Preconditioning Jacobian Systems by Superimposing Diagonal Blocks

M. Ali Rostami[1][0000−0001−6154−4464] and H. Martin Bücker[1,2][0000−0002−5210−0789]

[1] Institute for Computer Science, Friedrich Schiller University Jena, 07737 Jena
[2] Michael Stifel Center Jena for Data-driven and Simulation Science, 07737 Jena

Abstract. Preconditioning constitutes an important building block for the solution of large sparse systems of linear equations. If the coefficient matrix is the Jacobian of some mathematical function given in the form of a computer program, automatic differentiation enables the efficient and accurate evaluation of Jacobian-vector products and transposed Jacobian-vector products in a matrix-free fashion. Standard preconditioning techniques, however, typically require access to individual nonzero elements of the coefficient matrix. These operations are computationally expensive in a matrix-free approach where the coefficient matrix is not explicitly assembled. We propose a novel preconditioning technique that is designed to be used in combination with automatic differentiation. A key element of this technique is the formulation and solution of a graph coloring problem that encodes the rules of partial Jacobian computation that determines only a proper subset of the nonzero elements of the Jacobian matrix. The feasibility of this semi-matrix-free approach is demonstrated on a set of numerical experiments using the automatic differentiation tool ADiMat.

Keywords: Combinatorial scientific computing · Partial Jacobian computation · Partial graph coloring · Sparsity exploitation · ADiMat.

1 Introduction

Large sparse systems of linear equations are critical to computational methods in science, technology, and society. A key characteristic of iterative methods for the solution of such systems is that they can be implemented in a matrix-free fashion [12]. That is, given an N-dimensional right-hand side vector $\mathbf{b}$ and an $N \times N$ nonsingular coefficient matrix J, these methods aim to solve systems of the form

$$J\mathbf{y} = \mathbf{b} \tag{1}$$

by making use of J solely in the form of matrix-vector products, $J\mathbf{z}$, or transposed matrix-vector products, $J^T\mathbf{z}$, where the symbol $\mathbf{z}$ denotes some given N-dimensional vector. Therefore, there is no need to assemble the coefficient matrix in some sparse data storage format. We consider a rather typical situation in computational science where the coefficient matrix J is the Jacobian of

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50417-5_8

some mathematical function given in the form of a computer program. Jacobian-vector products as well as transposed Jacobian-vector products can be efficiently and accurately computed by automatic differentiation (AD) [6, 11] without explicitly setting up the Jacobian matrix. Thus, the major computational kernels of iterative methods match to the functionality that is provided by AD.

In practice, iterative methods involve preconditioning techniques [1, 12] that transform (1) into an equivalent system of the form

$$M^{-1}J\mathbf{y} = M^{-1}\mathbf{b}, \tag{2}$$

whose solution $\mathbf{y}$ is the same as the solution of (1). Here, the $N \times N$ nonsingular matrix M is the preconditioner that is to be constructed such that M is somehow close to J, i.e.,

$$M \approx J.$$

Preconditioning techniques typically need access to individual nonzero elements of the coefficient matrix [1, 12]. However, in a matrix-free approach, such accesses to individual Jacobian entries are computationally expensive, not only in automatic differentiation but also in numerical differentiation.

To bridge the gap between preconditioned iterative methods and AD, we propose a novel approach that is based on superimposing two diagonal block schemes. The first scheme consists of nonoverlapping diagonal blocks of size r that represent a sparsification operation. These blocks are used to define the required nonzero elements [3] of a partial Jacobian computation [9]. The required nonzeros are then determined by AD employing the solution of a suitably defined graph coloring problem [5] that colors a subset of the vertices encoding the rules of partial Jacobian computation.

The second scheme consists of nonoverlapping diagonal blocks of size d that define a simple preconditioner. A standard preconditioning approach is taken that applies ILU decomposition separately on each diagonal block. Here, we deliberately choose $d \geq r$ enabling to incorporate a maximal number of nonrequired nonzero elements outside of the $r \times r$ diagonal blocks of the Jacobian that are produced as by-products of the partial Jacobian computation.

The structure of this article is as follows. In Sect. 2, the overall approach is sketched that consists of a problem arising from scientific computing. It involves the computation of a subset of the nonzero elements of the Jacobian matrix by AD. This partial Jacobian computation problem is then modeled by a suitably defined graph coloring problem in Sect. 3. In Sect. 4 implementation details of the approach are given. Numerical experiments are reported in Sect. 5 and concluding remarks are presented in Sect. 6.

2 Preconditioning via Two Block Schemes

The novel preconditioning approach is inspired by the semi-matrix free preconditioning technique introduced in [3]. The approach in [3] for the solution of (2) is summarized as follows:

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50417-5_8

- Carry out Jacobian-vector products $J\mathbf{z}$ or transposed matrix-vector products $J^T\mathbf{z}$ by applying AD with a seed matrix that is identical to the vector $\mathbf{z}$.
- Choose a block size r and get the sparsified matrix of the Jacobian J denoted by $\rho_r(J)$. Here, the sparsification $\rho_r(J)$ consists of the nonzero elements of the $r \times r$ diagonal blocks of J. Assemble $\rho_r(J)$ via AD and store it explicitly.
- Construct a preconditioner M from $\rho_r(J)$ by performing an ILU(0) decomposition [12] on each block of $\rho_r(J)$. That is, no fill-in elements are allowed during the decomposition.

The nonzero elements of J that are selected by the sparsification $\rho_r(J)$ are called *required* nonzero elements. The symbols used for the block size r and the sparsification $\rho_r(J)$ indicate that these quantities define the $r$equired elements. We also denote the nonzero pattern of the required elements by the set $\mathbf{R}$. As usual for sparsity patterns, we use the binary matrix and the set of the positions of the nonzero elements interchangeably. That is, symbols like $\mathbf{R}$ denoting sparsity patterns are either matrices or sets, depending on the context.

The novel approach borrows the first and the second item of the previous list and replaces the third item by a different preconditioning scheme. The new idea is that AD does not only compute the required elements $\mathbf{R}$, but also certain additional information at no extra computational cost. However, only parts of this additional information is immediately useful for preconditioning. This useful information is called *by-product* and is denoted by the set $\mathbf{B}$. The overall approach is detailed in the remaining part of this section.

Like the previous approach in [3], the new approach is based on computing only a proper subset of the nonzero elements of the Jacobian J, which is referred to as *partial Jacobian computation* [9, 5, 10, 8, 7]. We summarize partial Jacobian computation by considering Fig. 1 taken from [3]. Suppose that we are interested in computing the nonzeros of J on all 2×2 diagonal blocks, but are not interested in the remaining nonzeros. In this example, all nonzeros on the diagonal blocks of size $r = 2$ are the required nonzeros, which are denoted by black disks in the sparsity pattern of the Jacobian depicted in this figure left. All remaining nonzeros of J are called *nonrequired* elements, represented by black circles.

The relative computational cost associated with the forward mode of AD computing the matrix-matrix product $J \cdot S$ is given by the number of columns of the seed matrix S, see [6, 11]. We stress that AD does not assemble the matrix J, but computes the product $J \cdot S$ for a given S directly. The symbol $\mathrm{cp}(J) := J \cdot S$ represents this so-called *compressed Jacobian matrix*.

The exploitation of sparsity has a long tradition in AD; see the survey [5]. The main idea behind sparsity exploitation is to form groups of columns of J. This grouping is denoted by colors in the middle of the Fig. 1. If J is an $N \times N$ matrix, all (zero and nonzero) elements of J are computed by setting the seed matrix to the identity of order N. The relative computational cost of this approach is then the number of columns of the identity given by N. However, exploiting the grouping of columns it is possible to find a seed matrix with fewer than N columns. In the middle of Fig. 1, there are three colors representing three groups of columns. Each group of columns in J corresponds to a single column in the

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50417-5_8

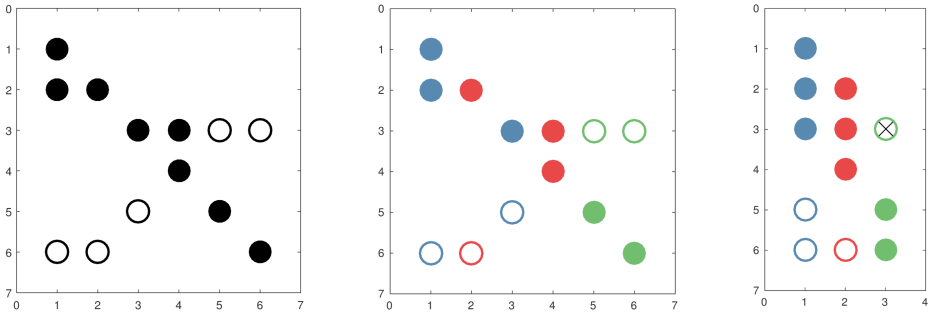

Fig. 1. Sparsity patterns of a 6×6 Jacobian J shown left and its compressed version $\mathrm{cp}(J)$ shown right. Grouping of columns of J is denoted by colors in the middle. (Figure taken from [3].)

compressed Jacobian matrix depicted in the right. More precisely, a column of $\mathrm{cp}(J)$ with a certain color c is the linear combination of those columns of J that belong to the group of columns with the color c. Equivalently, there is a binary seed matrix S whose number of columns corresponds to the number of colors such that all required nonzero elements $\mathbf{R}$ of J also appear in $\mathrm{cp}(J)$.

In the semi-matrix-free approach [3], given the sparsity pattern of J and the set of required elements $\mathbf{R}$, the problem of assembling the required nonzero elements with a minimal relative computational cost is as follows.

Problem 1 (Block Seed). Let J be a sparse $N \times N$ Jacobian matrix with known sparsity pattern and let $\rho_r(J)$ denote its sparsification using $r \times r$ blocks on the diagonal of J. Find a binary $N \times p$ seed matrix S with a minimal number of columns, p, such that all nonzero entries of $\rho_r(J)$ also appear in the compressed matrix $\mathrm{cp}(J) := J \cdot S$.

The compressed Jacobian $\mathrm{cp}(J)$ contains by definition all required elements of J. However, by inspecting the example in Fig. 1, it also contains additional nonzero elements. These additional nonzero elements decompose into two different classes. There are nonzero elements of $\mathrm{cp}(J)$ that are nonrequired elements of J. In the example, the three nonzeros at the positions $(5,1)$, $(6,1)$ and $(6,2)$ belong to this class. The other class of nonzero elements of $\mathrm{cp}(J)$ consists of those nonzeros that are linear combinations of nonzero entries of J. For instance, the nonzero at the position $(3,3)$ in $\mathrm{cp}(J)$ is the sum of $J(3,5)$ and $J(3,6)$.

The overall idea of the novel approach is to incorporate into the preconditioning not only the required elements of J, but also a certain subset of the nonzero elements of $\mathrm{cp}(J)$ that are nonrequired elements of J. To this end, another sparsification operator $\rho_d(\cdot)$ is introduced that extracts from $\mathrm{cp}(J)$ the nonzero elements of the $d \times d$ diagonal blocks of J that are not required. The set of by-products $\mathbf{B}$ is then defined as those nonzero elements of the compressed Jacobian $\mathrm{cp}(J)$ that are nonzeros within these $d \times d$ blocks of J and that are not contained in the set of required elements $\mathbf{R}$. In other words, the by-products $\mathbf{B}$ are obtained from the compressed Jacobian $\mathrm{cp}(J)$ by removing all entries that

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50417-5_8

are linear combinations of nonzeros of J and by additionally removing all (required and nonrequired) nonzeros of J that are outside the $d \times d$ diagonal blocks. The preconditioner M that approximates J is then constructed by assembling the nonzeros $\mathbf{R} \cup \mathbf{B}$ in a matrix denoted as $\mathrm{rc}(J)$ and using an ILU decomposition on the $d \times d$ diagonal blocks. The symbols used for the block size d and the sparsification operator $\rho_d(\cdot)$ indicate that these quantities are used to carry out a $d$ecomposition on each block.

We remark that the sparsification operators, $\rho_r(\cdot)$ and $\rho_d(\cdot)$, that extract the diagonal blocks reduce the size of the bottom right block accordingly if the order of the matrix is not a multiple of the block size. For instance, returning to the example in Fig. 1 with $r = 2$ and assuming that $d = 5$, then the operator $\rho_d(\cdot)$ leads to a top left 5×5 block and a bottom right 1×1 block. The set of by-products $\mathbf{B}$ then consists of the single nonzero entry $J(5, 3)$ which is stored in $\mathrm{cp}(J)$ at position (5,1).

In summary, a high-level description of the new preconditioning approach that uses two diagonal block schemes of size r and of size d is given as follows:

- Carry out Jacobian-vector products $J\mathbf{z}$ or transposed matrix-vector products $J^T\mathbf{z}$ using AD.
- Choose a block size r, solve Problem 1, and compute $\mathrm{cp}(J)$ using AD.
- Choose a block size d and assemble the required elements $\mathbf{R}$ as well as the by-products $\mathbf{B}$ from $\mathrm{cp}(J)$ using the sparsification operator $\rho_d(\cdot)$. Store $\mathbf{R} \cup \mathbf{B}$ explicitly in a matrix $\mathrm{rc}(J)$.
- Construct a preconditioner M from $\mathbf{R} \cup \mathbf{B}$ by performing an ILU decomposition on each diagonal $d \times d$ block of $\mathrm{rc}(J)$.

The only other work that is related to our approach is the preconditioning technique introduced in [4], which is also based on partial matrix computation, but differs in formulating balancing problems.

The purpose of the following section is to reformulate the combinatorial problem from scientific computing given by Problem 1 in terms of an equivalent graph coloring problem.

3 Modeling via Partial Graph Coloring

Recall from the previous section that the exploitation of sparsity is a well-studied topic in derivative computations [5]. Interpreting these scientific computing problems in the language of graph theory does not only give us a better insight to the abstract problem structure but also offers an intimate connection to the rich history of research in graph theory that can lead to efficient algorithms for the solution of the resulting problems. In this section, we consider the graph problem corresponding to the scientific computing problem that was introduced in the previous section.

In the spirit of [3], we define a combinatorial model that handles the decomposition of the nonzero elements of J into two sets called required and nonrequired elements. The following new definition introduces the concept of structurally ρ_r-orthogonal columns.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50417-5_8

Definition 1 (Structurally ρ_r-Orthogonal). *A column $J(:,i)$ is structurally ρ_r-orthogonal to column $J(:,j)$ if and only if there is no row position ℓ in which $J(\ell,i)$ and $J(\ell,j)$ are nonzero elements and at least one of them belongs to the set of required element $\rho_r(J)$.*

Next, we define the ρ_r-column intersection graph which will be used to reformulate Problem 1 arising from scientific computing.

Definition 2 (ρ_r-Column Intersection Graph). *The ρ_r-column intersection graph $G_{\rho_r} = (V, E_{\rho_r})$ associated with a pair of $N \times N$ Jacobians J and $\rho_r(J)$ consists of a set of vertices $V = \{v_1, v_2, \ldots, v_N\}$ whose vertex v_i represents the ith column $J(:,i)$. Furthermore, there is an edge (v_i, v_j) in the set of edges E_{ρ_r} if and only if the columns $J(:,i)$ and $J(:,j)$ represented by v_i and v_j are not structurally ρ_r-orthogonal.*

That is, the edge set E_{ρ_r} is constructed in such a way that columns represented by two vertices v_i and v_j need to be assigned to different column groups if and only if $(v_i, v_j) \in E_{\rho_r}$.

Using this graph model, Problem 1 from scientific computing is transformed into the following equivalent graph theoretical problem.

Problem 2 (Minimum Block Coloring). Find a coloring of the ρ_r-column intersection graph G_{ρ_r} with a minimal number of colors.

The solution of this graph coloring problem corresponds to a seed matrix S which is then used to compute the compressed Jacobian $\mathrm{cp}(J) = J \cdot S$ using AD. Recall from the previous section that the required elements of J are contained in $\mathrm{cp}(J)$. However, we already pointed out that some additional useful information $\mathbf{B}$ is also contained in $\mathrm{cp}(J)$. In the following section, we discuss how to recover these by-products $\mathbf{B}$ from $\mathrm{cp}(J)$ and how to use it for preconditioning.

4 Implementation Details

Given the sparsity pattern $\mathbf{P}$ of the Jacobian matrix J, the following pseudocode summarizes the new preconditioning approach:

1: $\mathbf{R} = \rho_r(\mathbf{P})$
2: $S = \mathrm{partial\_coloring}(\mathbf{P}, \mathbf{R})$
3: Compute $\mathrm{cp}(J) = J \cdot S$ by AD
4: $\mathrm{rc}(J) = \rho_d(\mathrm{partial\_recover}(\mathbf{P}, S, \mathrm{cp}(J), \mathbf{R}))$
5: Construct M as the ILU decomposition of $\mathrm{rc}(J)$
6: Solve the preconditioned linear system (2)

In this pseudocode, we first compute the required elements $\mathbf{R}$ using the sparsification operator $\rho_r(\cdot)$. The required elements $\mathbf{R}$ are taken as an input to solve Problem 2 using a partial graph coloring algorithm [3]. The solution of this graph coloring problem corresponds to a seed matrix S that is used by the AD tool ADiMat [2, 14] to compute the compressed Jacobian $\mathrm{cp}(J)$. Then, we need a function `partial_recover()` to recover the nonzero elements $\mathbf{R} \cup \mathbf{B}$ of J

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50417-5_8

from the compressed Jacobian $\mathrm{cp}(J)$. The preconditioner M is constructed by a blockwise ILU decomposition of $\mathrm{rc}(J)$ and the preconditioned system is solved by Jacobian-vector products using ADiMat.

To introduce the function `partial_recover()`, it is convenient to consider the standard approach of recovering the nonzeros of a Jacobian from its compressed version [6]. The standard approach assumes that all nonzeros of a sparse Jacobian are to be determined, whereas in a partial Jacobian approach we are interested in a subset of the nonzeros. In the standard approach, the nonzeros are recovered using the following MATLAB-like pseudocode:

```
1: procedure RECOVER(P, S, cp(J))
2:     J = zeros(size(P))
3:     for i = 1 : size(cp(J), 1) do
4:         I = P(i, :) ~= 0
5:         S_I = S(I, :)
6:         [row, col] = find(S_I)
7:         [rs, perm] = sort(row)
8:         J(i, I) = cp(J)(i, col(perm))
```

Given the pattern $\mathbf{P}$ of a sparse Jacobian J, the seed matrix S, and the compressed Jacobian $\mathrm{cp}(J) = J \cdot S$, this procedure recovers the Jacobian matrix J. It reconstructs every row i of J step by step. In each step, it first computes the indices I of the nonzeros of the row i of J. Then, it considers a reduced seed matrix $S_I = S(I, :)$. Here, S_I is a matrix containing those rows of S that correspond to the nonzeros of J in the row i. Suppose that there is a nonzero element in J in position (i, k). We then need the column index of the entry 1 in the row k of the reduced seed matrix. With this column index, the corresponding nonzero is extracted from $\mathrm{cp}(J)$. Because of MATLAB's implementation of `find()`, the row indices in *row* have to be sorted in increasing order.

For partial Jacobian computation where only a subset of nonzeros is determined, we need to extend the previous procedure to recover the Jacobian matrix using the seed matrix which is computed by the partial coloring. The following pseudocode introduces the new procedure `partial_recover()` which computes the Jacobian J from its compressed version $\mathrm{cp}(J)$ in partial Jacobian computation. Compared to the previous procedure `recover()`, this procedure needs the pattern of the required elements $\mathbf{R}$ as an additional input.

```
1: procedure PARTIAL_RECOVER(P, S, cp(J), R)
2:     NR = P − R
3:     J = zeros(size(P))
4:     for i = 1 : size(cp(J), 1) do
5:         I = P(i, :) ~= 0
6:         S_I = S(I, :)
7:         [row, col] = find(S_I)
8:         [rs, perm] = sort(row)
9:         J(i, I) = cp(J)(i, col(perm))
10:        colS = ones(1, size(S(I, :), 1)) · S(I, :)
11:        positions = find(colS > 1)
```

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50417-5_8

12: **if** $\sim$ isempty($positions$) **then**
13: **for** $p = 1 :$ length($positions$) **do**
14: $r_i =$ find($S(:, positions(p))$)
15: **if** sum($\mathbf{NR}(i, r_i)$) $\sim= 1$ **then**
16: $J(i, r_i) = 0$

The first steps up to the step 9 of this procedure are similar to the previous procedure `recover()`. The new procedure, however, needs to take into account the nonrequired elements. So, it looks for the columns of S_I which have more than one nonzero (in steps 10 and 11) since there are the columns in which the addition of two nonrequired elements can happen. Then, it goes through all of those columns, if any, and checks if any nonrequired elements is added. If such an addition happens in a column r_i, we put a zero in the corresponding entry $J(i, r_i)$ in the recovered Jacobian. The variable **NR** in this algorithm contains the positions of the nonrequired elements in $\mathbf{P}$.

After recovering the Jacobian matrix J via the procedure `partial_recover()`, we need to make sure that only those elements will remain that are inside the diagonal blocks of size d. That is, we need to compute the by-products $\mathbf{B}$ using the sparsification operator $\rho_d(\cdot)$ which, in the current implementation, is carried out outside of the procedure `partial_recover()`; see also the sparsification operator $\rho_d(\cdot)$ in the algorithm sketched at the beginning of this section.

5 Numerical Experiments

Here, we employ the semi-matrix-free approach for the solution of a system of linear equations of the form (2). This system arises in the solution of an optimal boundary control problem for radiative transfer. Throughout the following experiments, the resulting coefficient matrix has the order $N = 1,944$ and contains $49,856$ nonzero elements. Its nonzero pattern is depicted in the left of Fig. 2. In the middle of this figure, the pattern of the sparsification $\rho_r(J)$ is depicted for a block size of $r = 100$. The $\lceil N/r \rceil = 20$ blocks are visible and are highlighted using a gray background. Notice that the last block is considerably smaller than the remaining blocks. To illustrate the preconditioning approach, the pattern of $\mathrm{rc}(J)$ is also plotted for a block size of $d = 500$ in the right of this figure.

The present experiments are carried out in MATLAB, R2019b. All derivative computations are computed by ADiMat. The right-hand side $\mathbf{b}$ of the linear system is chosen as the sum of all columns of J such that the exact solution $\mathbf{y}$ to (2) is given by the vector containing ones in all positions.

The linear system is solved using the Generalized Minimal RESidual method (GMRES) [13] with restart parameter of 20. We always take $\mathbf{y}_0 = \mathbf{0}$ as the initial guess. For the unpreconditioned system, the iteration is stopped in the nth step if

$$||\mathbf{b} - J\mathbf{y}_n||_2 / ||\mathbf{b}||_2 \leq \varepsilon. \tag{3}$$

For the preconditioned system, convergence is obtained if

$$||M^{-1}(\mathbf{b} - J\mathbf{y}_n)||_2 / ||M^{-1}\mathbf{b}||_2 \leq \varepsilon. \tag{4}$$

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50417-5_8

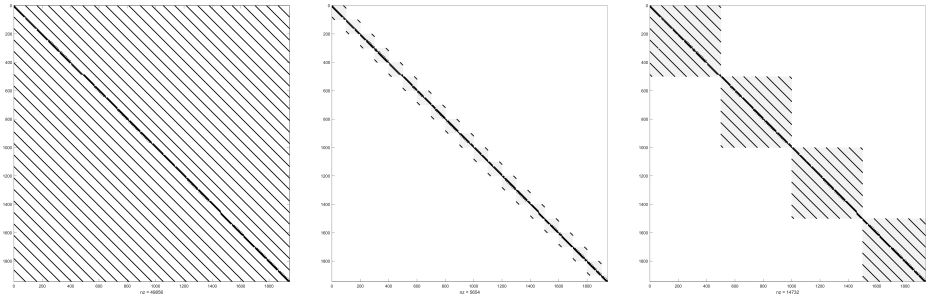

Fig. 2. Left: The nonzero pattern of the Jacobian J corresponding to the linear system (1) with the problem size $N = 1,944$. Middle: The pattern of the sparsified Jacobian $\rho_r(J)$ illustrating the required elements for a block size of $r = 100$. Right: The pattern of the required and by-product elements $\mathrm{rc}(J)$ for a block size of $d = 500$.

The tolerance $\varepsilon = 10^{-13}$ is chosen for both cases. All tests are carried out on an Intel Core i7-8550U CPU with a clock rate of 1.80GHz and 16GB RAM.

In Fig. 3, the convergence behavior using GMRES is plotted versus the number of matrix-vector products. The convergence is monitored by the residual vector of the nth iteration defined by $\mathbf{r}_n = \mathbf{b} - J\mathbf{y}_n$. More precisely, we show the norm of the residual scaled by the initial residual norm $||\mathbf{r}_0||_2$. We do not report the convergence versus the number of iterations for two reasons. Firstly, the number of matrix-vector products is known to be a better indication of the computing time than the number of iterations [12]; secondly, the number of matrix-vector products directly corresponds to the number of colors and thus makes it easy to relate the convergence to the cost of computing $\mathrm{cp}(J)$ that is once needed to set up the preconditioner. This aspect is crucial in applications such as Newton-like methods for nonlinear systems where a sequence of linear systems with the same Jacobian sparsity pattern arises and the cost of solving a single coloring problem is amortized over solving multiple linear systems.

On the other hand, the number of matrix-vector products is only an approximation of the computing time, in particular for GMRES without restarts, where the number of operations carried out in an iteration linearly increases with the iteration number. In the first set of experiments, where the block size for the sparsification operator $\rho_d(\cdot)$ is fixed to $d = 500$, the computing time needed to converge the preconditioned iteration is always smaller than for the unpreconditioned method, if the time for partial coloring and computing $\mathrm{cp}(J)$ is neglected. Taking this time into account so that the complete process of setting up the preconditioner is included, the preconditioned method is faster than the unpreconditioned method for all experiments where $r > 10$.

The unpreconditioned method exhibits the slowest convergence using the largest number of matrix-vector products needed to converge to the desired tolerance. This figure also contains six additional graphs by varying the block size $r = 4$, $r = 20$ and $r = 100$ and by employing two different preconditioning approaches. The approach advocated in this article is based on the blockwise

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50417-5_8

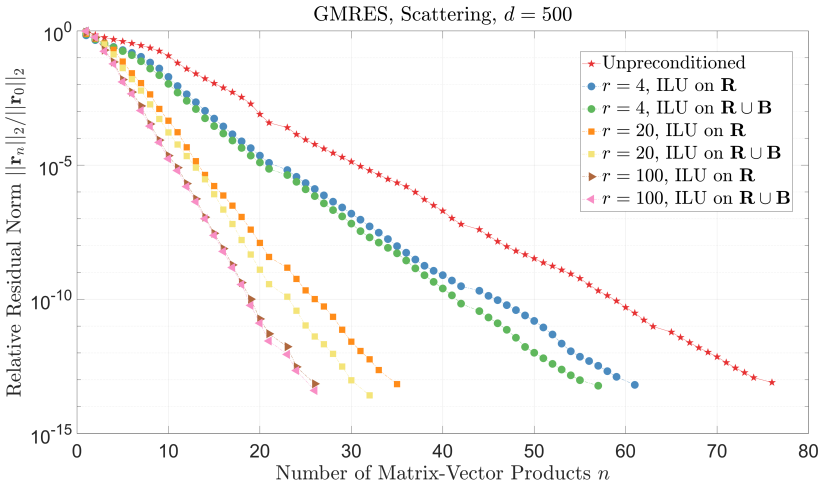

Fig. 3. Convergence behavior of GMRES.

ILU(0) decomposition of $\mathrm{rc}(J)$, the matrix that contains the nonrequired elements as well as the by-products. This approach is denoted by $\mathbf{R} \cup \mathbf{B}$. For the sake of comparison, we also investigate another approach that is identical to the previously mentioned approach with a single exception. Rather than using the information $\mathbf{R} \cup \mathbf{B}$ to construct the blockwise ILU(0) preconditioner, the diagonal blocks involve only the information $\mathbf{R}$. That is, the by-products $\mathbf{B}$ are discarded from the preconditioning process. This latter approach is similar to our previous work reported in [3]. However, in [3], we do not use two different block schemes with different block sizes.

For the three block size $r = 4$, $r = 20$ and $r = 100$, the two preconditioning techniques based on $\mathbf{R} \cup \mathbf{B}$ and $\mathbf{R}$ both converge faster than the unpreconditioned method. This statement is true for GMRES as well as for other Krylov solvers that we tested but whose results are omitted due to the lack of space. It is also interesting that the convergence is improved by increasing the block sizes from $r = 4$ via $r = 20$ up to $r = 100$. Furthermore, keeping the block size r fixed, the convergence of the approach $\mathbf{R} \cup \mathbf{B}$ tends to be faster than the approach using only $\mathbf{R}$. This observation is valid for the two block sizes $r = 4$ and $r = 20$. For large block sizes, however, it is unlikely that there will be a large set of by-products $\mathbf{B}$. So, the differences in the convergence behavior between an approach using $\mathbf{R} \cup \mathbf{B}$ and an approach using $\mathbf{R}$ tend to be small.

To better understand the preconditioning approach, we now focus on the number of nonzero elements when increasing the block size r. Figure 4 illustrates the number of required elements, $|\mathbf{R}|$, using black bars as well as the number of by-products, $|\mathbf{B}|$, using dark gray bars. The vertical axis (ordinate) is scaled to the number of nonzeros in J given by $49,856$. That is, the light gray bars denote the number of nonzero elements of J that are not taken into account when the preconditioner is constructed. For a block size of $d = 500$, this diagram shows

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50417-5_8

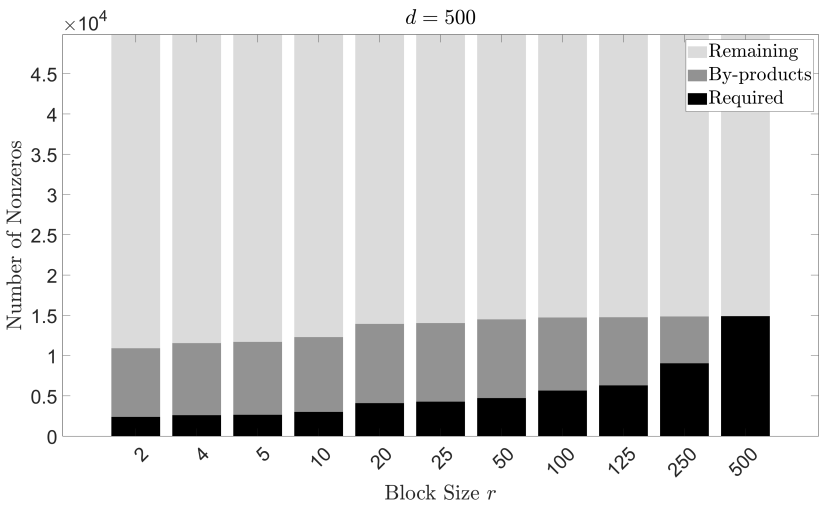

Fig. 4. Number of nonzeros varying the block size r for a fixed block size $d = 500$.

that the number of required elements increase only mildly when increasing the block size r up to moderate values. However, when increasing r significantly, there is also a corresponding increase in the number of required elements; see the block sizes at the right of this figure.

The sum $|\mathbf{R}| + |\mathbf{B}|$ is rather constant when increasing the block size r. So, the approach tends to be relevant in particular for small block sizes r where the number of by-products is comparatively large. In this situation, the information available in $\mathbf{B}$ is particularly attractive since it comes from the partial Jacobian computation without any extra computational cost.

Next, we consider the number of colors needed for the solution of the partial graph coloring problem that is formally specified by Problem 2. This number of colors is depicted in Fig. 5. Here, the block size r is varied in the same range as in Fig. 4. Since the number of colors is an estimate for the relative computational cost to compute the compressed Jacobian $\mathrm{cp}(J) = J \cdot S$ using AD, a slight increase in the number of colors can be harmful. This figure illustrates that the number of colors increases with the block size. Once more, this is an indication that the preconditioning approach is particularly relevant for small block sizes. Also, for small block sizes the storage requirement tends to be lower than for larger block sizes which corresponds to the overall setting in which a sparse data structure for the Jacobian is assumed to exceed the available storage capacity.

Finally, we analyze the number of nonzeros and the number of colors not only for a varying block size r, but also when varying the block size d. In Fig. 6, the results are depicted for the three block sizes $d = 300$, $d = 400$ and $d = 500$. For each value of d, this set of experiments involves those block sizes r that are divisors of d. The legend contains the union of all divisors of the three block sizes d. In the layout of this figure, a number of nonzeros is indicated by a

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50417-5_8

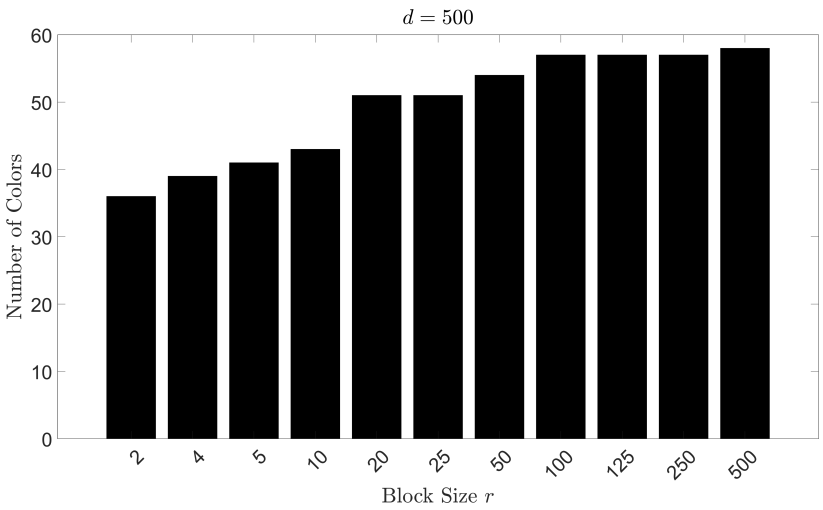

Fig. 5. Number of colors varying the block size r.

bar to which the left ordinate is associated. A number of colors is denoted by a disk whose value is specified on the right ordinate. As in Fig. 4, black bars here denote the number of required nonzeros. In contrast, the by-products are now given by bars whose colors correspond to the values of r. The results show a similar behavior for all three values of d. The number of required nonzeros increase with increasing r. Compared to the number of required nonzeros, the number of by-products is in a similar magnitude, except when r approaches d. (By construction, there cannot be any by-product for $r = d$.) The number of by-products tends to by reasonably large, even for small values of r. At the same time, the number of colors is small for small block sizes r. In other words, small block sizes r are not only attractive because (i) they deliver additional information (represented by nonzero elements) that is useful for preconditioning without any extra computational cost and, at the same time, (ii) they lead to a low relative computational cost associated with AD (represented by colors).

6 Concluding Remarks

While matrix-free iterative methods and (transposed) Jacobian-vector products computed by automatic differentiation match well to each other, today, there is still a gap between preconditioning and automatic differentiation. The reason is that, in a matrix-free approach, accesses to individual nonzero entries of the Jacobian coefficient matrix which are needed by standard preconditioning techniques are computationally expensive. This statement holds not only for automatic differentiation but also for numerical differentiation.

The major new contribution of this article is a semi-matrix-free preconditioning approach that uses two separate diagonal block schemes partitioning the

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50417-5_8

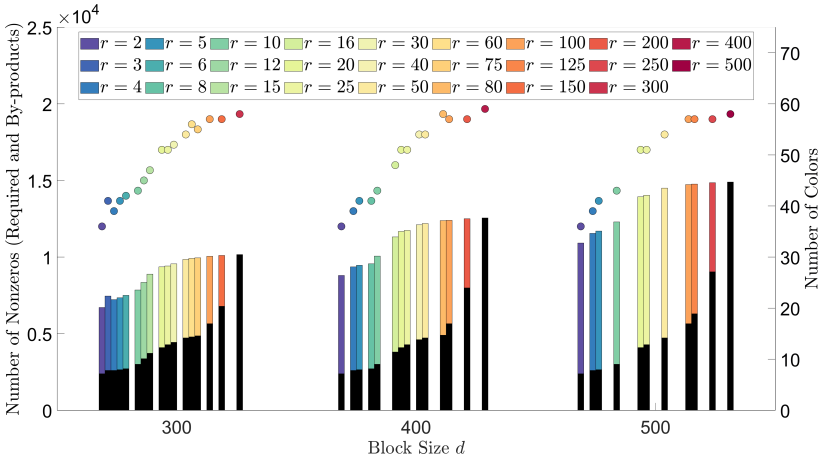

Fig. 6. Number of required nonzeros (black bars) and by-products (colored bars) on the left axis and the number of colors (colored disks) on the right axis varying the two block sizes r (color) and d (groups of bars and disks).

coefficient matrix into smaller submatrices. In both schemes, the diagonal blocks do not overlap. The first scheme employs blocks that define the required nonzero elements of a partial Jacobian computation. This scheme is relevant for minimizing the relative computational cost of the partial Jacobian computation. The resulting minimization problem is equivalent to a partial graph coloring problem. The second scheme is based on blocks whose sizes are larger than those of the first scheme. The blocks of this second scheme define the positions from which by-products of the partial Jacobian computation are extracted. Together with the required nonzero elements these by-products are used to construct a preconditioner that applies ILU decompositions to each of these blocks. Numerical experiments using the automatic differentiation tool ADiMat are reported demonstrating the feasibility of the new preconditioning technique.

There is room for further investigations that aim at bridging the gap between preconditioning and automatic differentiation. For instance, it is interesting to study more advanced preconditioning techniques and analyze to what extent they are capable of exploiting the information available in the by-products of the partial Jacobian computation.

Acknowledgements

Several computational experiments were performed on resources of Friedrich Schiller University Jena supported in part by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – INST 275/334–1 FUGG; INST 275/363–1 FUGG. These resources were additionally supported by Freistaat Thüringen grant 2017 FGI 0031 co-funded by the European Union in the framework of Europäische Fonds für regionale Entwicklung (EFRE).

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50417-5_8

References

1. Benzi, M.: Preconditioning techniques for large linear systems: A survey. Journal of Computational Physics **182**(2), 418–477 (2002). https://doi.org/10.1006/jcph.2002.7176
2. Bischof, C.H., Bücker, H.M., Lang, B., Rasch, A., Vehreschild, A.: Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In: Proc. 2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002). pp. 65–72. IEEE Computer Society, Los Alamitos, CA, USA (2002). https://doi.org/10.1109/SCAM.2002.1134106
3. Bücker, H.M., Lülfesmann, M., Rostami, M.A.: Enabling implicit time integration for compressible flows by partial coloring: A case study of a semi-matrix-free preconditioning technique. In: 2016 Proc. 7th SIAM Workshop on Combinatorial Scientific Computing. pp. 23–32. SIAM, Philadelphia, PA, USA (2016). https://doi.org/10.1137/1.9781611974690.ch3
4. Cullum, K.J., Tůma, M.: Matrix-free preconditioning using partial matrix estimation. BIT Numerical Mathematics **46**(4), 711–729 (2006). https://doi.org/10.1007/s10543-006-0094-8
5. Gebremedhin, A.H., Manne, F., Pothen, A.: What color is your Jacobian? Graph coloring for computing derivatives. SIAM Review **47**(4), 629–705 (2005). https://doi.org/10.1137/S0036144504444711
6. Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 105 in Other Titles in App. Math., SIAM, Philadelphia, PA, USA, 2nd edn. (2008). https://doi.org/10.1137/1.9780898717761
7. Lülfesmann, M.: Partielle Berechnung von Jacobi-Matrizen mittels Graphfärbung. In: Informatiktage 2007, Fachwissenschaftlicher Informatik-Kongress. Lecture Notes in Informatics – Seminars, vol. S-5, pp. 21–24. Gesellschaft für Informatik e.V. (2007), http://dl.gi.de/handle/20.500.12116/4920
8. Lülfesmann, M.: Graphfärbung zur Berechnung benötigter Matrixelemente. Informatik-Spektrum **31**(1), 50–54 (2008). https://doi.org/10.1007/s00287-007-0199-8
9. Lülfesmann, M.: Full and partial Jacobian computation via graph coloring: Algorithms and applications. Dissertation, Dept. Computer Science, RWTH Aachen University (2012), https://d-nb.info/1023979144/34
10. Petera, M., Lülfesmann, M., Bücker, H.M.: Partial Jacobian computation in the domain-specific program transformation system ADiCape. In: Proc. Internat. Multiconf. on Computer Science and Information Technology. vol. 4, pp. 595–599. IEEE Computer Society, Los Alamitos, CA, USA (2009). https://doi.org/10.1109/IMCSIT.2009.5352778
11. Rall, L.B.: Automatic Differentiation: Techniques and Applications, Lecture Notes in Computer Science, vol. 120. Springer, Berlin (1981). https://doi.org/10.1007/3-540-10861-0
12. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM, Philadelphia, 2nd edn. (2003). https://doi.org/10.1137/1.9780898718003
13. Saad, Y., Schultz, M.H.: GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM Journal on Scientific and Statistical Computing **7**(3), 856–869 (1986). https://doi.org/10.1137/0907058
14. Willkomm, J., Bischof, C.H., Bücker, H.M.: A new user interface for ADiMat: Toward accurate and efficient derivatives of Matlab programs with ease of use. International Journal of Computational Science and Engineering **9**(5/6), 408–415 (2014). https://doi.org/10.1504/IJCSE.2014.064526

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50417-5_8