of Jacobi method.

From the Table 4.2, we observe that less iteration count does not means shorter runtime. We try more threshold parameters using SAIT_Thr$(0, m)$, shown in Figure

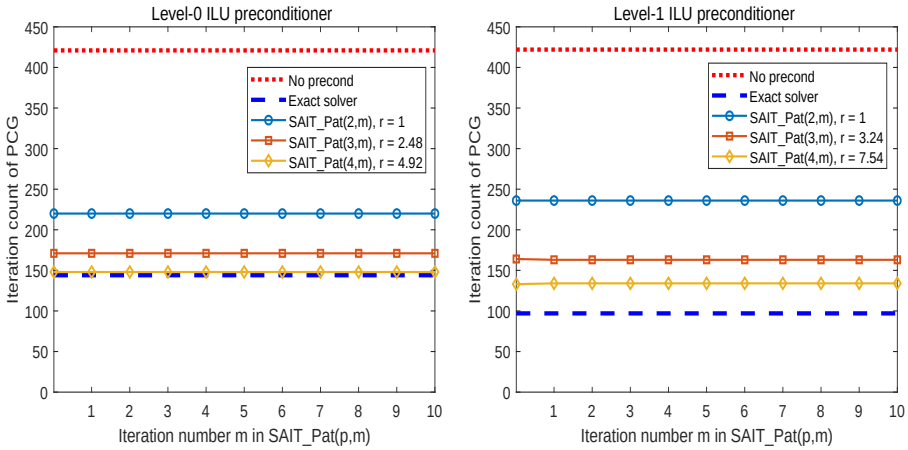

Fig. 4.3: The iteration counts of PCG with SAIT preconditioners generated by SAIT_Pat(p, m) with different p and m for level-0 (left picture) and level-1 (right picture) ILU factors.

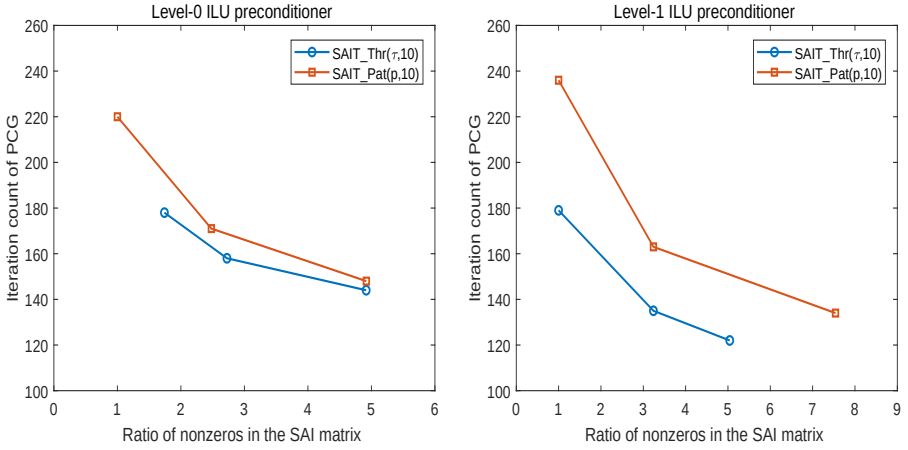

Fig. 4.4: The comparisons of the iteration counts of PCG with SAIT preconditioner generated by SAIT_Thr(τ, m) and SAIT_Pat(p, m) for level-0 (left picture) and level-1 (right picture) ILU factors.

4.5. The iteration count does not decrease continuously with smaller threshold τ or allowing more nonzeros after some point. Even though we use the exact inverses which are usually full triangular matrices, the iteration count is equal to that using an exact triangular solver, which is not an arbitrarily small number. However, a SAIT matrix with too much nonzeros can result in expensive cost of matrix-vector multiplication in the preconditioning procedure of each PCG iteration. From the relation between the runtime and numbers of nonzeros, the right picture of Figure 4.5, there should exists an optimal number of nonzeros or optimal threshold τ in respect of the runtime. There is no existing theory to predict such optimal parameter. It can be found by

	level-0 ILU			level-1 ILU		
	ratio	iter	time (s)	ratio	iter	time (s)
SAIT_Thr(0.05,10)	1.74	189	***9.95***	1.00	184	***9.97***
SAIT_Thr(0.02,10)	2.73	168	10.52	3.37	133	12.73
SAIT_Thr(0.01,10)	4.92	***154***	13.24	5.17	***121***	15.48
SAIT_Pat(1,10)	1.00	228	10.83	1.00	229	12.22
SAIT_Pat(2,10)	2.48	177	10.95	3.25	158	15.37
SAIT_Pat(3,10)	4.92	***154***	13.25	7.54	129	22.09

Table 4.2: The ratios the nonzeros, iteration count and runtime of solveing the 3D FDM system using PCG method with different SAIT preconditoners. The red italic numbers are the shortest runtime of SAIT preconditoners generated by level-0 and level-1 ILU factors, respectively.

Jacobi iter	level-0 ILU		level-1 ILU	
	iter	time (s)	iter	time (s)
1	423	29.88	423	31.87
2	229	21.03	240	24.78
3	173	***19.33***	169	22.10
4	152	20.43	134	***21.19***
5	152	23.23	116	21.48
6	149	25.94	108	23.45
7	147	28.19	104	24.74
8	146	30.76	101	27.21
9	145	33.67	97	28.38
10	145	36.60	98	31.31
11	145	39.53	98	33.90
12	145	42.28	98	36.50
13	145	45.66	98	39.25
14	145	48.44	97	41.43
15	145	51.54	97	47.38

Table 4.3: The iteration counts and runtime of solving the 3D FDM system using PCG method with Jacobi method with different iteration number in the preconditioning procedure. The red italic numbers are the shortest runtime of level-0 and level-1 ILU factors, respectively.

numerical experiments for each specific problem.

We apply the SAIT algorithms to more examples, shown in Table 4.4. Table 4.5 are the results using the SAIT_Thr(τ, m) preconditioners. Here, the ratios of nonzeros in the SAIT matrices generated by their corresponding thresholds (may be not optimal) are mainly in the range of 1 to 2. We also try SAIT_Pat(p, m) for these examples, shown in Table 4.6. Compared with the results of the two dropping strategies, the threshold-based SAIT_Thr(τ, m) preconditioners preform much better than the pattern-based SAIT_Pat(p, m) for these problems. In Table 4.5, we compare the iteration counts of SAIT preconditioners and exact solvers. Even though more iterations, the runtime of SAIT can be reduced by multicore computers easily.

	row/column	nnz
apache1	80800	542184
apache2	715176	4817870
thermal1	82654	574458
thermal2	1228045	8580313
parabolic_fem	525825	3674625
G3_circuit	1585478	7660826
ecology2	999999	4995991
thermomech_dM	204316	1423116
thermomech_TC	102158	711558
offshore	259789	4242673

Table 4.4: The numbers of rows and nonzeros of some SPD matrices from [9].

	no p.c.	level-0 ILU					level-1 ILU				
		exact	τ	ratio	SAIT		exact	τ	ratio	SAIT	
apache1	3777	365	0.05	1.46	439	20.3%	249	0.03	1.80	316	26.9%
apache2	5528	882	0.05	1.39	1092	23.8%	587	0.03	1.70	797	35.8%
thermal1	1707	651	0.05	1.43	703	8.0%	363	0.04	1.76	435	19.8%
thermal2	6626	2555	0.05	1.42	2763	8.1%	1401	0.04	1.72	1674	19.5%
parabolic_fem	3515	1423	0.1	0.96	1640	15.2%	845	0.04	1.32	946	12.0%
thermomech_dM	89	10	0.06	1.01	11	10.0%	6	0.02	1.02	8	33.3%
thermomech_TC	89	10	0.06	1.01	11	10.0%	6	0.02	1.02	8	33.3%
ecology2	7127	2123	0.08	2.00	2830	33.3%	1303	0.06	3.25	1942	49.0%
G3_circuit	21205	1182	0.08	2.07	1582	33.8%	643	0.08	2.38	1174	82.6%
offshore	-	574	0.05	1.07	690	20.2%	-	-	-	-	-

Table 4.5: The iteration counts of PCG method with the exact solver and SAIT matrices generated by SAIT_Thr(τ, m) in the preconditioning procedure. The level-1 ILU does not work for the *offshore* problem.

	level-0 ILU				level-1 ILU			
	m = 2		m = 3		m = 2		m = 3	
	ratio	SAIT	ratio	SAIT	ratio	SAIT	ratio	SAIT
thermomech_dM	1	11	1.60	10	1	10	2.37	6
thermomech_TC	1	11	1.60	10	1	10	2.37	6
thermal1	1	847	1.85	690	1	743	2.49	440
thermal2	1	3284	1.85	2674	1	2868	2.48	1635
parabolic_fem	1	1678	1.56	1451	1	1444	2.38	893
apache1	1	3252	2.41	2236	1	4102	3.11	2570
apache2	1	2753	2.42	1818	1	3521	3.12	2500
G3_circuit	1	3993	2.04	2751	1	3713	2.37	2245
ecology2	1	3738	2.00	2799	1	3765	2.25	2549
offshore	1	1389	3.96	726	-	-	-	-

Table 4.6: The iteration counts of PCG method with the SAIT preconditioners generated by SAIT_Pat(p, m).

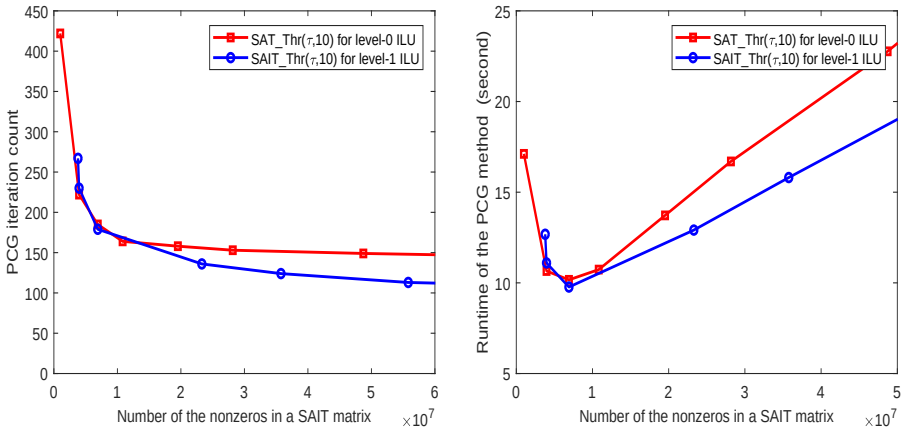

Fig. 4.5: Left: the relation between the PCG iteration counts and the numbers of nonzeros in the SAIT preconditioners. Right:the relation between the runtime of PCG method and the numbers of nonzeros in the SAIT preconditioners.

	Ax	$x-y$	(x,y)	$norm(x)$
CPU	1.62e-2 (875.7)	1.41e-3 (76.2)	6.08e-4 (32.9)	2.65e-4 (14.3)
GPU	6.36e-5 (3.4)	1.85e-5 (1)	1.58e-4 (8.5)	1.31e-4 (7.1)
speed-up	254.7	76.2	3.9	2.0

Table 4.7: The runtime of several matrix and vector operations on CPU and GPU. The numbers in the brackets are the scalarized time, where the shortest time in this table is taken as the unit one.

4.3. Solving with GPU. The main computations of the primal conjecture gradient method are a matrix-vector multiplication and several vector-inner products. With the IterILU-SAIT preconditioners, the preconditioning procedure become two matrix-vector multiplications. All of these operations can be highly parallelized. The preconditioner solver can be moved to GPU almost without any modification. We compare the performances of these operations on the CPU and GPU of this laptop, shown in Table 4.7. The runtime is reduced a lot by the GPU, especially the matrix-vector multiplication. We run the solver with the same preconditioners in Table 4.2 on the GPU. Though the iteration counts are the same with running on CPU, the codes are accelerated several times by CPU, shown in Table 4.8.

4.4. Preconditioned solver for eigenvalue problems. When using the simultaneous preconditioned methods to compute the eigenvalues of a system, there are many vectors to deal with in the preconditioning procedure of each iteration. With IterILU-SAIT preconditioners, this procedure can be done through two sparse matrix-matrix multiplications. Here, we use the LOBPCG method to compute the first 4 eigenvalues of the three-dimensional FDM matrix with 10^6 rows/columns in the previous part of this section. In order to improve the convergence efficiency, we enlarge the subspace in the LOBPCG method by computing one more eigenvalue. When the errors of the first 4 eigenvalues reach the stopping criteria 10^{-10}, we stop

	level-0 ILU			level-1 ILU		
	ratio	iter	time (s)	ratio	iter	time (s)
SAIT_Thr(0.05,10)	1.74	189	1.34	1.00	184	1.34
SAIT_Thr(0.02,10)	2.73	168	1.33	3.37	133	1.59
SAIT_Thr(0.01,10)	4.92	154	1.81	5.17	121	1.81
SAIT_Pat(1,10)	1.00	228	1.47	1.00	229	1.67
SAIT_Pat(2,10)	2.48	177	1.36	3.25	158	1.92
SAIT_Pat(3,10)	4.92	154	1.80	7.54	129	2.38

Table 4.8: The same codes in Table 4.2 are run on GPU.

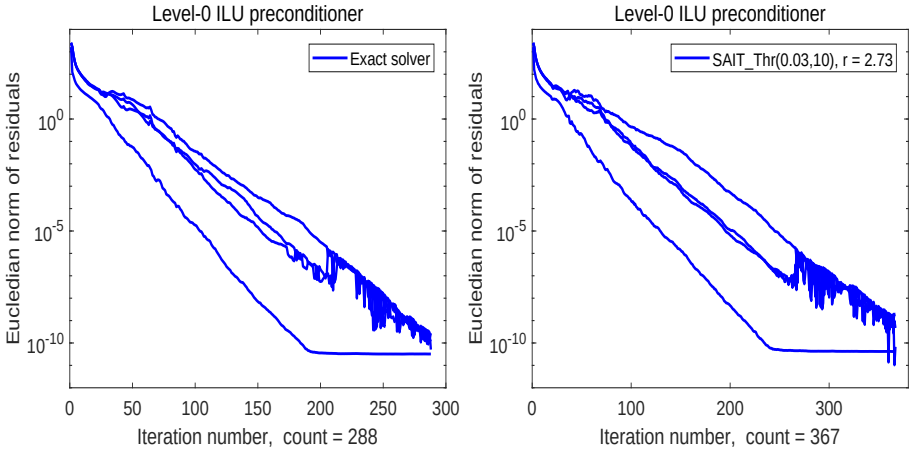

Fig. 4.6: The convergence history in solving Laplace eigenvalue problem using LOPBCG method with the exact solver and a SAT preconditioner generate by SAIT_Thr$(0.03, 10)$ for level-0 ILU factors in the preconditioning procedure.

iterating. We use level-0 and level-1 ILU factorizations, and set the threshold $\tau = 0.03$ in SAIT_Thr(τ, m) to compute their SAIT matrices. Table 4.6 and 4.6 show their results, respectively.

5. Conclusion. We derive a sparse approximate inverse for triangular systems based in Jacobi iteration and specify the algorithm with two dropping strategies, threshold and pattern. Associated with the iterative ILU factorization in [XX], we propose the *IterILU-SAIT* preconditioner. Both the generation and application of this preconditioner are of fine-grained parallelism and easy to be implemented in various parallel platforms. We present some numerical examples to show the effects of IterILU-SAIT preconditioners. Though the iteration counts of IterILU-SAIT are larger than exact triangular solvers, its runtime is easy to be reduced by parallel running.

Appendix A. The Matlab code of SAIT_Thr(τ, m).

```
function M = SAIT_Thr(T,tau,m)
```

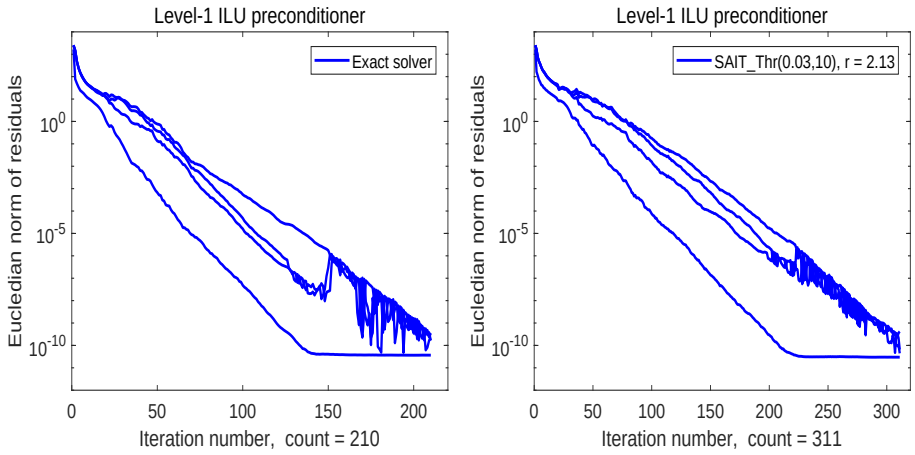

Fig. 4.7: The convergence history in solving Laplace eigenvalue problem using LOPBCG method with the exact solver and a SAT preconditioner generated by SAIT_Thr$(0.03, 10)$ for level-1 ILU factors in the preconditioning procedure.

```
 I = speye(size(T,1));
Dn = diag(diag(T).^(-1));
T0 = I - Dn*T;
 M = I;

for i = 1 : m
    M = T0*M + I;
    P = (abs(M)>tau);
    M = M.*P;
end

M = M*Dn;

end
```

REFERENCES

[1] Ahmad Abdelfattah, Hartwig Anzt, Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, Ichitaro Yamazaki, and Asim YarKhan. Linear algebra software for large-scale accelerated multicore computing. *Acta Numerica*, 25:1–160, 2016.

[2] Edward Anderson and Youcef Saad. Solving sparse triangular linear systems on parallel computers. *Internat. J. High Speed Comput.*, 1(01):73–95, 1989.

[3] Hartwig Anzt, Thomas K Huckle, Jürgen Bräckle, and Jack Dongarra. Incomplete sparse approximate inverses for parallel preconditioning. *Parallel Comput.*, 71:1–22, 2018.

[4] Michele Benzi. Preconditioning techniques for large linear systems: a survey. *J. Comput. Phys.*, 182(2):418–477, 2002.

[5] Michele Benzi, Carl D Meyer, and Miroslav Tůma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM J. Sci. Comput.*, 17(5):1135–1149, 1996.

[6] Michele Benzi and Miroslav Tuma. A comparative study of sparse approximate inverse preconditioners. *Appl. Numer. Math.*, 30(2-3):305–340, 1999.

[7] Edmond Chow, Hartwig Anzt, Jennifer Scott, and Jack Dongarra. Using Jacobi iterations and

blocking for solving sparse triangular systems in incomplete factorization preconditioning. *J. Parallel Distrib. Comput.*, 119:219–230, 2018.

[8] Edmond Chow and Aftab Patel. Fine-grained parallel incomplete LU factorization. *SIAM J. Sci. Comput.*, 37(2):C169–C193, 2015.

[9] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM T. Math. Software (TOMS)*, 38(1):1, 2011.

[10] Stephen Demko, William F Moss, and Philip W Smith. Decay rates for inverses of band matrices. *Math. Comp.*, 43(168):491–499, 1984.

[11] Victor Eijkhout and Ben Polman. Decay rates of inverses of banded M-matrices that are near to toeplitz matrices. *Linear Algebra Appl.*, 109:247–277, 1988.

[12] Ivar Gustafsson and Gunhild Lindskog. Completely parallelizable preconditioning methods. *Numerical linear algebra with applications*, 2(5):447–465, 1995.

[13] Carlo Janna, Massimilano Ferronato, and Giuseppe Gambolati. A block FSAI-ILU parallel preconditioner for symmetric positive definite linear systems. *SIAM J. Sci. Comput.*, 32(5):2468–2484, 2010.

[14] Reinhard Nabben. Decay rates of the inverse of nonsymmetric tridiagonal and band matrices. *SIAM J. Matrix Anal. Appl.*, 20(3):820–837, 1999.

[15] Yousef Saad. *Iterative methods for sparse linear systems*, volume 82. SIAM, 2003.

[16] Henk A van Der Vorst. A vectorizable variant of some ICCG methods. *SIAM J. Sci. Statist. Comput.*, 3(3):350–356, 1982.

[17] Arno CN Van Duin. Scalable parallel preconditioning with the sparse approximate inverse of triangular matrices. *SIAM J. Matrix Anal. Appl.*, 20(4):987–1006, 1999.

[18] Wen-tsun Wu. *Grand Series of Chinese Mathematics (in Chinese)*, volume V. Beijing Normal University Publishing House, 2000.