KØBENHAVNS UNIVERSITET

Parallel Implementation of the SPAI algorithm

Caroline Amalie Kierkegaard and Mikkel Willén

May 19, 2023

Abstract

Contents

1 Introduction

For centuries mathematicians have been faced with the challenge of solving systems of linear equations, and without calculators and computers solving larger systems proved difficult. Directly finding the exact inverse A^{-1} of a large and sparse matrix A would back then be infeasible, and nowadays extremely expensive in terms of work and space. In this thesis we will consider an iterative method for constructing a sparse approximation to the inverse of such a matrix A, where the sparsity is preserved in the inverse. Due to the preservation of the sparsity pattern, the work will be proportional with the amount of nonzero elements and not the size of the matrix. This technique can be effectively used in many applications of numerical analysis, e.g. for preconditioning of a linear system.

The technique of preconditioning that have been investigated since the 1970's and lead to great results. Among these are Grote and Huckle's sparse approximate inverse preconditioner (SPAI) based on the Frobenius norm minimisation (Grote 1995). SPAI is an effective preconditioner, because it is able to update a given sparsity structure automatically while maintaining the sparsity for each iteration. Since it splits up each computation into an independent least squares problem for each column, it is inherently parallel. Inherently parallel preconditioners become of increasing importance, as new parallel clusters with millions of cores come into existence and the paradigm of parallel programming develops (Selacek 2012). A great advantage is the time and space complexity, which will for a sparse matrix be dependent on the number of nonzero elements instead of the size of the matrix.

To find the exact inverse of a dense matrix, we augment it with the identity matrix and perform row operations trying to make the identity matrix to the left. This will result in the inverse to the right. Since this method will be extremely laborious for a large, sparse matrix, the method of finding the approximate inverse is very relevant. Developing on the initial SPAI algorithm by Grote and Huckle, several theses have been dedicated to improving the sparse approximate inverse preconditioner.

Among these are the dissertation *Modified Sparse Approximate Inverses (MSPAI) for Parallel Preconditioning* by Kallischko (Kallischko 2007), which introduces variants of the sparse approximate inverse technique; the modified SPAI (MSPAI) algorithm and the factorised SPAI (FSPAI) algorithm. The dissertation *Sparse Approximate Inverses for Preconditioning, Smoothing, and Regularization* by Sedlacek improves on the SPAI algorithm and its variants. Both dissertations present solutions to problems we have encountered in our implementation, which we will discuss in the discussion section.

The reason for the continuing efforts in developing the SPAI algorithm, is the relevant real worlds applications. An often used method in calculus, is the Newton method, which finds the roots of a differential function F, which is the solutions to the equation $F = 0$. In order to do so, we have to find the inverse of a Hessian matrix. This is were the SPAI preconditioner becomes highly relevant, because computing the exact inverse of the Hessian in high dimensions can be very expensive. Determining the optimal solutions inexpensively with the Newton method is essential in different areas, e.g. statistics, applied mathematics, numerical analysis, economics and finance (McAleer 2019).

In this thesis we will implement the SPAI algorithm in CUDA both sequentially and parallel.

Section 2 will provide the theoretical basis of the work and fulfill the theoretical learning goals. We will account for preconditioning and condition numbers. We will explain the compressed sparse column (CSC) format used in the implementation.

Section 3 is devoted to the sparse approximate preconditioner (SPAI) by Grote and Huckle and will elaborate on sparse approximate inverses and the algorithm for determining such. We will describe and explain efficient parallel implementations aimed at GPU execution for the SPAI preconditioner. We will methodically go trough the algorithm and explain in depth the most relevant parts. We will explain the Frobenius norm minimisation, how to compute the sparsity structure $\mathcal{J}$ and $\mathcal{I}$ and how to apply the Householder QR decomposition. We will describe how to improve upon M by updating the sparsity structure and the QR decomposition. We will show our pseudocode, used as a foundation for our implementation. We will discuss the time complexity of the algorithm.

Section 4 concerns our implementations of the SPAI algorithm. We have implemented a sequential prototype in Python to use as a basis for our tests. In CUDA we have implemented the algorithm both sequential and parallel. We will explain the choices we have made in our implementation and highlight relevant code snippets.

Section 5 shows and analyses our test results of both the accuracy and time efficiency of respectively the sequential and parallel implementations. We have performed experiments...

Section 6 compares the sequential and parallel implementations and analyses the effectiveness of the parallel algorithms compared to a non-parallel implementation. We will also compare both accuracy and speed to other implementations such as our own Python-prototype and cuSolver??. RESULTS. We will discuss the advantages of parallelism. RESULTS. This section will also discuss how we could further improve our implementation and develop further on the algorithm.

Section 7 is the conclusion.

2 Theory

This section will provide the theoretical basis for the sparse approximate inverse preconditioner and the implementation of it.

2.1 Preconditioning

Preconditioning is a technique used for enhancing convergence speed. Many iterative methods converge very slow for not preconditioned systems. The goal of preconditioning is to modify the system in such a way that an iterative method converges significantly faster.

The concept of preconditioning is theoretically straightforward, but practical implementations require careful consideration of various technical details. When constructing a preconditioner M, we must find a balance between reducing the number of iterations required and not making the preconditioner too computationally expensive to construct. A well chosen preconditioner must be

- relatively cheap to construct
- not use too much memory
- improve the condition number of A
- be parallelisable when both constructed and implemented
- and performing linear solves with the preconditioner should be cheap (Geffen 2013)

2.2 Sparse matrices

Matrices can have specific structures or hold special properties. Some matrices only has a small amount of nonzeros compared to their dimension, and these are called sparse matrices. There is not a strict definition of a sparse matrix, but a general perception is that a matrix will be called sparse if the storage of only the nonzero elements and the application of an algorithm adapted to sparse matrices lead to a computational advantages over a dense matrix (Kallischko, page 13).

Sparse matrices enables us to construct methods which profits from the sparse structure in terms of fast algorithms and efficient storage.

2.3 Sparse Compressed Column format

When storing a sparse matrix, we do not want to waste space storing the great amount of zeros. We can store the nonzeros either row- or column-wise. Since the SPAI algorithm is computed column-wise, we want fast access to the columns and therefore we would like to store our data in a column format. The format we will use in this paper is called Sparse Compressed Columnn (CSC) format.

In the format, the matrix is stores in three arrays. The first array `offset` contains the accumulated number of nonzeros elements in each row. The `flatData` contains the nonzero entries ordered after column. The `flatRowIndex` contains the row indices of the nonzero entries also ordered after column. Here is an example of the CSC storage format:
The compressed storage saves communication between processing elements i parallel environments, because we only have to exchange the nonzero elements (Sedlacek 2012, page 18).

$$A = \begin{pmatrix} 20 & 0 & 0 \\ 0 & 30 & 10 \\ 0 & 0 & 0 \\ 0 & 40 & 0 \end{pmatrix} \xrightarrow{\text{CSC format}} \begin{array}{lcl} \texttt{offset} & = & \{0,1,3,4\} \\ \texttt{flatData} & = & \{20,30,40,10\} \\ \texttt{flatRowIndex} & = & \{0,1,3,1\} \end{array}$$

Figure 1: Example of the CSC storage format

3 The SPAI Algorithm

In 1995, Grote and Huckle published *Parallel preconditioning with Sparse Approximate Inverses* (reference), which presented an algorithm designed to find the approximate inverse of a sparse matrix by minimising the Frobenius norm. This section will go trough the SPAI algorithm and explain the algorithm in depth.

3.1 Frobenius norm minimisation

We can compute sparse approximate inverses $M \approx A^{-1}$ via Frobenius norm minimisation. The main idea is to build a sparse matrix from a matrix A, so that the matrix-vector products do not become too expensive. The minimisation

$$\min_M \|AM - I\|_F^2 \tag{1}$$

with $A \in \mathbb{R}^{n\times n}$ and the unity matrix I with the same dimensions leads to the sparse approximate inverse M.
We can split the minimisation up into a sum of Euclidean norms, like this

$$\min_M \|AM - I\|_F^2 = \sum_{k=1}^{n} \|(AM - I)e_k\|_2^2 \tag{2}$$

$$= \sum_{k=1}^{n} \|Am_k - e_k\|_2^2 \tag{3}$$

where m_k and e_k denotes the k'th column of respectively M and I. Since each summand in (ref) represents a least squares problem for every column m_k og M, we can solve them independently from each other. This property gives the algorithm a great advantage by being inherently parallel.

3.2 Computing the sparsity structure $\mathcal{J}$ and $\mathcal{I}$

We want to find a $\mathcal{J}$ that represents the allowable positions for the preconditioner values in M. Computing an efficient sparsity pattern, will help us avoid entries which will only contribute little to M.
Let $\mathcal{J}$ be the set of indices j where m_k has nonzero entries, and denote the length of this set by n_2.

$$\mathcal{J} = \{j : m_k(j) \neq 0\} \tag{4}$$

$$n_2 = |\mathcal{J}| \tag{5}$$

We denote the reduced vector $m_k(\mathcal{J})$ by $\hat{m}_k$. Now because of the sparsity, the reduced system $A(.,\mathcal{J})$ will properly have a lot of zero rows, which is irrelevant for the solution. We eliminate these by letting $\mathcal{I}$, be the set if indices i where $A(i, \mathcal{J})$ is not identically zero

$$\mathcal{I} = \left\{ i : \sum_{j \in \mathcal{J}} |a_{ij}| \neq 0 \right\} \tag{6}$$

$$n_1 = |\mathcal{I}| \tag{7}$$

The resulting submatrix $A(\mathcal{I}, \mathcal{J})$, will be denoted by $\hat{A}$.

$$\underset{A}{\begin{pmatrix} \times & 0 & 0 & 0 \\ 0 & \times & 0 & \times \\ 0 & 0 & 0 & \times \\ 0 & \times & 0 & 0 \end{pmatrix}} \underset{m_k}{\begin{pmatrix} 0 \\ \times \\ 0 \\ \times \end{pmatrix}} \Rightarrow \underset{A(.,\mathcal{J})}{\begin{pmatrix} 0 & 0 \\ \times & \times \\ 0 & \times \\ \times & 0 \end{pmatrix}} \underset{\hat{m}_k}{\begin{pmatrix} \times \\ \times \end{pmatrix}} \Rightarrow \underset{\hat{A}}{\begin{pmatrix} \times & \times \\ 0 & \times \\ \times & 0 \end{pmatrix}} \underset{\hat{m}_k}{\begin{pmatrix} \times \\ \times \end{pmatrix}}$$

Figure 2: $\mathcal{J}$ is the indices of the blue columns and $\mathcal{I}$ is the indices of the red rows. $\mathcal{J}$ = 1,3 and $\mathcal{I}$ = 1,2,3

We can then reduce the minimisation problem (make ref to earlier) to

$$\min_{\hat{m}_k} ||\hat{A}\hat{m}_k - \hat{e}_k||_2 \tag{8}$$

with the definitions and sizes

$$\hat{A}_k := A(\mathcal{I}, \mathcal{J}) \in \mathbb{R}^{n_1 \times n_2}, \qquad \hat{m}_k := m_k(\mathcal{J}) \in \mathbb{R}^{n_2}, \qquad \text{and } \hat{e}_k := e_k(\mathcal{I}) \in \mathbb{R}^{n_1}. \tag{9}$$

The dimensions of this least squares problem is very small, since A and M er both very sparse matrices.

3.3 Applying the Householder QR decomposition

For a the nonsingular matrix A, the submatrix $\hat{A}$ must have full rank, and thus we can apply the QR decomposition

$$\hat{A} = Q \begin{pmatrix} R \\ 0 \end{pmatrix}, \tag{10}$$

where

$$Q \in \mathbb{R}^{n_1 \times n_2} \text{ and } R \in \mathbb{R}^{n_2 \times n_2}. \tag{11}$$

We will base our solutions of the minimisation problems on the QR decomposition via Householder transformations, which is an $O(n^3)$ algorithm.

A matrix H of the form

$$H = I - 2\frac{vv^T}{v^T v} \tag{12}$$

is called a Householder reflection, with v being a Householder vector. We can use the Householder reflections to transform the matrix $\hat{A}$ of our least-squares problem to an upper triangular form. We don't compute our Q explicitly but instead as a sum of Householder reflections.

$$Q = H_1 \cdots H_k, \tag{13}$$

where $k = \min(n_1 - 1, n2)$ and H is the Householder reflections. $\tilde{R}$ is found by isolating it in the QR decomposition.

$$\tilde{R} = \begin{pmatrix} R \\ 0 \end{pmatrix} = H_k \cdots H_1 \hat{A} \tag{14}$$

Since we can reconstruct the Householder transformations H_k quite easily from the Householder vectors, we can store them efficiently in the lower triangular part of $\tilde{R}$, that is not used for anything else than zeros.

3.4 Computing the nonzero entries of $\hat{m}_k$

We compute the nonzero entries of $\hat{m}_k$ by

$$\hat{m}_k = R^{-1}\hat{c}, \text{ where } \hat{c} = Q^T \hat{e}_k \tag{15}$$

We solve this for each $k = 1, ..., n$ and set $m_k(\mathcal{J}) = \hat{m}_k$. This way we have updated the entries of M to be closer to an approximate inverse, which minimises the frobenius norm $||AM - I||_F$.

3.5 Improving upon M by updating the sparsity structure

Now we have updated M once, but the goal is to keep improving upon M to obtain an even more precise inverse. To do so, we will augment the sparsity structure and reduce the current error. We will explain this by showing how to update a single column m_k.

We calculate the residual vector r of m_k, which is the k'th column of M.

$$r = Am_k - e_k \tag{16}$$

If the vector r only contains zeros, then m_k is already the k'th column of A^{-1} and thus cannot be improved upon.

For the sake of the explanation, we assume that $r \neq 0$ and therefore we must augment the set of indices $\mathcal{J}$ to reduce the norm of the residual r. We denote by $\mathcal{L}$ the set of indices for which r has nonzero entries. Since A and m_k are sparse most entries of r will be zero. If the set $\mathcal{I}$ does not contain k, it must also be included in $\mathcal{L}$, since $r(k)$ would then be equal to -1.

$$\mathcal{L} = \{l : r(l) \neq 0\} \cup \{k\} \tag{17}$$

For every $l \in \mathcal{L}$ we can define a set $\mathcal{N}_l$, which consists of the nonzero indices of the l'th row in A

$$\mathcal{N}_l = \{j : a_{lj} \neq 0\} \tag{18}$$

The column indices are potential new candidates to be added to $\mathcal{J}$, since the other would vanish by multiplying with zero when computing the matrix-vector product. We will denote these new candidates as

$$\tilde{\mathcal{J}} = \bigcup_{l \in \mathcal{L}} \mathcal{N}_l \tag{19}$$

Now we will select the indices j that will lead to the most profitable reduction in $||r||_2$, by solving for each $j \in \mathcal{J}$ the 1D minimisation problem

$$\min_{\mu_j} ||r + \mu_j A e_j||_2 \tag{20}$$

The solution to this minimisation problem is

$$\mu_j = -\frac{r^T A e_j}{||Ae_j||_2^2} \tag{21}$$

In order to see which μ_j would result in the smallest norm of the residual, we calculate the new residuals we would achieve if we added the candidate j to $\mathcal{J}$. So for the new residual $r_{new} = r + \mu_j A e_j$ we compute the 2-norm ρ_j for each j.

$$\rho_j^2 = \|r_{new}\|_2^2 - \frac{(r^T A e_j)^2}{\|Ae_j\|_2^2} \tag{22}$$

When implementing the SPAI algorithm we will create a parameter for the limit of new indices to be added to the current J in each update step. The algorithm will take this number of indices with the smallest values of ρ_j and union with $\mathcal{J}$. There is a variety of ways to make this choice, which we will discuss in the implementation section (Section ??). However, there must be at least one index $j \in \tilde{\mathcal{J}}$ that will result in a smaller residual.

Now $\mathcal{J} \cup \tilde{\mathcal{J}}$ contains the columns indices of all the nonzero entries of $A(\mathcal{L}, .)$ and $\mathcal{J} \cap \tilde{\mathcal{J}} = \varnothing$. We will use this augmented set to solve the least squares problem again and hence obtain a better approximation of m_k of the k'th column of A^{-1}.

In a similiar fashion, we have computed the new column indices $\tilde{\mathcal{J}}$, we also have to compute the corresponding row indices $\tilde{\mathcal{I}}$. $\tilde{\mathcal{I}}$ is the indices of the rows, which corresponds to the nonzero rows of $A(., \mathcal{J} \cup \tilde{\mathcal{J}})$ not contained in $\mathcal{I}$ yet. We denote by $\tilde{n_1}$ and $\tilde{n_1}$ the length of the sets $\tilde{\mathcal{I}}$ and $\tilde{\mathcal{J}}$, respectfully. We then have to solve the LS problem for the larger submatrix

$$\tilde{A} = A(\mathcal{I} \cup \tilde{\mathcal{I}}, \mathcal{J} \cup \tilde{\mathcal{J}}) \tag{23}$$

of size $n_1 + \tilde{n_1} \times n_2 + \tilde{n_2}$. To solve this, we will use the QR decomposition of $\tilde{A}$, but there is no need to compute the full QR decomposition again. This will be discussed in the next section (Section X).

As long as norm of the residual is larger than a tolerance provided by the user or the maximal amount of fill-ins has been reached, the update step is repeated until one the criteria are met. The algorithm will have computed the k'th column of A^{-1} or an approximate as close as the parameters and input would let us. In section X we present the full SPAI algorithm as pseudocode.

3.6 Updating the QR decomposition

An expensive step in the algorithm is computing the QR decomposition for each iteration of the while-loop, since augmenting the sparsity pattern will lead to expanding LS problems. So instead of performing a new decomposition, it is possible to update the one we already computed in a previous step. We want to use the available QR decomposition for $\hat{A}$ to update the QR decomposition for $A(\mathcal{I} \cup \tilde{\mathcal{I}}, \mathcal{J} \cup \tilde{\mathcal{J}})$.

First, we compute $\tilde{A}$ with rows from the union of $\mathcal{I}$ and $\tilde{\mathcal{I}}$ and the columns from the union of $\mathcal{J}$ and $\tilde{\mathcal{J}}$.

$$\tilde{A} = A(I \cup \tilde{\mathcal{I}}, \mathcal{J} \cup \tilde{\mathcal{J}}) = \begin{pmatrix} \hat{A} & A(\mathcal{I}, \tilde{\mathcal{J}}) \\ 0 & A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}}) \end{pmatrix} \tag{24}$$

Our $\tilde{A}$ can be rewritten as

$$\tilde{A} = \begin{pmatrix} Q & \\ & I_{\tilde{n_1}} \end{pmatrix} \begin{pmatrix} R & \breve{A} \\ 0 & A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}}) \end{pmatrix} \tag{25}$$

where we denote $\breve{A} = Q^T A(\mathcal{I}, \tilde{\mathcal{J}})$. This $\breve{A}$ can be split in $Q_1^T A(\mathcal{I}, \tilde{\mathcal{J}})$, which has size $n_2 \times n_2$ and $Q_2^T A(\mathcal{I}, \tilde{\mathcal{J}})$, which has size $\tilde{n_1} + n_1 - n_2 \times n_2$. Simultaneously R is being split from it's zeros in

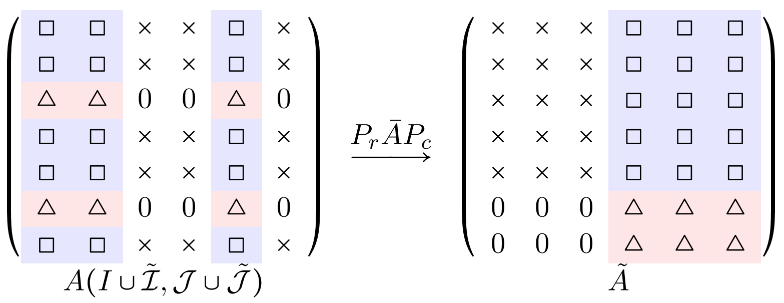
$P_r\bar{A}P_c$
$A(I \cup \tilde{\mathcal{I}}, \mathcal{J} \cup \tilde{\mathcal{J}})$
$\tilde{A}$

Figure 3: With row and column permutations P_r and P_c, the resulting matrix $\tilde{A}$ will contain a zero block, $\hat{A}$ that is denoted by the ×'s, $A(\mathcal{I}, \tilde{\mathcal{J}})$ that is denoted by □'s and $A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}})$ that is denoted by △'s. $\tilde{\mathcal{J}}$ is presented with the blue columns and $\tilde{\mathcal{I}}$ is presented the red rows.

the bottom. This is how we obtain B_1 and B_2.

$$\tilde{A} = \begin{pmatrix} Q & \\ & I_{\tilde{n}_1} \end{pmatrix} \left(\begin{array}{c|c} R & \breve{A}(0:n_2, 0:n_2) \\ \hline 0 & \breve{A}(n_2+1:n_1, 0:\tilde{n}_2) \\ 0 & A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}}) \end{array}\right) = \begin{pmatrix} Q & \\ & I_{\tilde{n}_1} \end{pmatrix} \left(\begin{array}{c|c} R & B_1 \\ \hline 0 & B_2 \end{array}\right) \tag{26}$$

B_1 is $n_2 \times n_2$ matrix and B_2 is a $\tilde{n}_1 + n_1 - n_2 \times \tilde{n}_2$ matrix. As it can be seen above, we only need to compute Q and R of B_2. We let

$$B_2 = Q_B \begin{pmatrix} R_B \\ 0 \end{pmatrix}, \tag{27}$$

Q_B has dimensions $\tilde{n}_1 + n_1 - n_2 \times \tilde{n}_1 + n_1 - n_2$ and R_B has dimensions $\tilde{n}_1 + n_1 - n_2 \times \tilde{n}_2$. Then we obtain

$$\tilde{A} = \begin{pmatrix} Q & \\ & \mathcal{I}_{\tilde{n}_1} \end{pmatrix} \begin{pmatrix} I_{\tilde{n}_1} & \\ & Q_B \end{pmatrix} \begin{pmatrix} R & B_1 \\ 0 & R_B \\ 0 & 0 \end{pmatrix} \tag{28}$$

from which we can compute Q_B and R_B.

We solve the augmented LS problem

$$\min_{\mathcal{J} \cup \tilde{\mathcal{J}}} ||\tilde{A}\tilde{M} - \tilde{e}||_2, \tag{29}$$

where $\tilde{e} = P_r e(\mathcal{I} \cup \tilde{\mathcal{I}})$ and $\tilde{M} = P_c^T M(\mathcal{J} \cup \tilde{\mathcal{J}})$. Finally, we recover the solution in the correct order with a final column permutation $M(\mathcal{J} \cup \tilde{\mathcal{J}}) = P_c \tilde{M}$.

By performing the QR decomposition on the smaller matrix B_2, instead of computing the full QR decomposition of $\tilde{A}$, we reduce the computational cost.

We have based our pseudocode for the QR update on algorithm 5 from Sedlacek's dissertation *Sparse Approximate Inverses for Preconditioning, Smoothing, and Regularization* (INSERT REF).

Update the QR decomposition

(a) Compute $\tilde{A}$ from $\hat{A}, A(I, \tilde{\mathcal{J}})$ and $A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}})$

(b) Compute $\breve{A} = Q^T A(\mathcal{I}, \tilde{\mathcal{J}})$

(c) Compute $B_1 = \breve{A}(0 : n_2, 0 : n_2)$

(d) Compute $B_2 = \breve{A}(n_2 + 1 : n_1, 0 : \tilde{n_2})$ above $A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}})$

(e) Do QR decomposition of B_2

(f) Compute Q_B and R_B from

$$A(I \cup \tilde{\mathcal{I}}, \mathcal{J} \cup \tilde{\mathcal{J}}) = \begin{pmatrix} Q & \\ & I_{\tilde{n_1}} \end{pmatrix} \begin{pmatrix} I_{\tilde{n_1}} & \\ & Q_B \end{pmatrix} \begin{pmatrix} R & B_1 \\ 0 & R_B \\ 0 & 0 \end{pmatrix}$$

(g) Solve the augmented LS problem for $\hat{m_k}$ and compute new residual

(h) Set $M(\mathcal{J} \cup \tilde{\mathcal{J}})$

3.7 Pseudocode

Inspired by the SPAI algorithm by Grote and Huckle (Grote 1995), we have written our own pseudocode, which explains some of the steps of the algorithm more explicitly.

```
For every column $\hat{m}_k$ of $M$ do
(a) Find initial sparsity $\mathcal{J}$ of $\hat{m}_k$
(b) Compute the row indices $\mathcal{I}$ of the corresponding nonzero entries of $A(\mathcal{I},\mathcal{J})$
(c) Create $\hat{A} = A(\mathcal{I},\mathcal{J})$
(d) Do QR decomposition of $\hat{A}$
(e) Compute the solution $m_k$ for the least squares problem
    (a) Compute $\hat{c} = Q^T\hat{e}_k$
    (b) Compute $\hat{m}_k = R^{-1}\hat{c}$
    (c) Set $m_k(\mathcal{J}) = \hat{m}_k$
    (d) Compute residual
    While residual < tolerance do
    (a) Let $\mathcal{L}$ be the set of indices, where $r(l) \neq 0$
    (b) Set $\tilde{\mathcal{J}}$ to all new column indices of $A$ that appear in all $\mathcal{L}$ rows, but not in $\mathcal{J}$ yet
    (c) For each $j$ in $\tilde{\mathcal{J}}$ compute: $\rho_j^2 = ||r_{new}||_2^2 - \frac{(r^TAe_j)^2}{||Ae_j||_2^2}$
    (d) Find the indices $\tilde{\mathcal{J}}$ corresponding to the smallest $s$ elements of $\rho^2$
    (e) Determine the new indices $\tilde{\mathcal{I}}$
    (f) Make $I \cup \tilde{\mathcal{I}}$ and $\mathcal{J} \cup \tilde{\mathcal{J}}$
    (g) Update the QR decomposition
        (a) Compute $\tilde{A}$ from $\hat{A}, A(I,\tilde{\mathcal{J}})$ and $A(\tilde{\mathcal{I}},\tilde{\mathcal{J}})$
        (b) Compute $\breve{A} = Q^TA(\mathcal{I},\tilde{\mathcal{J}})$
        (c) Compute $B_1 = \breve{A}(0:n_2, 0:n_2)$
        (d) Compute $B_2 = \breve{A}(n_2+1:n_1, 0:\tilde{n}_2)$ above $A(\tilde{\mathcal{I}},\tilde{\mathcal{J}})$
        (e) Do QR decomposition of $B_2$
        (f) Compute $Q_B$ and $R_B$ from
            $$A(I \cup \tilde{\mathcal{I}}, \mathcal{J} \cup \tilde{\mathcal{J}}) = \begin{pmatrix} Q & \\ & I_{\tilde{n}_1} \end{pmatrix} \begin{pmatrix} I_{\tilde{n}_1} & \\ & Q_B \end{pmatrix} \begin{pmatrix} R & B_1 \\ 0 & R_B \\ 0 & 0 \end{pmatrix}$$
        (g) Solve the augmented LS problem for $\hat{m}_k$ and compute new residual
        (h) Set $M(\mathcal{J} \cup \tilde{\mathcal{J}})$
    (h) Set $\mathcal{I} = I \cup \tilde{\mathcal{I}}$ and $\mathcal{J} = \mathcal{J} \cup \tilde{\mathcal{J}}$ and $A' = A(\mathcal{I},\mathcal{J})$
(f) Set $m_k(\mathcal{J}) = \hat{m}_k$
```

4 Implementation

The previous section was devoted to the theoretical development and presentation of the SPAI algorithm. In this section we will focus on the implementation of the SPAI preconditioner in both a sequential and parallel environment.

4.1 Structure for the parallel implementation

parallelSPAI

(a) Compute the number of batches

(b) For loop iterating through the number of batches

- (a) Kernel - computes $\mathcal{J}$ and $\mathcal{J}$ and their lengths n_1 and n_2 for all batch subproblems
- (b) compute the max n_1 and n_2 of all batches
- (c) Kernel - makes $\hat{A}$ and pads them with zeros, so they all have size $maxn_1 * maxn_2$
- (d) run `qrBatched` on $\hat{A}$
  - (a) Run `cublasSgeqrfBatched` from the cuBLAS library
  - (b) Kernel - getting all batch R's from all batch AHat's
  - (c) Compute Q with Algorithm 1 from Kerr Campbell Richards
    - (a) Kernel - computes $Qv = Q * v$
    - (b) Kernel - computes $Qvvt = Qv * v^T$
    - (c) Kernel - computes $Q - Qvvt$
- (e) Solve the LS problem and make $\hat{m}_k$
  - (a) nogle kerner eller hvad?
- (f) While loop running until tolerance or max number of iterations is met
  - (a) Kernel - computes L, $\tilde{\mathcal{I}}$ and $\tilde{\mathcal{J}}$
  - (b) Find the max $\tilde{n}1$ and $\tilde{n}2$ of all batches
  - (c) Update the QR decomposition with algorithm 5 from Sedlacek
    - (a) Kernel - makes $\tilde{A}$
    - (b) Kernel - makes permutation matrices
    - (c) Kernel - computes $\breve{A} = Q^T \cdot A(\mathcal{I}_k, \tilde{\mathcal{J}}_k)$
    - (d) Kernel - sets B_1
    - (e) Kernel - sets B_2
    - (f) run `qrBatched` on B2
    - (g) Kernel - puts the new Q together
    - (h) Kernel - puts the new R together
    - (i) Solve the LS problem
    - (j) Update $\hat{m}_k$
  - (d) Update $\mathcal{I}$, $\mathcal{J}$, $\hat{A}$, Q and R for next iteration of the while loop

5 Testing and benchmarking

6 Conclusion

x