J. Parallel Distrib. Comput. 119 (2018) 219–230

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Using Jacobi iterations and blocking for solving sparse triangular systems in incomplete factorization preconditioning

Edmond Chow [a,*], Hartwig Anzt [b,c], Jennifer Scott [d], Jack Dongarra [c,e,f]

[a] School of Computational Science and Engineering, Georgia Institute of Technology, USA
[b] Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Germany
[c] Innovative Computing Laboratory, University of Tennessee, USA
[d] Scientific Computing Department, STFC Rutherford Appleton Laboratory, UK
[e] School of Mathematics and School of Computer Science, University of Manchester, UK
[f] Oak Ridge National Laboratory, USA

HIGHLIGHTS

- Jacobi solvers for sparse incomplete factor preconditioning are extensively tested.
- Such solvers are generally more effective than sparse triangular solvers on GPUs.
- Blocking improves the robustness of Jacobi solvers for incomplete factors.
- The blocking should be chosen depending on the structure of the matrix.

ARTICLE INFO

Article history:
Received 4 May 2017
Received in revised form 13 February 2018
Accepted 27 April 2018
Available online 8 May 2018

Keywords:
Sparse linear systems
Triangular solves
Iterative solvers
Preconditioning

ABSTRACT

When using incomplete factorization preconditioners with an iterative method to solve large sparse linear systems, each application of the preconditioner involves solving two sparse triangular systems. These triangular systems are challenging to solve efficiently on computers with high levels of concurrency. On such computers, it has recently been proposed to use Jacobi iterations, which are highly parallel, to approximately solve the triangular systems from incomplete factorizations. The effectiveness of this approach, however, is problem-dependent: the Jacobi iterations may not always converge quickly enough for all problems. Thus, as a necessary and important step to evaluate this approach, we experimentally test the approach on a large number of realistic symmetric positive definite problems. We also show that by using *block* Jacobi iterations, we can extend the range of problems for which such an approach can be effective. For block Jacobi iterations, it is essential for the blocking to be cognizant of the matrix structure.

1. Introduction

As computers increasingly rely on various forms of parallelism to obtain high performance, the design and performance optimization of large-scale numerical algorithms must focus on the efficient exploitation of parallel architectures. Unfortunately, simple algorithms frequently become remarkably complex when implemented in parallel and the most efficient parallel algorithm may not be the one that is most intuitive.

In this paper, we investigate a non-intuitive approach to solving the sparse triangular systems that arise when using incomplete factorization preconditioners with an iterative method for solving sparse linear systems. The conventional method of solving triangular systems is to use forward or backward substitution and this can be parallelized using level scheduling [2,21,26,33,38]. Here we investigate using an iterative method—in particular, Jacobi relaxation. The use of an iterative method is feasible when an approximate solution is acceptable, as in the case of preconditioning. The approach is less applicable to the case of solving with factors from a direct (not incomplete) factorization where the factors are much less sparse and typically an exact solve is desired.

In recent previous work, this approach demonstrated significant reductions in total solution time for the preconditioned conjugate gradient (PCG) method on highly parallel architectures such as Intel Xeon Phi co-processors and graphics processing units (GPUs) [3,10], even though additional PCG iterations may be required to achieve the requested accuracy. The improved speed for each triangular solve is because, for some problems, particularly

* Corresponding author.
E-mail addresses: echow@cc.gatech.edu (E. Chow), hanzt@icl.utk.edu (H. Anzt), jennifer.scott@stfc.ac.uk (J. Scott), dongarra@icl.utk.edu (J. Dongarra).

https://doi.org/10.1016/j.jpdc.2018.04.017
0743-7315/

with high levels of fill in the triangular factors, level scheduling is unable to reveal sufficient parallelism to fully exploit the GPU hardware. On the other hand, Jacobi relaxation, which primarily relies on the sparse matrix vector product (SpMV) operation to compute a residual vector, is highly parallel and can exploit the substantial efforts that have been invested in optimizing SpMV on various parallel architectures.

Although the iterative triangular solve approach can result in significant speedups on highly parallel architectures, the approach does not work on all problems. It is possible for the iterations on the triangular systems to converge too slowly and thus be uncompetitive with the conventional level-scheduled approach. The goal of this paper is to investigate the range of applicability of using iterative triangular solves for incomplete factorization preconditioning by testing it with a large set of sparse symmetric positive definite linear systems. Further, we introduce the idea of using block Jacobi iterations,[1] which improves the robustness of the iterative approach. Our hypothesis is that for matrices that model many types of physical systems, especially partial differential equations, an iterative approach to solving the systems involving its triangular factors can be effective. In particular, although these matrices may not be diagonally dominant and may be ill-conditioned, many have *relatively* large diagonal entries compared to the off-diagonal entries. If the incomplete factorizations of these matrices are stable, they are also likely to have relatively large diagonal entries. Systems with such matrices can typically be solved efficiently by iterative methods. If convergence is poor, a block diagonal scaling can improve diagonal dominance.

Alternatives to sparse triangular solves for incomplete factorization preconditioning have been proposed before. One major alternative is to compute and use sparse approximate inverses of the incomplete factors, so that preconditioning reduces to SpMV operations [9,37]. Related to this is representing the inverse of a sparse triangular matrix as the product of sparse triangular factors [1,30]. Another possible approach is to use Neumann series approximations to the inverse of the incomplete factors [9,36]. Preconditioning is again reduced to a sequence of SpMV operations. This approach was found to be potentially competitive with other techniques for nonsymmetric problems but it lacks robustness, while for symmetric problems a number of more efficient alternatives are available [9]. The Neumann series technique is the same as the Jacobi relaxation approach investigated here if a diagonal scaling is first applied to the system in the former. Recently, based on the encouraging results in [10] for using Jacobi iterations to solve triangular systems, Huckle and coauthors [5] have suggested the use of stationary iterations for solving sparse triangular systems based on sparse approximate inverses.

In the symmetric case, the amount of parallelism in level-scheduled sparse triangular solves can be increased by using multicolor reordering of the rows and columns of the original matrix and computing the incomplete Cholesky factors of this reordered matrix [24]. However, multicolor reorderings generally give poorer PCG convergence results compared to other orderings such as the band reducing RCM ordering [12] or profile reducing Sloan ordering [34] (see, for example, [8,14,15,17,18,29]). For some "easy" problems, the convergence rate may be degraded by as much as 60–100% but this can be compensated for by the additional parallelism in the solves. For harder problems, however, multicolor reorderings may result in no convergence.

The rest of this paper is organized as follows. In Section 2, we describe the use of Jacobi and block Jacobi relaxation for the iterative solution of triangular systems. In particular, various blocking strategies are reviewed. Section 3 presents our experimental study and demonstrates the potential effectiveness of Jacobi and block Jacobi solves using a large set of test problems. Concluding remarks are made in Section 4.

[1] Initial ideas were presented orally at the 2015 SIAM Conference on Computational Science and Engineering.

2. Background and methodology

2.1. Jacobi and block Jacobi relaxation for triangular systems

Consider a triangular system of equations

$$Ry = c, \tag{1}$$

where R is either upper or lower triangular. In our application in which we want to solve the system $Ax = b$, R is an incomplete factor of A and, in particular, in the symmetric case, $R = L$ or L^T, where L is an incomplete Cholesky factor of A. The Jacobi iteration for solving (1) is

$$y_{k+1} = y_k + D^{-1}(c - Ry_k), \tag{2}$$

where D is the diagonal of R. For the sparse triangular solves, we take the initial guess

$$y_0 = D^{-1}c. \tag{3}$$

We will refer to the number of Jacobi iterations performed as the number of *Jacobi sweeps*. The iteration matrix

$$G = I - D^{-1}R \tag{4}$$

is strictly triangular with a zero diagonal. Thus the iteration is guaranteed to converge. However, in practice, because G is triangular and thus non-normal, the iteration may diverge before converging. If this initial divergence stage is long and/or causes numerical overflow, then using Jacobi iterations will not be an effective method of solution. Thus effectiveness will depend on the degree of non-normality of G. Non-normality can be measured in different ways. For triangular matrices, having only small off-diagonal entries results in a small departure from non-normality.

Block Jacobi relaxation or, equivalently, block scaling of R, can reduce non-normality. If a block structure is imposed on R then it can be trivially written as the sum

$$R = \tilde{R} + D, \tag{5}$$

where $\tilde{R}$ is strictly block triangular and D is now block diagonal with ith block D_{ii}. The iteration is the same as in (2), where the ith block row of the iteration matrix (4) now has off-diagonal blocks $G_{ij} = D_{ii}^{-1}R_{ij}$ $(j \neq i)$. If the norm of D_{ii}^{-1} is small, then the norm of the off-diagonal blocks will be small, as desired. As a proxy, we will seek to maximize the size of the entries in the diagonal blocks of A. This blocking for A is then imposed on the lower and upper triangular incomplete factors.

Observe that, in general, we seek a blocking in which the block size is small compared to the order n of A so as to limit the cost of applying block Jacobi. With small blocks, the cost of block Jacobi is not significantly greater than for scalar Jacobi because the main cost in both is the computation of the residual $c - Ry_k$. For small blocks, one possibility to assist with efficiency is to store the inverses D_{ii}^{-1} explicitly. We also note that the matrix $D^{-1}R$ is not formed, as this may require more storage than storing D^{-1} and R separately.

2.2. Blocking vs. reordering

The most general form of blocking involves two steps: (1) identifying rows (or columns) of a matrix that should be grouped together into a "block", and then (2) reordering the matrix such that the rows within each block are numbered consecutively. (We use the term "block" to mean either a *group* of rows or columns

(or their indices), or a *submatrix* of a matrix; the meaning should be clear from the context.) For symmetric matrices, the rows and columns are reordered the same way. It is known, however, that reordering a matrix affects the quality of its incomplete factorization, and that some orderings, such as RCM, are better than others when these factorizations are used as preconditioners. Arbitrary blockings (and thus reorderings) of a matrix can lead to a failure of the incomplete factorization [11].

Blocking, however, may not require the first reordering step, and these cases should be distinguished from the general case. For a matrix that already has a desired ordering, it is possible to group consecutively numbered rows (and columns) together. The simplest way to impose such a blocking is to divide the rows into the requested number of blocks such that each block contains (approximately) the same number of rows. A locality-preserving band or profile reducing ordering, such as RCM or Sloan, permutes nonzero entries of the matrix close to the diagonal of the matrix, and thus these entries are likely to lie in the block diagonal part of the reordered matrix.

In this paper, we use a simple blocking technique that we call *supervariable amalgamation*, which is described in the next subsection. For completeness, we also briefly mention other blocking techniques.

2.3. Supervariable blocking and supervariable amalgamation

The set of variables that correspond to a set of matrix columns with the same sparsity pattern is called a *supervariable*. Supervariables occur frequently as a result of each node or element of a partial differential equation (PDE) discretization having multiple variables or degrees of freedom associated with it. These variables are typically tightly coupled, leading to dense diagonal blocks with large norm.

If a matrix has a supervariable structure, then it is natural to associate each supervariable with a block. Assuming that the variables in a supervariable are numbered consecutively, then this blocking does not require reordering the matrix. If blocks larger than the size of the supervariables are desired, then larger blocks can be formed by amalgamating supervariables.

When amalgamating adjacent supervariables, it is important that these supervariables represent adjacent or nearby nodes or elements in the PDE mesh. This is because there is no advantage in grouping variables together that are not coupled within the block. To help ensure that adjacent supervariables are coupled, the matrix should be ordered so that variables that are close in the ordering are nearby in the PDE mesh. This can be accomplished by applying a locality-preserving ordering to the supervariable structure (called the *condensed graph*) before the supervariables are amalgamated. Fortunately, as already observed, locality-preserving orderings such as RCM are also beneficial for incomplete factorization preconditioners.

In this paper, instead of identifying a supervariable structure and then applying RCM ordering to that structure, we simply apply RCM ordering to the original rows and columns of the matrix before identifying supervariables. RCM ordering tends to preserve any existing supervariable structure because variables belonging to a supervariable will tend to stay numbered together.

The input to the supervariable amalgamation algorithm is the sparsity pattern of a matrix that has been reordered using RCM. The supervariable structure is found by comparing the sparsity patterns of adjacent matrix columns. Supervariables are then amalgamated into blocks with a given maximum block size m by merging adjacent supervariables. The algorithm does not alter the ordering of the matrix and only outputs the grouping of consecutive rows and columns into blocks. We note that when identifying supervariables, if there are supervariable blocks that are larger than the maximum allowed block size, then these blocks are recursively divided until the block size limit is satisfied.

When identifying supervariables, techniques based on hashing are available to rapidly determine if the structure of two columns differ [6,32] (further discussion of efficient algorithms for identifying supervariables is given in [22]). In some cases, two columns may be associated to different variables at the same PDE grid point, but may not have identical structure because of numerical zero values in the columns. In this case, approximate matching of column structures can be useful, and the "cosine" between the structure of two columns can be checked [32]. In this paper, however, we use exact matching to identify supervariables. (In general, the cosine technique can be used to identify variables that are "near" one another; however, we use the simpler RCM algorithm for this.)

2.4. Other blocking techniques

Many blocking techniques can be interpreted as graph partitioning. A sparse matrix can be associated with a graph, where the matrix rows or columns (in the symmetric case) are identified with graph *nodes*, and nonzeros are identified with graph *edges*. Graph-based approaches can be used to find a blocking of a sparse matrix, where each block corresponds to a partition of the graph nodes. Usually, the partitions of the graph are contiguous, and thus the nodes in the same partition are nearby in the graph. Graph partitioning techniques minimize the number of edges shared between partitions, which corresponds to minimizing the number of nonzeros that fall outside the diagonal block structure being created. This is in contrast to supervariable amalgamation, where the number of nonzero entries outside the diagonal block structure is not explicitly minimized.

Graph-based approaches generally block together matrix rows and columns that are not necessarily numbered together; therefore, reordering the matrix is necessary. However, the ordering of the blocks is not specified by a graph partitioning. If an ordering to produce an accurate incomplete factorization is desired, then it can be effective to order the block structure (the condensed graph) using RCM or the Sloan algorithm, especially if the block sizes are small. The ordering of the variables within blocks is also flexible; one choice is to order the variables within a block in ascending order of their original numbering.

Graph partitioning approaches, e.g., the package METIS [25], typically partition a coarse version of the graph, and then uncoarsen and refine the partitioning over several stages called levels. When large numbers of small partitions are desired (as in our case where the partitions are often so small that the corresponding diagonal blocks are almost dense) then the original graph cannot be coarsened too far, and the quality of the result depends highly on the quality of the partitioning of the coarsest version of the graph.

A different graph-based blocking approach is used by PABLO (parametrized block ordering) [20,28], which is designed for block Jacobi and block Gauss–Seidel preconditioning. Here, blocks are constructed one-by-one. A block initially consists of a single node. Additional nodes are added to the block if they satisfy a "fullness" criterion (maximizing the number of connections between nodes in the same block) or a "connectivity" criterion (minimizing the number of connections between blocks) until a maximum block size is reached.

All the algorithms we have described so far utilize only the structure of the sparse matrix, and not its values. More effective methods may be constructed if values are also considered, explicitly placing large entries in the block diagonal portion of the blocked matrix. METIS can partition graphs with weighted edges by minimizing weighted edge cuts. Edge weights may be defined as being proportional to the absolute values of the corresponding off-diagonal matrix entries. PABLO has been extended to TPABLO

(threshold PABLO) and the family of PABLO algorithms is implemented as the package XPABLO [20]. TPABLO uses a threshold on the magnitude of the nonzeros in a matrix when applying its fullness and connectivity criteria. All nonzeros above the threshold are treated equally, while those below the threshold are discounted. A possible disadvantage of TPABLO is that it does not prioritize a large nonzero over a smaller nonzero if they are both above or below the threshold.

A blocking algorithm sensitive to matrix values that does not use a threshold is the HDPRE algorithm of Duff and Kaya [16]. A block diagonal structure is generated by considering the nonzero values in decreasing order of their size. The HDPRE algorithm is designed for matrices with nonsymmetric structure and utilizes the idea of partitioning a graph such that each subcomponent is strongly connected [35]. For our symmetric positive definite test problems, we use a much simpler version of the algorithm, which we call *priority blocking* that we describe next.

2.5. Priority blocking algorithms

In priority blocking, a block is identified by a set of graph nodes. Initially, each graph node corresponds to a block or set of size 1. Each nonzero or graph edge considered creates a new diagonal block, to which the nonzero will belong, by merging two existing blocks, unless the resulting block exceeds a prescribed maximum size. By considering nonzeros in decreasing order of their size, large nonzeros are effectively placed into the diagonal block structure before small nonzeros.

The priority blocking algorithm is specified as Algorithm 1. For efficiency, a priority queue, implemented as a heap, is used to select the nonzeros in order of decreasing size. The algorithm provides a blocking but does not specify an ordering. As for other graph-based blocking approaches, RCM ordering can be applied to the condensed graph.

The priority blocking algorithm with *dynamic edge weights* is specified in Algorithm 2. This algorithm attempts to improve the priority blocking algorithm by adjusting the edge weights (absolute value of the corresponding nonzero element) when blocks are merged. Consider edge (i, j) where node i belongs to set S_{l_i} and node j belongs to set S_{l_j}. Also assume that there exists a node k such that edges (i, k) and (j, k) exist in the graph but have yet to be considered by the priority blocking algorithm. When sets S_{l_i} and S_{l_j} are merged, then the edges (i, k) and (j, k) collapse into a single edge between node k and the newly created block. The new edge replaces the collapsed edges and has weight equal to the sum of the weights of the edges it replaces. This technique is analogous to graph coarsening (see, for example, [25]).

Again, a priority queue, implemented as a heap, is used to store and extract the edges in the graph. When a new edge replaces two or more collapsed edges, one of these edges has its weight increased and the other edges are removed. This entails updating the heap data structure. Although increasing the priority of an item in a heap can be done efficiently, in time proportional to $\log_2$ of the number of elements in the heap, not all priority queues implement this operation. In this case, we can simply add the edge with the updated weight to the heap. The extra edges (with smaller weights) will be removed from the heap in turn, but will have no effect on the result of the algorithm. The least scalable component of the priority blocking algorithms is choosing the edge with the largest weight. This cost is $O(n_z \log_2 n_z)$ when a heap is used, where n_z is the number of nonzeros in the matrix.

Algorithm 1 Priority blocking. The input to the algorithm is a symmetric positive definite matrix $A = \{a_{ij}\}$ and a maximum block size m. The output is a set of blocks, where each block is a set of graph nodes (i.e., row or column indices). Each node is always in exactly one block. A label l_i maps a node i to a block index. We use $|S|$ to denote the number of elements in set S.

```
Initialize blocks: S_i = {i}, 1 ≤ i ≤ n
Initialize labels: l_i = i, 1 ≤ i ≤ n
Initialize heap: for all a_ij ≠ 0, i < j, add (i, j) to heap H with priority |a_ij|/√(a_ii a_jj)
while H is not empty do
  (i, j) = remove_largest(H)
  if |S_{l_i}|+|S_{l_j}| ≤ m then
    l_k = i for all k ∈ S_{l_j}
    S_{l_i} = merge(S_{l_i}, S_{l_j})
    S_{l_j} = ∅
  end if
end while
print S_i for all |S_i| > 0
```

Algorithm 2 Priority blocking with dynamic edge weights. The inputs and outputs are identical to those of Algorithm 1.

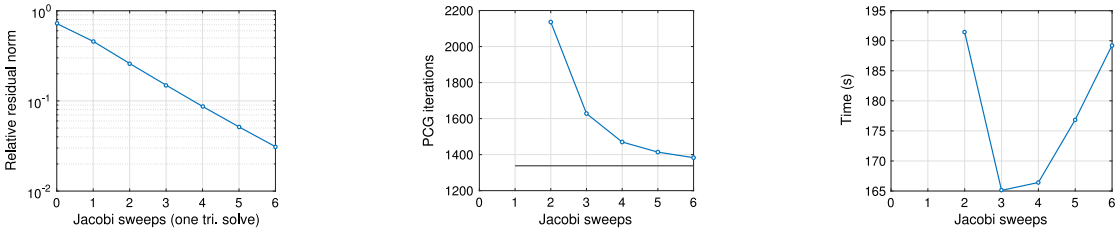
```
Initialize blocks: S_i = {i}, 1 ≤ i ≤ n
Initialize labels: l_i = i, 1 ≤ i ≤ n
Initialize heap: for all a_ij ≠ 0, i < j, add (i, j) to heap H with priority |a_ij|/√(a_ii a_jj)
Initialize adjacency lists: N_i = {(j, |a_ij|/√(a_ii a_jj)), a_ij ≠ 0}, 1 ≤ i ≤ n
while H is not empty do
  (i, j) = remove_largest(H)
  if |S_{l_i}|+|S_{l_j}| ≤ m then
    N_{l_i} = merge adjacency lists N_{l_i} and N_{l_j}
    N_{l_j} = ∅
    if (k, v) ∈ N_{l_i} is the result of combining two or more adjacencies then
      Add (l_i, k) to heap H with priority v
    end if
    l_k = i for all k ∈ S_{l_j}
    S_{l_i} = merge(S_{l_i}, S_{l_j})
    S_{l_j} = ∅
  end if
end while
print S_i for all |S_i| > 0
```

3. Results and discussion

Our goal in this paper is to test the use of iterative methods for approximately solving the triangular systems that arise when an incomplete Cholesky factorization is used as a preconditioner in PCG. We are interested in assessing the overall effectiveness of both scalar Jacobi and block Jacobi methods. For the latter, we briefly examine the use of alternative blocking strategies.

In our numerical experiments, we use symmetric positive definite matrices from the SuiteSparse Matrix Collection [13]. For the results reported in this paper, RCM ordering is used unless indicated otherwise. Each test matrix A is scaled on the left and right by a diagonal matrix whose jth diagonal entry is the reciprocal square root of the 2-norm of column j of A. We use IC(k) ($k = 0, 1$) to denote an incomplete Cholesky factorization of level k; IC(k)-PCG denotes PCG with IC(k) preconditioning. For each matrix, we use IC(k)-PCG to solve a system of equations where the right-hand side vector is the vector of all ones. The initial approximation to the solution is the zero vector. IC(k)-PCG is considered to have converged when the residual 2-norm relative to the initial residual 2-norm is less than 10^{-6}.

Timings (all reported in seconds) are collected using an NVIDIA Tesla K40 GPU, using double precision arithmetic. This GPU consists of 15 Streaming Multiprocessors, each containing 64 double precision (DP) units. For both the scalar and block Jacobi triangular solves, we map matrix rows to threads. Theoretically, this strategy fills the DP units for matrices with at least 960 rows. However, for efficient processing it is essential to oversubscribe the processing units and switch between contexts to hide main memory latency. In our experiments, we consider matrices with 1000–1.5 million rows. The small problems are too small to efficiently utilize the hardware, but we expect that the GPU easily hides the memory latency for the large problems.

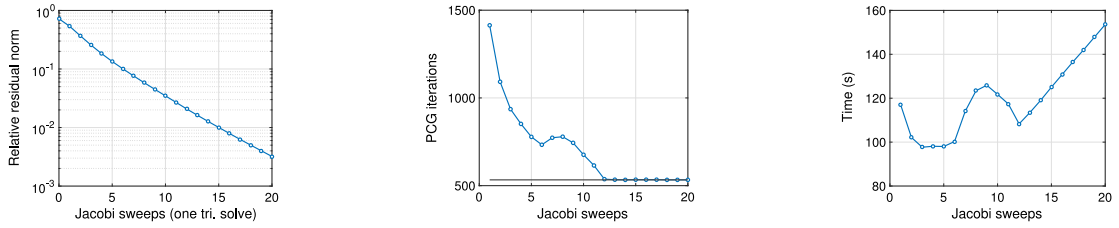


(a) Scalar Jacobi triangular solve convergence.

(b) IC(0)-PCG convergence.

(c) IC(0)-PCG run time.

Fig. 1. For Hook_1498, the effects of increasing the number of scalar Jacobi sweeps. Using exact triangular solves requires 1338 IC(0)-PCG iterations and 660 s.



(a) Scalar Jacobi triangular solve convergence.

(b) IC(0)-PCG convergence.

(c) IC(0)-PCG run time.

Fig. 2. For Geo_1438, the effects of increasing the number of scalar Jacobi sweeps. Using exact triangular solves requires 533 IC(0)-PCG iterations and 320 s.

The algorithms are implemented using the MAGMA-sparse software package, part of the MAGMA open source software library [23]. MAGMA-sparse uses libraries and compilers from CUDA version 8.0. In particular, exact triangular solves using parallel level-scheduled forward/backward substitution (trsm) use the code implemented in the NVIDIA cuSPARSE library. For the block Jacobi solves, the diagonal blocks are inverted using a batched Gauss–Jordan elimination GPU kernel [4]. We note that we store the inverse of a block diagonal matrix, D^{-1}, as a sparse matrix. For higher efficiency, D^{-1} could be stored using a specialized data structure as a sequence of dense matrices corresponding to its diagonal blocks.

3.1. Examples

In Fig. 1 we present a detailed analysis for the matrix Hook_1498. Fig. 1(a) shows the residual norm history (relative to the norm of the right-hand side) for a single solve with the lower triangular IC(0) factor of the matrix using the scalar Jacobi method. We use a random right-hand side vector, and zero sweeps corresponds to using the initial guess (3). For solving a system of equations with the Hook_1498 matrix, Fig. 1(b) reports the number of IC(0)-PCG iterations required for convergence when up to 6 Jacobi sweeps are used for each triangular solve with the IC(0) factors. As the number of Jacobi sweeps increases, the number of iterations approaches 1338, which is the number required if exact triangular solves are used, corresponding to the conventional approach.

A single scalar Jacobi sweep on an IC(0) factor takes 6.5 ms for this problem, which is more than 33× faster than the cuSPARSE triangular solve, taking 216.7 ms. As we see in Fig. 1(b), a single Jacobi sweep does not provide sufficient accuracy, and also few sweeps degrade the preconditioner quality making additional PCG iterations necessary. In terms of the PCG run time, the optimal number of Jacobi sweeps for this problem is 3, see Fig. 1(c). With 3 sweeps, the residual norm for a triangular solve is reduced by less than one order of magnitude (see Fig. 1(a)).

Overall, IC(0)-PCG using exact level-scheduled triangular solves implemented by cuSPARSE requires 660 s compared to 165 s when 3 Jacobi sweeps are used, a speedup of a factor of 4. We note that this runtime does not include the preconditioner setup time, which can be large for the exact triangular solves as generating the level scheduling data structures is generally a sequential process.

Fig. 2 shows the corresponding results for the matrix Geo_1438. The behavior here is different from that observed for the previous example. In particular, the IC(0)-PCG iteration run time (Fig. 2(c)) is not convex. In terms of run time, the optimal number of Jacobi sweeps for this problem is 3 (although the times are very similar for 4 and 5 sweeps), giving a run time speedup of a factor of 3.3.

We observed the behaviors illustrated by these two examples for other matrices, but other behaviors are also possible. For example, as the number of Jacobi sweeps increases, the PCG iteration run time can initially increase and then decrease (possibly below its initial value) before increasing again. Moreover, the residual norm may initially diverge before converging, and a large number of sweeps may be needed to achieve PCG convergence. This is the case for the test matrix bcsstk24. Although this example is relatively small (of order 3562), it is challenging to solve: the matrix has large off-diagonal entries (even after scaling), both diagonal preconditioning and symmetric Gauss–Seidel preconditioning fail to give convergence within a number of steps equal to the dimension of the matrix, and the IC(0) factorization breaks down (that is, a zero or negative pivot is encountered during the factorization). However, the IC(1) factorization exists and provides a good preconditioner, with convergence of IC(1)-PCG in 56 iterations using exact triangular solves. Thus we use IC(1) preconditioning for this matrix.

Fig. 3 (left) shows the residual norm for solving with the lower triangular incomplete Cholesky factor of bcsstk24 using the Jacobi method. As the iteration matrix for this factor is highly non-normal, we observe a large increase in the residual norm before convergence sets in. Using fewer than 150 Jacobi sweeps with IC(1) does not give a useful preconditioner. Fig. 3 (right) shows the residual norm if block Jacobi is used, using supervariable amalgamation

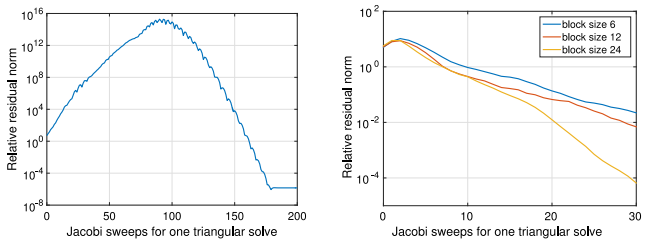


Fig. 3. For bcsstk24, the relative residual norm for a triangular solve with the lower triangular IC(1) factor and a random right-hand side, without blocking (left) and with supervariable amalgamation (right).

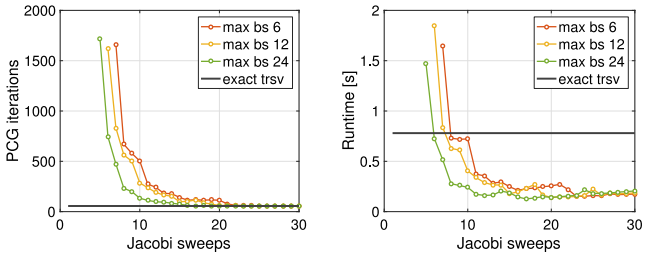


Fig. 4. For bcsstk24, the effect of increasing the number of Jacobi sweeps on IC(1)-PCG iteration counts (left) and run times (right). Different maximum block sizes are used (max bs). The result of using exact triangular solves is also shown.

and maximum block sizes $m = 6$, 12, and 24. (Note that bcsstk24 essentially has a supervariable structure with a block size 6.) Again there is an initial increase in the residual norm when the number of block Jacobi sweeps is small, but this increase is much less than for scalar Jacobi and the number of sweeps for which it occurs is also much less.

Fig. 4 shows IC(1)-PCG iteration counts and run times for different numbers of block Jacobi sweeps. We observe that despite the difficulties using scalar Jacobi sweeps, IC(1)-PCG converges rapidly using 15 block Jacobi sweeps. This example demonstrates the effectiveness of block Jacobi compared to scalar Jacobi iterations.

3.2. Increasing the factorization accuracy

In Fig. 5 we analyze the relation between the optimal number of scalar Jacobi sweeps and the incomplete Cholesky preconditioner with different levels of fill-in for the thermal_1 matrix. The PCG iteration counts in Fig. 5(a) indicate that the number of Jacobi sweeps necessary to match using exact triangular solves (solid lines) increases with the level of fill-in. Fig. 5(b) reveals that PCG using scalar Jacobi sweeps for the preconditioner application is faster in all cases than PCG based on exact triangular solves. Fig. 5(c) suggests that the optimal number of Jacobi sweeps (giving the lowest PCG execution time) increases with the fill-in level of incomplete Cholesky. This is consistent with the idea that one should try to solve more accurately with a factorization that is more accurate.

3.3. Comprehensive results for Jacobi and block Jacobi iterations

This section presents results for a large number of test problems. With a few exceptions, we chose all the real symmetric positive definite matrices from the SuiteSparse Matrix Collection that are of order at least 1000 and that are not diagonal. Three matrices from the af_shell set were discarded because of their similarity to af_shell7, which was retained. The Andrews matrix was removed because it uses an integer data type for matrix values. The audikw_1 and Flan_1565 matrices were also removed as the IC(1) preconditioner for these matrices, even after RCM ordering, could not fit in GPU memory. We also discarded three matrices, bloweybq, LFAT5000, and LF10000, whose condition numbers reported at the SuiteSparse Matrix Collection website exceed 10^{17}. This gives us a set of 161 matrices. From this set, we chose two subsets: those that can be solved by IC(0)-PCG (using exact triangular solves) within 3000 iterations and those that can be solved in the same way using IC(1)-PCG. This resulted in a set of 69 matrices for IC(0)-PCG and a set of 78 matrices for IC(1)-PCG. We list the matrices of the two sets along with some key characteristics in the Appendix.

Table 1
Number of problems that can be solved when exact triangular solves in IC(k)-PCG ($k = 0$, 1) are replaced by Jacobi and block Jacobi triangular solves.

	IC(0)	IC(1)
Num. solved using exact triangular solves	69	78
Num. solved using Jacobi triangular solves	58 (84%)	58 (74%)
Num. solved using block Jacobi triangular solves	65 (92%)	73 (94%)

Our goal is to find what fraction of these test sets can be solved using IC(k)-PCG when the exact triangular solves are replaced by Jacobi and block Jacobi iterations. We allow up to 20 Jacobi or block Jacobi sweeps for each triangular solve. For block Jacobi, we use supervariable amalgamation with a maximum block size of $m = 12$. This was chosen because many PDE systems have 3, 4, or 6 degrees of freedom per node.

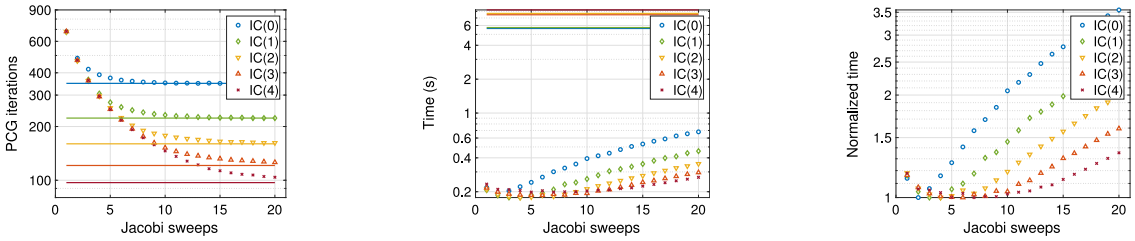
Table 1 summarizes our findings. In the case of IC(0), 84% of the problems that can be solved by IC(0)-PCG can also be solved when the exact triangular solves are replaced by Jacobi iterations. The fraction increases to 92% when block Jacobi is used. For IC(1), the fractions are 74% and 94%, respectively. These results show that for a significant proportion of our test problems, the exact triangular solves can be replaced by iterative solves. We note, however, that level-based incomplete Cholesky factorizations may not provide the best preconditioner among all possible preconditioners for all of the test problems; our results only show the potential efficacy of iterative triangular solves if a level-based incomplete Cholesky factorization is used as the preconditioner.

For the problems that are successfully solved using Jacobi and block Jacobi, Fig. 6 shows how many Jacobi and block Jacobi sweeps, respectively, are needed to minimize the IC(k)-PCG iteration run time. The figures are histograms showing the frequency of the number of sweeps. In all cases, there is a clustering of the optimal number of sweeps around 3, and for most of the problems the optimal number of sweeps is fewer than 10.

Fig. 7 shows the speedup of the PCG run iteration time when using Jacobi and block Jacobi triangular solves (using the best number of sweeps for each problem) compared to using exact level-scheduled triangular solves implemented within cuSPARSE. The plots show that the speedup can exceed 10× and that only a small number of problems suffer a slow down. The results also show that compared to scalar Jacobi, block Jacobi results in fewer problems experiencing a slow down.

We observe from Table 1 that using Jacobi iterations for IC(1)-PCG solves results in a smaller percentage of problems being solved compared to using Jacobi iterations for IC(0)-PCG. This suggests that the IC(1) incomplete factors are less diagonally dominant than the IC(0) factors. On the other hand, we found a similar percentage of problems that were solved when block Jacobi iterations are used for IC(0)-PCG and for IC(1)-PCG.

To investigate these issues, we compute the *off-diagonal* dominance of the IC(0) and IC(1) incomplete factors. We define the

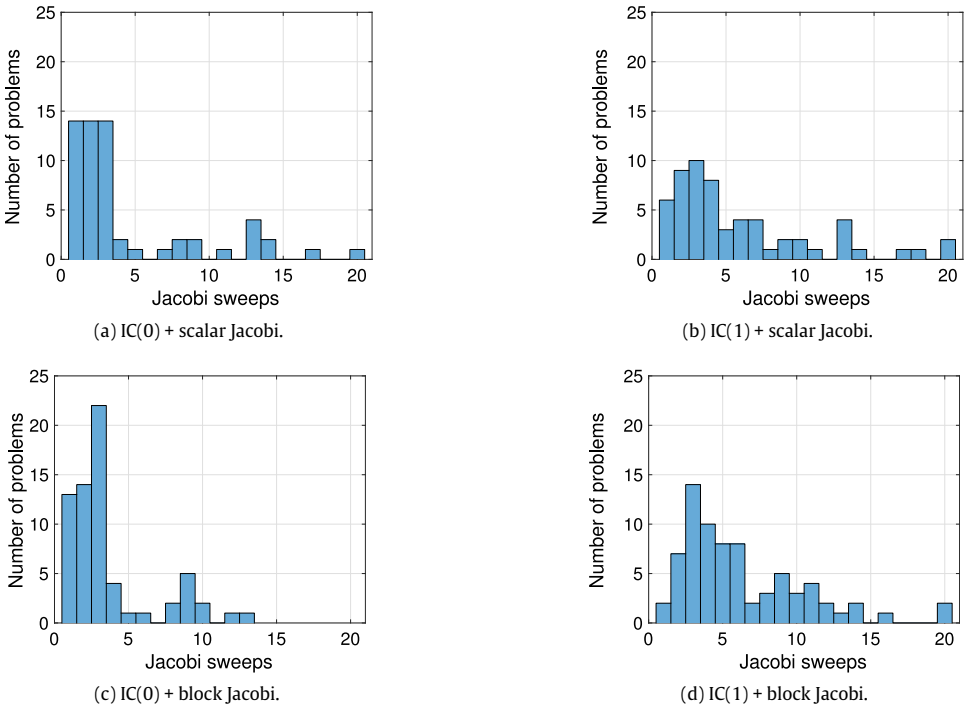


(a) PCG iterations using IC preconditioning with different levels of fill.

(b) PCG run time.

(c) PCG run time, normalized by the fastest run time for each IC level.

Fig. 5. For thermal_1, the effects of increasing the number of scalar Jacobi sweeps.



(a) IC(0) + scalar Jacobi.

(b) IC(1) + scalar Jacobi.

(c) IC(0) + block Jacobi.

(d) IC(1) + block Jacobi.

Fig. 6. Optimal number of sweeps to minimize PCG iteration run time. The block Jacobi results (bottom row) use supervariable amalgamation and a maximum block size of 12.

off-diagonal dominance of an $n \times n$ matrix A as

$$\frac{1}{n}\sum_{i}^{n}\sum_{j\neq i}^{n}\frac{|a_{ij}|}{|a_{ii}|}.$$

Likewise, we define *block off-diagonal* dominance of a matrix A with $k \times k$ blocking by

$$\frac{1}{k}\sum_{i}^{k}\sum_{j\neq i}^{k}\frac{\|A_{ij}\|}{\|A_{ii}^{-1}\|^{-1}},$$

where A_{ij} denotes (i, j)th block of the matrix. This definition follows the definition of *block diagonal dominance* of Feingold and Varga [19]. For convenience, we use the Frobenius norm in the above definition.

Fig. 8 plots these two quantities for the 65 matrices in our test set that have both IC(0) and IC(1) factors. The results show that for nearly every matrix, the (block) off-diagonal dominance of the IC(1) factor is either the same or higher than for the IC(0) factor. This suggests that Jacobi iterations will become less effective as more accurate incomplete factorizations are used. However, the numerical results we have already presented indicate that using block Jacobi with IC(1) we are able to solve a large fraction of the test problems where IC(1) was successful with exact solves.

3.4. Results with priority blocking

The blocking technique used may have a large influence on the results of block Jacobi iteration. For simplicity, we used supervariable amalgamation in our experiments. In this section, however, we briefly show that more delicate blocking techniques may be effective. Our motivation is not to advocate a single blocking method, but to emphasize that a user should carefully select or design a blocking method that is suitable for the problem.

From the results for IC(0)-PCG in Table 1, there are four test examples that could not be solved when using block Jacobi solves with supervariable amalgamation but were solved using direct triangular solves. These examples are problems cfd1, crankseg_1,

Table 2
For IC(0), results using exact triangular solves (cuSPARSE) for problems using three different orderings: RCM, PB, and PBD. The table shows the number of IC(0)-PCG iterations, the iteration run time, and the number of levels in the level scheduling.

	IC(0)-PCG iterations			Run time			Number of levels		
	RCM	PB	PBD	RCM	PB	PBD	RCM	PB	PBD
cfd1	445	496	537	24.8	15.7	32.4	2 831	1 249	2 903
crankseg_1	88	190	163	23.3	99.8	96.7	6 841	13 594	16 086
Emilia_923	305	604	1346	226.1	99.7	269.8	33 495	3 749	4 938

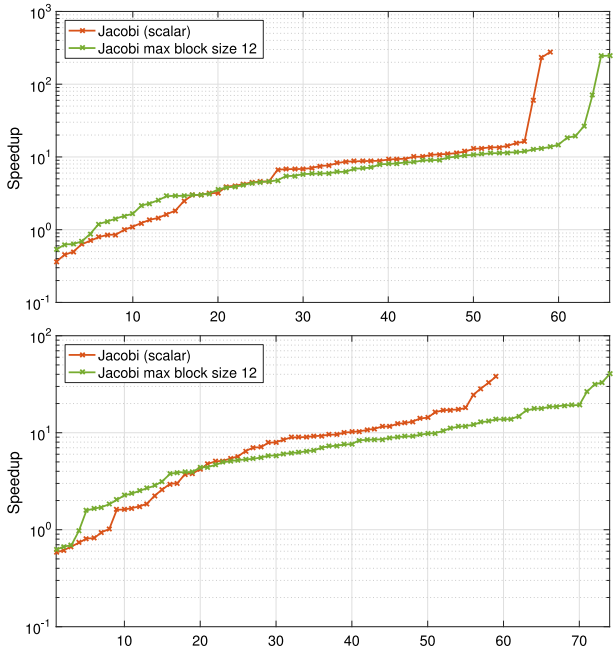


Fig. 7. Speedup of the PCG iteration run time when using Jacobi and block Jacobi triangular solves compared to using exact triangular solves. Top: IC(0). Bottom: IC(1). Matrix problems on the x-axis are sorted by speedup, and are not the same for the different curves. Note that the curves have different lengths because different numbers of problems can be solved for each preconditioner configuration (block size and IC level). These speedups are for the same runs shown in the previous histograms.

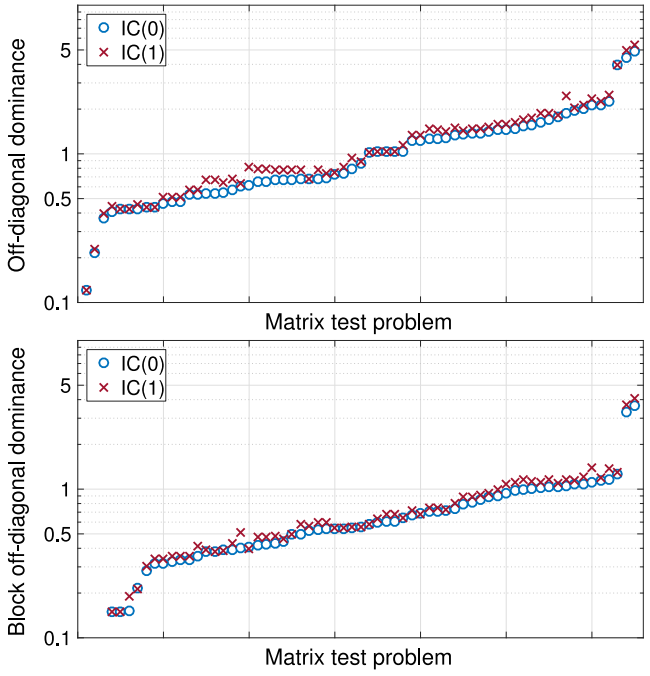


Fig. 8. Off-diagonal dominance (top) and block off-diagonal dominance (bottom) for 65 test matrices, with matrices sorted by the value for the IC(0) factor. Smaller values mean a matrix is more diagonally dominant.

crankseg_2, and Emilia_923. We now show that these can be solved using the two priority blocking methods mentioned earlier. Note that detailed results for crankseg_2 are not shown, because of its similarity to crankseg_1.

Recall that once the blocks are identified by a priority blocking algorithm, an RCM ordering is applied to the condensed graph to determine the final ordering of the matrix. This ordering generally differs from an RCM ordering of the original matrix. We refer to the orderings resulting from the priority blocking and priority blocking with dynamic edge weight algorithms as PB and PBD orderings, respectively. We first compare the effect of the RCM, PB, and PBD orderings on convergence and parallel performance using IC(0)-PCG with exact triangular solves. For the PB and PBD orderings, the blockings are determined using a maximum block size of 12, the same value as was used earlier for supervariable amalgamation. Results are given in Table 2. In addition to the IC(0)-PCG iteration counts and the iteration run time, the number of levels in the level scheduling for the exact triangular solves is reported because this affects how efficiently the exact triangular solves are parallelized. We observe that for our three test matrices, the PB and PBD orderings generally lead to increased iteration counts, confirming that RCM is a good ordering for incomplete Cholesky factorizations. However, we also observe the anomaly that the PB and PBD orderings can reduce the run times. Because of the locations of large nonzeros in the matrix, PB and PBD can (but not always) result in an ordering that *reduces* the number of levels, meaning that the level-scheduled triangular solves can be more efficiently parallelized, leading to a decrease in the iteration run time.

We next use the PB and PBD orderings for IC(0)-PCG with block Jacobi solves. We also test orderings generated from METIS partitionings of the matrices using edge weights (with average block size 12). As with the PB and PBD orderings, the METIS ordering is obtained by applying RCM to a condensed graph. In Fig. 9 (bottom row) we present results for the METIS ordering for problems cfd1 and crankseg_1; the IC(0) factorization broke down for problem Emilia_923 with the METIS ordering so is omitted. Note that we also tested TPABLO on these matrices using a maximum block size of 12 and a range of parameter values, including the default. However, the block Jacobi solves converged slowly and did not provide effective preconditioners.

Fig. 9 shows the IC(0)-PCG iteration counts (left column) and the iteration run time (right column) as the number of block Jacobi sweeps is varied. Although the block Jacobi triangular solves were not effective when supervariable amalgamation was used for these matrices, we observe that after only a small number of block Jacobi sweeps (fewer than 10), the number of IC(0)-PCG iterations is

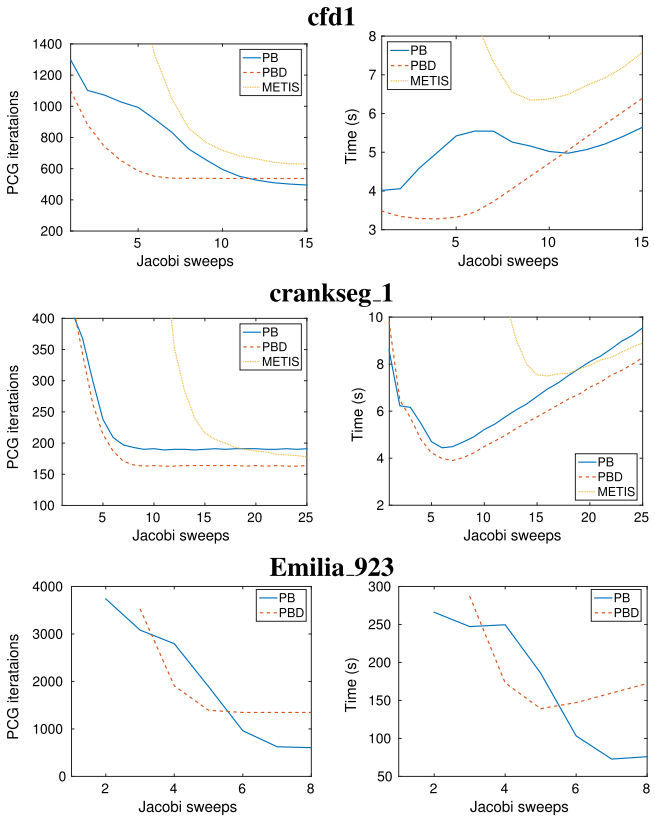


Fig. 9. IC(0)-PCG iterations and iteration run time as a function of the number of block Jacobi sweeps, using the PB, PBD, and METIS orderings.

Table 3
The optimal number of block Jacobi sweeps and the corresponding number of IC(0)-PCG iterations and iteration time for the PB and PBD orderings.

	PB			PBD		
	sweeps	iters	time	sweeps	iters	time
cfd1	1	1298	4.0	4	651	3.3
crankseg_1	6	209	4.4	7	171	3.9
Emilia_923	7	625	72.8	5	1392	139.2

almost the same as for exact triangular solves. The METIS orderings give IC(0)-PCG iteration run times that are worse than those for the PB and PBD orderings.

Table 3 shows the optimal number of block Jacobi sweeps for the PB and PBD orderings to minimize the iteration run time. The corresponding number of PCG iterations and the run time are also reported. The timings in this table can be compared to those in Table 2. We observe that for each system, the best run times for block Jacobi sweeps are always faster than using exact triangular solves. If we compare block Jacobi solves using the PB ordering to exact triangular solves using RCM, the run time speedups are 6.2, 5.3, and 3.1 for our three test matrices.

From the results for IC(1) in Table 1, there are five problems that could not be solved when using block Jacobi with supervariable amalgamation. These problems are nd3k, crankseg_1, crankseg_2, m_t1, and Emilia_923. Once again, we try to solve these problems with the two priority blocking methods. In this case, we are able to solve all the problems except m_t1. Tables 4 and 5 report results for IC(1)-PCG analogous to those for IC(0)-PCG shown in Tables 2 and 3, respectively. (Again, we only show results for crankseg_1 and not for crankseg_2.) We see that IC(1)-PCG with iterative triangular solves is faster than using exact triangular solves.

Table 5
The optimal number of block Jacobi sweeps and the corresponding number of IC(1)-PCG iterations and iteration time for PB and PBD ordering.

	PB			PBD		
	sweeps	iters	time	sweeps	iters	time
nd3k	22	69	1.8	23	60	1.4
crankseg_1	9	61	3.0	9	54	2.6
Emilia_923	8	496	99.0	5	1181	167.3

The system with matrix m_t1 could not be solved with IC(1)-PCG using either Jacobi or block Jacobi sweeps. We note that m_t1 has large off-diagonal entries, essentially as large as the diagonal values themselves. When a matrix has a full subdiagonal of large nonzero values, for example, there is no blocking that can prevent some of these large entries from being outside a block diagonal structure.

4. Conclusions

The main goal of this paper is to understand the applicability of Jacobi and block Jacobi iterations for solving the sparse triangular systems arising from incomplete Cholesky preconditioning. As always, artificial problems can be constructed to defeat the Jacobi approach. However, we find that for a diverse set of *realistic* symmetric positive definite test problems using IC(0)-PCG and IC(1)-PCG, Jacobi iterations are effective for a large proportion of the problems. We also showed that by using block Jacobi, robustness could be enhanced. Even better results for some problems could be obtained by using blocking techniques that use matrix values (e.g., priority blocking).

We note that these results do depend on the parallelism of the computing architecture. The use of (block) Jacobi is targeted for high parallelism cases, like GPUs and manycore CPUs. With lower amounts of parallelism, there are two main effects: (1) the advantage of using (block) Jacobi over exact solves inside PCG is smaller or may not be effective, and (2) it is possible that the PCG solve time as a function of the number of sweeps might only increase and not decrease, i.e., more sweeps always increase the PCG solve time because the (block) Jacobi triangular solves are not computed fast enough for there to be an advantage to using multiple sweeps.

This paper did not address how to choose the optimal number of Jacobi sweeps to use in the PCG solver. A fixed number of sweeps is desirable, as the preconditioner is then a fixed operator and a "flexible" solver is not needed. However, in practice, the number of sweeps could be adjusted dynamically and the iterations restarted based on the convergence of PCG. If a flexible solver is available, e.g., FGMRES [31] or flexible conjugate gradients [7,27],

Table 4
For IC(1), results using exact triangular solves (cuSPARSE) for problems using three different orderings: RCM, PB, and PBD. The table shows the number of IC(1)-PCG iterations, the iteration run time, and the number of levels in the level scheduling.

	PCG iterations			Time			Num. levels		
	RCM	PB	PBD	RCM	PB	PBD	RCM	PB	PBD
nd3k	51	63	58	38.1	50.4	44.7	7 716	7 564	7 052
crankseg_1	40	59	49	39.2	177.9	142.6	15 856	33 357	32 864
Emilia_923	102	496	1121	149.3	359.8	1220.8	60 264	17 913	31 071

Table 6
Table listing the matrices along with key characteristics that are included in the IC(0) tests.

Name	#rows	#nonzeros	Name	#rows	#nonzeros
e1138_bus	1 138	4 054	wathen100	30 401	471 601
Chem97ZtZ	2 541	7 361	gridgena	48 962	512 084
bcsstk08	1 074	12 960	apache1	80 800	542 184
mhd3200b	3 200	18 316	wathen120	36 441	565 761
bcsstm12	1 473	19 659	thermal1	82 654	574 458
mhd4800b	4 800	27 520	crystm03	24 696	583 770
plbuckle	1 282	30 644	finan512	74 752	596 992
bcsstk27	1 224	56 126	thermomech_TC	102 158	711 558
nasa2146	2 146	72 250	thermomech_TK	102 158	711 558
fv1	9 604	85 264	Pres_Poisson	14 822	715 804
fv2	9 801	87 025	G2_circuit	150 102	726 674
crystm01	4 875	105 339	bundle1	10 581	770 811
bodyy4	17 546	121550	Dubcova2	65 025	1 030 225
aft01	8 205	125567	thermomech_dM	204 316	1 423 116
bodyy5	18 589	128 853	t2cubes_sphere	101 492	1 647 264
bodyy6	19 366	134 208	qa8fm	66 127	1 660 579
ted_B	10 605	144 579	cfd1	70 656	1 825 580
ted_B_unscaled	10 605	144 579	Dubcova3	146 689	3 636 643
Muu	7 102	170 134	parabolic_fem	525 825	3 674 625
t2dah_e	11 445	176 117	offshore	259 789	4 242 673
obstclae	40 000	197 608	pdb1HYS	36 417	4 344 765
torsion1	40 000	197608	apache2	715 176	4 817 870
jnlbrng1	40 000	199 200	tmt_sym	726 713	5 080 961
minsurfo	40 806	203 622	G3_circuit	1 585 478	7 660 826
s1rmt3m1	5 489	217651	thermal2	1 228 045	8 580 313
s2rmt3m1	5 489	217681	crankseg_1	52 804	10614210
Dubcova1	16 129	253 009	crankseg_2	63 838	14 148 858
s1rmq4m1	5 489	262 411	af_shell7	504 855	17 579 155
s2rmq4m1	5 489	263 351	bundle_adj	513 351	20207907
bcsstk16	4 884	290378	Emilia_923	923 136	40 373 538
crystm02	13 965	322905	Hook_1498	1 498 023	59 374 451
shallow_water1	81 920	327680	Geo_1438	1 437 960	60 236 322
shallow_water2	81 920	327680	Trefethen_2000	2 000	41 906
Kuu	7 102	340200	Trefethen_20000	20 000	554 466
gyro_m	17 361	340 431			

Table 7
Table listing the matrices along with key characteristics that are included in the IC(1) tests.

Name	#rows	#nonzeros	Name	#rows	#nonzeros
e1138_bus	1 138	4 054	crystm02	13 965	322 905
Chem97ZtZ	2 541	7 361	shallow_water1	81 920	327 680
bcsstk08	1 074	12 960	shallow_water2	81 920	327 680
mhd3200b	3 200	18 316	Kuu	7 102	340 200
bcsstm12	1 473	19 659	gyro_m	17 361	340 431
bcsstk10	1 086	22 070	wathen100	30 401	471 601
msc01050	1 050	26 198	gridgena	48 962	512 084
mhd4800b	4 800	27 520	apache1	80 800	542 184
plbuckle	1 282	30 644	wathen120	36 441	565 761
bcsstk27	1 224	56 126	thermal1	82 654	574 458
bcsstk14	1 806	63 454	crystm03	24 696	583 770

(*continued on next page*)

then a different number of Jacobi sweeps based on the residual norm reduction for each approximate triangular solve could be used.

A caveat when reading the results of this paper is that no preconditioner is best for all problems, as the choice of preconditioner must be matched to the difficulty (e.g., conditioning) of the problem. Thus in this paper we do not imply that incomplete factorization preconditioning combined with Jacobi iterative triangular solves is the best method to use for all problems. Instead, if incomplete factorization is deemed to be effective for a given problem compared to other preconditioners, then it is likely that Jacobi triangular solves will yield a speedup over traditional triangular solves on highly parallel computer architectures.

Acknowledgments

This research was supported by the Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Numbers DE-SC-0016564 and DE-SC-0016513. This research was also supported by EPSRC grant EP/I013067/1. H. Anzt was partially supported by the "Impuls und Vernetzungsfond of the Helmholtz Association" under grant VH-NG-1241.

Appendix

See Tables 6 and 7.

A.1. Problems for the IC(0) analysis

See Table 6.

A.2. Problems for the IC(1) analysis

See Table 7.

Table 7 (*continued*)

Name	#rows	#nonzeros	Name	#rows	#nonzeros
nasa2146	2 146	72 250	finan512	74 752	596 992
fv1	9 604	85 264	cbuckle	13 681	676 515
fv2	9 801	87 025	thermomech_TC	102 158	711 558
msc04515	4 515	97 707	thermomech_TK	102 158	711 558
crystm01	4 875	105 339	Pres_Poisson	14 822	715 804
bcsstk15	3 948	117 816	G2_circuit	150102	726 674
bodyy4	17 546	121 550	bundle1	10 581	770 811
aft01	8 205	125 567	Dubcova2	65 025	1030 225
bodyy5	18 589	128 853	thermomech_dM	204 316	1 423 116
bodyy6	19 366	134 208	t2cubes_sphere	101 492	1 647 264
ted_B	10 605	144 579	qa8fm	66 127	1 660 579
ted_B_unscaled	10 605	144 579	nd3k	9 000	3 279 690
bcsstk24	3 562	159 910	Dubcova3	146 689	3 636 643
Muu	7 102	170 134	parabolic_fem	525 825	3 674 625
t2dah_e	11 445	176 117	cant	62 451	4 007 383
obstclae	40 000	197 608	pdb1HYS	36 417	4 344 765
torsion1	40 000	197 608	apache2	715 176	4 817 870
jnlbrng1	40 000	199 200	ecology2	999 999	4 995 991
minsurfo	40 806	203 622	tmt_sym	726 713	5 080 961
s3rmt3m3	5 357	207 123	G3_circuit	1 585 478	7 660 826
s1rmt3m1	5 489	217 651	thermal2	122 8045	8 580 313
s3rmt3m1	5 489	217669	m_t1	97 578	9 753 570
s2rmt3m1	5 489	217 681	crankseg_1	52 804	10 614 210
Dubcova1	16 129	253 009	crankseg_2	63 838	14 148 858
s1rmq4m1	5 489	262 411	af_shell7	504 855	17 579 155
s3rmq4m1	5 489	262 943	Emilia_923	923 136	40 373 538
s2rmq4m1	5 489	263 351	Hook_1498	1 498 023	59 374 451
bcsstk16	4 884	290 378	Geo_1438	1 437 960	60 236 322

References

[1] F.L. Alvarado, R. Schreiber, Optimal parallel solution of sparse triangular systems, SIAM J. Sci. Comput. 14 (2) (1993) 446–460.

[2] E.C. Anderson, Y. Saad, Solving sparse triangular systems on parallel computers, Int. J. High Speed Comput. 1 (1989) 73–96.

[3] H. Anzt, E. Chow, J. Dongarra, Iterative sparse triangular solves for preconditioning, Lecture Notes in Comput. Sci. 9233 (2015) 650–661.

[4] H. Anzt, J. Dongarra, G. Flegar, E.S. Quintana-Ortí, Variable-size batched Gauss–Jordan elimination for block-Jacobi preconditioning on graphics processors, Parallel Comput. (2018).

[5] H. Anzt, T.K. Huckle, J. Bräckle, J. Dongarra, Incomplete sparse approximate inverses for parallel preconditioning, Parallel Comput. 71 (Suppl. C) (2018) 1–22.

[6] C. Ashcraft, Compressed graphs and the minimum degree algorithm, SIAM J. Sci. Comput. 16 (6) (1995) 1404–1411.

[7] O. Axelsson, Iterative Solution Methods, Cambridge University Press, Cambridge, 1994.

[8] M. Benzi, W.D. Joubert, G. Mateescu, Numerical experiments with parallel orderings for ILU preconditioners, Electron. Trans. Numer. Anal. 8 (1999) 88–114.

[9] M. Benzi, M. Tůma, A comparative study of sparse approximate inverse preconditioners, Appl. Numer. Math. 30 (1999) 305–340.

[10] E. Chow, A. Patel, Fine-grained parallel incomplete LU factorization, SIAM J. Sci. Comput. 37 (2) (2015) C169–C193.

[11] E. Chow, Y. Saad, Experimental study of ILU preconditioners for indefinite matrices, J. Comput. Appl. Math. 86 (2) (1997) 387–414.

[12] E.H. Cuthill, J. McKee, Reducing the bandwidth of sparse symmetric matrices, in: Proceedings 24th National Conference of the ACM, ACM Press, 1969, pp. 157–172.

[13] T.A. Davis, Y. Hu, The University of Florida sparse matrix collection, ACM Trans. Math. Softw. 38 (1) (2011) 25 pages.

[14] S. Doi, On parallelism and convergence of incomplete LU factorizations, Appl. Numer. Math. 7 (5) (1991) 417–436.

[15] S. Doi, T. Washio, Ordering strategies and related techniques to overcome the trade-off between parallelism and convergence in incomplete factorizations, Parallel Comput. 25 (13–14) (1999) 1995–2014.

[16] I.S. Duff, K. Kaya, Preconditioners based on strong subgraphs, Electron. Trans. Numer. Anal. 40 (2013) 225–248.

[17] I.S. Duff, G.A. Meurant, The effect of ordering on preconditioned conjugate gradients, BIT Numer. Math. 29 (4) (1989) 635–657.

[18] H.C. Elman, E. Agrón, Ordering techniques for the preconditioned conjugate-gradient method on parallel computers, Comput. Phys. Comm. 53 (1–3) (1989) 253–269.

[19] D.G. Feingold, R.S. Varga, Block diagonally dominant matrices and generalizations of the Gerschgorin circle theorem, Pacific J. Math. 12 (4) (1962) 1241–1250.

[20] D. Fritzsche, A. Frommer, D.B. Szyld, Extensions of certain graph-based algorithms for preconditioning, SIAM J. Sci. Comput. 29 (2007) 2144–2161.

[21] S.W. Hammond, R. Schreiber, Efficient ICCG on a shared memory multiprocessor, Int. J. High Speed Comput. 4 (1992) 1–21.

[22] J.D. Hogg, J.A. Scott, A modern analyse phase for sparse tree-based direct methods, Numer. Linear Algebra Appl. 20 (2013) 397–412.

[23] Innovative Computing Lab, Software distribution of MAGMA version 2.2.0, 2016. http://icl.cs.utk.edu/magma/.

[24] M.T. Jones, P.E. Plassmann, Scalable iterative solution of sparse linear-systems, Parallel Comput. 20 (5) (1994) 753–773.

[25] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. 20 (1) (1998) 359–392.

[26] R. Li, Y. Saad, GPU-accelerated preconditioned iterative linear solvers, J. Supercomput. 63 (2) (2013) 443–466.

[27] Y. Notay, Flexible conjugate gradients, SIAM J. Sci. Comput. 22 (4) (2000) 1444–1460.

[28] J. O'Neil, D.B. Szyld, A block ordering method for sparse matrices, SIAM J. Sci. Stat. Comput. 11 (5) (1990) 811–823.

[29] E.L. Poole, J.M. Ortega, Multicolor iccg methods for vector computers, SIAM J. Numer. Anal. 24 (6) (1987) 1394–1417.

[30] A. Pothen, F.L. Alvarado, A fast reordering algorithm for parallel sparse triangular solution, SIAM J. Sci. Comput. 13 (2) (1992) 645–653.

[31] Y. Saad, A flexible inner-outer preconditioned GMRES algorithm, SIAM J. Sci. Stat. Comput. 14 (1993) 461–469.

[32] Y. Saad, Finding exact and approximate block structures for ILU preconditioning, SIAM J. Sci. Comput. 24 (4) (2003) 1107–1123.

[33] J.H. Saltz, Aggregation methods for solving sparse triangular systems on multiprocessors, SIAM J. Sci. Stat. Comput. 11 (1) (1990) 123–144.

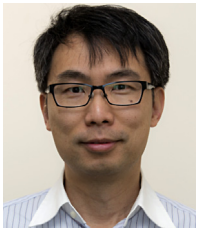
[34] S.W. Sloan, An algorithm for profile and wavefront reduction of sparse matrices, Internat. J. Numer. Methods Engrg. 23 (1986) 239–251.

[35] R.E. Tarjan, An improved algorithm for hierarchical clustering using strong components, Inform. Process. Lett. 17 (1) (1983) 37–41.

[36] H.A. Van der Vorst, A vectorizable variant of some ICCG methods, SIAM J. Sci. Comput. 3 (1982) 350–356.

[37] A.C.N van Duin, Scalable parallel preconditioning with the sparse approximate inverse of triangular matrices, SIAM J. Matrix Anal. Appl. 20 (4) (1999) 987–1006.

[38] M.M. Wolf, M.A. Heroux, E.G. Boman, Factors impacting performance of multithreaded sparse triangular solve, in: High Performance Computing for Computational Science –VECPAR 2010, Vol. 6449, 2010, pp. 32–44.



Edmond Chow is an Associate Professor in the School of Computational Science and Engineering at Georgia Institute of Technology. He previously held positions at D. E. Shaw Research and Lawrence Livermore National Laboratory. His research is in developing numerical methods specialized for high-performance computers, and applying these methods to solve large-scale scientific computing problems. Dr. Chow was awarded the 2009 ACM Gordon Bell prize and the 2002 U.S. Presidential Early Career Award for Scientists and Engineers (PECASE). He serves or has served as Associate Editor for ACM Transactions on Mathematical Software and SIAM Journal on Scientific Computing.

Hartwig Anzt is a research scientist in the Innovative Computing Laboratory at the University of Tennessee. He received his Ph.D. in Applied Mathematics from the Karlsruhe Institute of Technology in 2012. His research interests are in designing numerical methods for high-performance computing on multi- and manycore architectures. He is coauthor of more than 50 journal and conference papers, and contributed to several open source liner algebra packages. Since May 2017, Hartwig Anzt leads a research group on parallel computing at the Karlsruhe Institute of Technology (KIT).

Jennifer Scott leads the Computational Mathematics Group at the Rutherford Appleton Laboratory and is a Science and Technology Facilities Council Individual Merit Research Fellow. She is also Professor of Mathematics at the University of Reading, UK, and Director at Reading of the Centre for Doctoral Training in the Mathematics of Planet Earth. Jennifer received her B.A. (1981) and D.Phil. (1984) from the University of Oxford. Her main research interests are focused on sparse numerical linear algebra. She has developed a number of widely-used software through the HSL mathematical software library.

Jack Dongarra holds an appointment at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers. He was awarded the IEEE Sid Fernbach Award in 2004; in 2008 he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010 he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement; in 2011 he was the recipient of the IEEE IPDPS Charles Babbage Award; and in 2013 he received the ACM/IEEE Ken Kennedy Award. He is a Fellow of the AAAS, ACM, IEEE, and SIAM and a member of the National Academy of Engineering.