Project 2 - Randomized Nyström Algorithm

Stefano Ferioli, Oskar Koiner

December 2023

Introduction

The goal of this project is to analyze and implement the Randomized Nyström Algorithm using MPI. This algorithm is used for calculating low rank approximations of symmetric positive semidefinite (SPSD) matrices.

In Section 1, we introduce the algorithm from a theoretical point of view. In Section 2, we cover some theoretical elements of sketching and we present the two sketching matrices we use in this project. Section 3 details how we have implemented the algorithm in Python, starting with discussing the sequential implementation and exploring its parallelization using MPI. In Section 3.3, we explain how to generate the sketching matrices using multiple processors.

Moving forward, we examine the numerical and runtime performance of these implementations on five test matrices presented in Section 4.1.

In Section 4, we compare the numerical performance of our parallel implementation against the test matrices, when varying the different parameters of the algorithm. In Section 4.5, we investigate how effectively our approximations can estimate the leading singular values of the test matrices, providing a practical perspective on the algorithm's usefulness.

Finally, in Section 5, we analyze the runtime performance of our implementation, comparing its speed in sequential and on multiple cores.

1 Randomized Nyström Algorithm

The goal of Randomized Nyström Algorithm is to represent a given matrix $A \in \mathbb{R}^{n\times n}$ by some approximated matrix A_k with a lower rank $k < n$. The starting point of the algorithm is a rank ℓ approximation of A, in the form

$$A_{\mathrm{N}yst} := (A\Omega)(\Omega^\top A\Omega)^\dagger(\Omega^\top A) \in \mathbb{R}^{n\times n},$$

where $(\Omega^\top A\Omega)^\dagger$ is the pseudoinverse of $\Omega^\top A\Omega$ and $\Omega \in \mathbb{R}^{n\times \ell}$ is a sketching matrix. We assume $k < \ell \ll n$. To further reduce the rank of $A_{\mathrm{N}yst}$ to k we have two choices. We can either approximate only the core matrix $\Omega^\top A\Omega$ or the whole A_{Nyst}. In this project we focus on the second approach. In Algorithm 1, we see how this can be achieved.

Algorithm 1 Randomized Nyström approximation using the Cholesky decomposition

Step 1: Let $C = A\Omega.$ and $B = \Omega^\top C$.

Step 2: Calculate the Cholesky decomposition

$$B = LL^\top.$$

Step 3: Obtain a matrix Z by solving the linear system using back substitution

$$Z = C(L^\top)^{-1}.$$

Step 4: Calculate the QR decomposition

$$Z = QR.$$

Step 5: Calculate the singular value decomposition (SVD) of R and its truncation

$$R = U\Sigma V^\top \simeq U_k \Sigma_k V^\top{}_k.$$

Step 6: Set $\hat{U}_k = QU_k$ (or equivalently $\hat{U}_k = ZV_k\Sigma_k^{-1}$) and return

$$\hat{U}_k \Sigma_k^2 \hat{U}_k^\top \simeq A_{Nyst}.$$

To show why the result of Algorithm 1 is in facts a rank-k approximation of A_{Nyst}, we write

$$\begin{aligned}
\hat{U}_k \Sigma_k^2 \hat{U}_k^\top &= QU_k \Sigma_k \Sigma_k^\top U_k^\top Q^\top \\
&= QU_k \Sigma_k V_k^\top V_k \Sigma_k^\top U_k^\top Q^\top \\
&\simeq QRR^\top Q^\top \\
&= ZZ^\top \\
&= A\Omega (L^\top)^{-1} L^{-1} \Omega^\top A^\top \\
&= A\Omega (LL^\top)^{-1} \Omega^\top A^\top \\
&= A\Omega (\Omega^\top A \Omega)^{-1} \Omega^\top A^\top.
\end{aligned}$$

However, here we have the inverse of $\Omega^\top A\Omega$ instead of its pseudoinverse. Unfortunately, Algorithm 1 fails in **Step 2** if $\Omega^\top A\Omega$ is numerically singular. In this case, as in [1], we replace L by a square root of B in SVD form. In other words, we write $B = U_B \Sigma_B V_B^\top = U_B \Sigma_B U_B^\top$ (since B is SPSD), and we set $L = U_B \sqrt{\Sigma_B} U_B^\top$. By construction, we have $LL^\top = U_B \sqrt{\Sigma_B} U_B^\top (U_B \sqrt{\Sigma_B} U_B^\top)^\top = U_B \sqrt{\Sigma_B}\sqrt{\Sigma_B} U_B^\top = B$. We then replace $(L^\top)^{-1}$ with $(L^\top)^\dagger$. This can be simply calculated as

$$(L^\top)^\dagger = (U_B \sqrt{\Sigma_B} U_B^\top)^\dagger = U_B \sqrt{\Sigma_B}^\dagger U_B^\top, \tag{1}$$

where $\sqrt{\Sigma_B}^\dagger$ is a diagonal matrix whose entries are given by $(\sqrt{\Sigma_B}^\dagger)_{i,i} = \begin{cases} (\sqrt{\Sigma_B})_{i,i}^{-1} & \text{if } (\sqrt{\Sigma_B})_{i,i} \neq 0 \\ 0 & \text{otherwise} \end{cases}$

2 Sketching and sketching matrices

In this section, we give an overview of sketching and we present the sketching matrices we use in this project. As before, we sketch from the right, meaning the sketching matrices $\Omega \in \mathbb{R}^{n\times l}$ will be tall and skinny. They are applied to a matrix $A \in \mathbb{R}^{n\times n}$ from the right, as $A\Omega$. The idea behind sketching is to embed high dimensional data into a lower dimensional space without losing too much geometry. Being more precise, given any vectors x and y in the high dimensional space, we want their sketched versions $\hat{x}$ and $\hat{y}$ to satisfy

$$|\langle \hat{x}, \hat{y}\rangle - \langle x, y\rangle| \leq \varepsilon \|x\|_2 \|y\|_2 \tag{2}$$

for some $\varepsilon > 0$. Since the sketched dimension is (much) lower than the original one this cannot be achieved for all x and y. One remedy is to create Ω randomically and to require that Equation 2 holds with some probability $1 - \delta < 1$. This is usually combined with the definition of oblivious subspace embedding. A deeper analysis of these techniques can be found for instance in [6].

In the following two paragraphs, we present the two types of sketching matrix we use in this project.

2.1 Short-axis-sparse sketching operators

The first sketching operator we use is the short-axis-sparse sketching operator (SASO) [4]. It can be generated in two different ways. First we fix t, the number on non-zero elements in each row of Ω. For each row we can now sample the indices for these non-zero elements either by

- sampling t unique integers uniformly from $\{1, 2, \cdots, l\}$, or
- divide $\{1, 2, \cdots, l\}$ into t contiguous subsets of similar size and then sample one index from each of those.

We choose to utilise the second method. After deciding which entries of each line are non-zeros, we need to set their values. Usually, they are set to independent Rademacher random variables, that is either $+1$ or -1 with equal probability. However, as mentioned in [4, Section 2.4.1], there is a chance that doing so some column of Ω comes out to be perpendicular to some row A. To protect us from this risk we will draw the non-zero elements as independent uniform random variables on the set $[-2, -1] \cup [1, 2]$.

2.2 Block subsampled randomized Hadamard transform

The second sketching operator we use is the block subsampled randomized Hadamard transform (block SRHT) [1]. We consider Ω split into m blocks, as

$$\Omega = \begin{bmatrix} \Omega_1 \\ \Omega_2 \\ \vdots \\ \Omega_m \end{bmatrix} \in \mathbb{R}^{n \times \ell},$$

where $\Omega_i \in \mathbb{R}^{\frac{n}{m} \times \ell}$. Typically, m is set equal to the number of processors s, or to $\sqrt{s}$, like in our case.

We define $\Omega_i = \sqrt{\frac{n/m}{\ell}} \tilde{D}_i H R D_i$, where

- $\tilde{D}_i \in \mathbb{R}^{\frac{n}{m} \times \frac{n}{m}}$ and $D_i \in \mathbb{R}^{\ell \times \ell}$ for $i = 1...m$ are square diagonal matrices with independent Rademacher entries,
- $R \in \mathbb{R}^{\frac{n}{m} \times \ell}$ is a unique sampling matrix that picks ℓ random different columns from H, and
- $H = \frac{1}{\sqrt{n/m}} H_{n/m}$, where $H_{n/m}$ is the Hadamard matrix of order $\frac{n}{m}$, recursively defined as

$$H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \text{ and } H_{2n} = \begin{bmatrix} H_n & H_n \\ H_n & -H_n \end{bmatrix}.$$

3 Implementation of Randomized Nyström Algorithm

In this section, we present how Randomized Nyström Algorithm has been implemented using MPI. We first consider the sequential implementation, before presenting how this can be parallelized using MPI. Section 3.3 shows how the two sketching matrices can be generated using multiple processors.

For both the implementations, we require n to be a power of 2. This condition can always be achieved by padding the input data with zeros to make its dimension a power of 2 [1]. However, sometimes this translates in almost quadrupling the amount of memory required to run the algorithm, for example if $n = 2^a + 1$, $a \in \mathbb{N}^+$.

3.1 Sequential implementation

Implementing Algorithm 1 in sequential programming is quite straightforward. The only adjustment we need to make is to handle the case when $\Omega^\top A \Omega$ is numerically singular and **Step 2** fails. When this happens, we have to compute the pseudoinverse as given in Equation 1. Algorithm 2 includes this adjustment.

In practice, the condition of B being strictly positive definite can be implemented by trying to run the Cholesky decomposition of B and follow the other approach only if an error is caught.

Algorithm 2 Sequential Randomized Nyström Algorithm

Input: Matrices A, Ω, integer k
Output: rank-k approximation of A

Step 1:
$C \leftarrow A\Omega$
$B \leftarrow \Omega^\top C$

Step 2 and 3:
if B is strictly positive-definite **then**
    $L \leftarrow \text{cholesky}(B)$
    $Z \leftarrow C(L^\top)^{-1}$ using back substitution
else
    $[U_B, \Sigma_B, V_B^\top] \leftarrow \text{svd}(B)$
    $Z \leftarrow C U_B \sqrt{\Sigma_B}^\dagger V_B^\top$
end if

Step 4:
$[Q, R] \leftarrow \text{qr}(Z)$

16: **Step 5:**
17: $[U, \Sigma, V^\top] \leftarrow \text{svd}(R)$
18: $U_k \leftarrow U[:k, :k]$
19: $\Sigma_k \leftarrow \Sigma[:k, :k]$
20: $V_k^\top \leftarrow V^\top[:k, :k]$

21: **Step 6:**
22: $\hat{U}_k \leftarrow QU_k$
23: **return** $\hat{U}_k \Sigma_k^2 \hat{U}_k^\top$

3.2 Parallel implementation

In this section we present how we parallelized Algorithm 2 using MPI. We analyse how each step is parallelised. For the sake of simplicity, the analysis is done assuming to have only 4 processors ($s = 4$), nevertheless, the same reasoning can be generalized to any s power of 4.

The reason why s must be a power of 4 is the following. Firstly, the first multiplication in **Step 1** requires s to be a perfect square. Secondly, TSQR (**Step 4**) require s to be a power of 2. In general, it is possible to generalise the algorithm to any number of processors, but its not trivial. In Appendix A, we explore how the algorithm can be generalized also to odd power of 2.

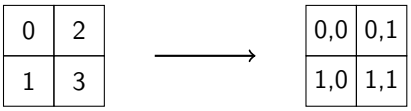
In Appendix B and [4, Appendix A.2.1], alternative implementations for block SRHT and SASO are presented.

Step 1

We want to compute $C = A\Omega$. The idea behind this step is to distribute the matrix A among the processors using a two-dimensional block distribution, and to execute the multiplication block-wise, as

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \begin{pmatrix} \Omega_0 \\ \Omega_1 \end{pmatrix} = \begin{pmatrix} A_{00}\Omega_0 + A_{01}\Omega_1 \\ A_{10}\Omega_0 + A_{11}\Omega_1 \end{pmatrix}. \tag{3}$$

The processors are organized into a square grid with dimensions $\sqrt{s} \times \sqrt{s}$. Row and column indices for each processor are computed using the formulas: $row = rank \% \sqrt{s}$ and $col = rank / \sqrt{s}$, where $\%$ is the euclidean remainder operator. We choose to number the processors column-wise, rather than row-wise, as this proves to be very useful in the next steps. An example of conversion from $rank$ to row and col is the following.



For matrix distribution, A is first partitioned horizontally into $\sqrt{s}$ wide blocks. These blocks are distributed among the processors in the 0-th column. Subsequently, each wide block is further scattered among processors on the same row. As a result, each processor owns a square block of A, $A_{row\,col} \in \mathbb{R}^{\frac{n}{\sqrt{s}} \times \frac{n}{\sqrt{s}}}$.

To execute multiplication 3, each processor executes one multiplication $A_{row\,col}\Omega_{col}$. To execute this multiplications, we generate Ω_{col} in each processor in such a way that it is the same for all the processor in the same column. We explain how this is done in Section 3.3.

The results are then summed together row-wise using a `Reduce`. We use row as root, so that the blocks of C are stored in the processors on the diagonal. This will be useful to execute the following multiplication

Now we want to compute $B = \Omega^\top A\Omega = \Omega^\top C$. Since C is tall and skinny, we want to have the matrix C distributed among the processors using a one-dimensional block distribution, and to execute the multiplication block-wise, as

$$\begin{pmatrix} \tilde{\Omega}_0 \\ \tilde{\Omega}_1 \\ \tilde{\Omega}_2 \\ \tilde{\Omega}_3 \end{pmatrix}^\top \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} = \begin{pmatrix} \tilde{\Omega}_0^\top & \tilde{\Omega}_1^\top & \tilde{\Omega}_2^\top & \tilde{\Omega}_3^\top \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} = \tilde{\Omega}_0^\top C_0 + \tilde{\Omega}_1^\top C_1 + \tilde{\Omega}_2^\top C_2 + \tilde{\Omega}_3^\top C_3\,,$$

where $\Omega_0 = \begin{pmatrix} \tilde{\Omega}_0 \\ \tilde{\Omega}_1 \end{pmatrix}$ and $\Omega_1 = \begin{pmatrix} \tilde{\Omega}_2 \\ \tilde{\Omega}_3 \end{pmatrix}$.

We scatter the blocks of C obtained in the previous step among the processors in the same column (this is why we reduced on the diagonal processors). Since Ω_{col} is already stored in the processors in the col-th column, we don't need to further move other matrices among the processors. We also notice that the blocks of C and Ω are stored in the processors in the correct order with respect to the rank. In other words, the processor $rank$ owns the $rank$-th block of C and Ω. This will be useful to execute TSQR and is the reason why we numbered the processors column-wise.

Each processor executes one multiplication $\tilde{\Omega}_{rank}^{\top} C_{rank}$. The results are then summed all together using a `Reduce`, having processor 0 as root.

Step 2 and 3

In this step, we have two scenarios: B can be strictly positive-definite or numerically singular.

In the first case, we want to compute the Cholesky factorization of B, $LL^{\top}$, and solve the linear system $CL^{-\top}$. Since $B \in \mathbb{R}^{\ell \times \ell}$ and $\ell \ll n$, we decide to compute its Cholesky factorization sequentially only on processor 0. Since C is already distributed among all the processors, we broadcast the result to all the processors in order to solve the linear system block-wise by back-substitution, as

$$\begin{pmatrix} Z_0 \\ Z_1 \\ Z_2 \\ Z_3 \end{pmatrix} = \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} L^{-\top} = \begin{pmatrix} C_0 L^{-\top} \\ C_1 L^{-\top} \\ C_2 L^{-\top} \\ C_3 L^{-\top} \end{pmatrix}.$$

If B is numerically singular, we want to compute the SVD of B, $U_B \Sigma_B U_B$, and multiply $CU_B \sqrt{\Sigma_B}^{\dagger} U_B^T$. Similarly, since $B \in \mathbb{R}^{\ell \times \ell}$ is relatively small, we decide to compute its SVD sequentially. We compute $(L^{\top})^{\dagger} = U_B \sqrt{\Sigma_B}^{\dagger} U_B^T$ and we broadcast the result to all the processors in order to execute the multiplication block-wise, as

$$\begin{pmatrix} Z_0 \\ Z_1 \\ Z_2 \\ Z_3 \end{pmatrix} = \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} (L^{\top})^{\dagger} = \begin{pmatrix} C_0 (L^{\top})^{\dagger} \\ C_1 (L^{\top})^{\dagger} \\ C_2 (L^{\top})^{\dagger} \\ C_3 (L^{\top})^{\dagger} \end{pmatrix}.$$

Step 4

We want to compute the QR factorization of $Z \in \mathbb{R}^{n \times \ell}$. Since each processor already owns a block of Z, $Z_{rank} \in \mathbb{R}^{\frac{n}{s} \times \ell}$, and Z is tall and skinny, we choose to do this using TSQR. Before proceeding, we need to add to our working assumption that Z_{rank} is a vertical matrix, this translates in $\frac{n}{s} \geq \ell$.

Firstly, every processor performs the QR factorization of the block they own, $Z_{rank} = Q_{rank} R_{rank}$, both Q and R factors are actually generated. Then, the R factors are paired, stacked together as $\begin{pmatrix} R_0 \\ R_1 \end{pmatrix}$ and $\begin{pmatrix} R_2 \\ R_3 \end{pmatrix}$ and factorized as $Q_{01} R_{01}$ and $Q_{23} R_{23}$ on processors 0 and 2, respectively. Lastly, $\begin{pmatrix} R_{01} \\ R_{23} \end{pmatrix}$ is factorized as $Q_{0123} R_{0123}$ on processor 0.

This process can be formalised as

$$Z = \begin{pmatrix} Z_0 \\ Z_1 \\ Z_2 \\ Z_3 \end{pmatrix} = \begin{pmatrix} Q_0 R_0 \\ Q_1 R_1 \\ Q_2 R_2 \\ Q_3 R_3 \end{pmatrix} = \begin{pmatrix} Q_0 & & & \\ & Q_1 & & \\ & & Q_2 & \\ & & & Q_3 \end{pmatrix} \left(\begin{pmatrix} R_0 \\ R_1 \\ R_2 \\ R_3 \end{pmatrix} \right) = \begin{pmatrix} Q_0 & & & \\ & Q_1 & & \\ & & Q_2 & \\ & & & Q_3 \end{pmatrix} \begin{pmatrix} Q_{01} R_{01} \\ Q_{23} R_{23} \end{pmatrix} =$$

$$= \begin{pmatrix} Q_0 & & & \\ & Q_1 & & \\ & & Q_2 & \\ & & & Q_3 \end{pmatrix} \begin{pmatrix} Q_{01} & \\ & Q_{23} \end{pmatrix} \begin{pmatrix} R_{01} \\ R_{23} \end{pmatrix} = \begin{pmatrix} Q_0 & & & \\ & Q_1 & & \\ & & Q_2 & \\ & & & Q_3 \end{pmatrix} \begin{pmatrix} Q_{01} & \\ & Q_{23} \end{pmatrix} Q_{0123} R_{0123}.$$

At the end of the algorithm R is stored in processor 0. The Q factors are located as follows.

Processor 0	Q_0	Q_{01}	Q_{0123}
Processor 1	Q_1		
Processor 2	Q_2	Q_{23}	
Processor 3	Q_3		

Step 5

We want to compute the rank-k truncated SVD of R, $U_k \Sigma_k V_k^\top$. Since $R \in \mathbb{R}^{\ell \times \ell}$, and $\ell \ll n$ we decide to compute its SVD sequentially only on processor 0.

Step 6

We want to compute $\hat{U}_k = QU_k$. We have two choices: we can either execute the multiplication of the orthogonal matrices we found above or compute $\hat{U}_k$ as $QU_k = ZR^{-1}U_k$. We choose the first approach. The multiplication we want to execute is

$$\begin{pmatrix} Q_0 & & & \\ & Q_1 & & \\ & & Q_2 & \\ & & & Q_3 \end{pmatrix} \begin{pmatrix} Q_{01} & \\ & Q_{23} \end{pmatrix} Q_{0123} U_k \,.$$

We start by executing $\hat{U}_k^{0123} = Q_{0123}U_k$ on processor 0 without requiring any communication. We then continue by exploiting the structure and the locality of the matrices in order to minimize the number of multiplications and communications. For example, we notice that

$$\begin{pmatrix} \hat{U}_k^{01} \\ \hat{U}_k^{23} \end{pmatrix} = \begin{pmatrix} Q_{01} & \\ & Q_{23} \end{pmatrix} U_k^{0123} = \begin{pmatrix} Q_{01}\hat{U}_k^{0123}[0:\ell,:] \\ Q_{23}\hat{U}_k^{0123}[\ell:2\ell,:] \end{pmatrix} .$$

Since, Q_{01} and Q_{23} are located in processors 0 and 1 respectively, this second multiplication can be easily parallelized by executing the two matrix multiplications directly there. This same reasoning can be extended to the other multiplications generating a sort of reversed tree multiplication.

After multiplying $\hat{U}_k$ is distributed on all the processor and can be easily reconstructed using a `gather`.

3.3 Block generation of the sketching matrices

In this section we explain how to generate the blocks of Ω utilised in **Step 1** in Section 3.2, namely, Ω_{col}.

As notation suggests, these blocks have to be the same for every processor in the same column col. This is in general not already guaranteed, since the sketching matrix Ω is generated randomly. In addition, sometimes there are properties of the sketching matrix that need to be the same in every block, and so in every processor.

To respect these specifics, we introduce two random seed per processor that we will use to generate the blocks of Ω. The first random seed is shared by all the processors, we call it `general_random_seed`. The second is shared by all the processors in the same column and is different among the processors in different columns, we call it `col_random_seed`. We use the `general_random_seed` to generate the aspects of Ω that are shared by the whole sketching matrix, and we use `col_random_seed` to generate the information that is specific of a particular block Ω_{col}.

In the following, we explain how the two sketching matrices we are analysing are generated. These same algorithms extend trivially to the sequential case.

Short-axis-sparse sketching

The process of generating a Short-axis-sparse sketch matrix is explained in Algorithm 3, where $r = \frac{n}{\sqrt{s}}$ is the number of rows in Ω_{col} and $t \leq \ell$ is the number of non-zero entries in each row.

Algorithm 3 Short-axis-sparse sketching block generation

Input: Integers ℓ, r, t, random seed `col_random_seed`
Output: $\Omega_{col} \in \mathbb{R}^{r \times \ell}$

Initialize random number generator `col_rng` with `col_random_seed`
Divide the interval $[0, \ell]$ into t continuous subintervals I_j of similar length

for $i = 1, ..., r$ **do**
    Set the i-th row of Ω_{col} to 0
    for $j = 1, ..., t$ **do**
        Use `col_rng` to sample uniformly one index h_j from I_j
        Use `col_rng` to sample U_{ij} uniformly in $[-2, -1] \cup [1, 2]$
        $\Omega_{col}^{i,h_j} \leftarrow U_{ij}$
    end for
end for

Block SRHT

The process of generating a block SRHT sketch matrix is explained in Algorithm 4, where $r = \frac{n}{\sqrt{s}}$ is the number of rows in Ω_{col}. The entries of the Hadamard matrix of order r, H_r, are computed as

$$H_{i,j}^r = (-1)^{\mathrm{bitcount}(i \& j)},$$

where $\&$ represent the bitwise multiplication, and $\mathrm{bitcount}$ returns the number of 1s in the binary representation of its argument.

Algorithm 4 Block SRHT sketching block generation

Input: Integers ℓ, r, random seeds `general_random_seed`, `col_random_seed`
Output: $\Omega_{col} \in \mathbb{R}^{r \times \ell}$

Initialize random number generator `general_rng` with `general_random_seed`
Initialize random number generator `col_rng` with `col_random_seed`

Use `general_rng` to sample ℓ unique indices h_i in $1...r$
Use `col_rng` to sample $\tilde{d} \in \mathbb{R}^r$ and $d \in \mathbb{R}^\ell$ with independent Rademacher entries
for $i = 1, ..., r$ **do**
    for $j = 1, ..., \ell$ **do**
        $\Omega_{col}^{i,j} \leftarrow \frac{1}{\sqrt{\ell}} \tilde{d}_i d_j H_r^{i,h_j}$ ▷ H_r is the Hadamard matrix of order r, as defined in Section 2.2
    end for
end for

4 Numerical testing

In this section, we investigate how our algorithm performs from a numerical point of view. The metric we use to assess the quality of our results is the trace relative error, defined as

$$\frac{\|A - A_{\mathrm{Nyst}}\|_*}{\|A\|_*},$$

where $\|\cdot\|_*$ is the nuclear norm, defined as the sum of the singular values of its argument.

All the tests have been run on a MacBook Air with the specifics in Table 1. At the beginning of each test, `OMP_NUM_THREADS` has been set to 1 in order to prevent the BLAS library from multi-threading. As suggested in [4, Appendix A.2], when using SASO sketching matrix we fix the number of non-zero elements to 8. We will further investigate this choice in Section 4.4.

Model Name	MacBook Air
Chip	Apple M2
Total Number of Cores	8 (4 performance and 4 efficiency)
Memory	16 GB

Table 1: System Specifics

4.1 Testing matrices

We present the matrices we utilize for numerical testing. Due to hardware limitations we fix $n = 2^{12}$.

The first two matrices we utilize are $\mathrm{PolyDecay}$ and $\mathrm{ExpDecay}$, as presented in [5]. We define them as

$$\mathrm{PolyDecay}(R,p) = \mathrm{diag}(\underbrace{1,\dots,1}_{R}, 2^{-p}, 3^{-p}, \dots, (n-R+1)^{-p}) \in \mathbb{R}^{n\times n} \quad \text{and}$$

$$\mathrm{ExpDecay}(R,p) = \mathrm{diag}(\underbrace{1,\dots,1}_{R}, 10^{-p}, 10^{-2p}, \dots, 10^{-(n-R)p}) \in \mathbb{R}^{n\times n}.$$

Like in [5], we further define $A_{\mathrm{Poly}} = \mathrm{PolyDecay}(10,1)$ and $A_{\mathrm{Exp}} = \mathrm{PolyDecay}(10,0.25)$. We will use these two matrices in the numerical testing.

The other matrices we utilize are described in [1]. We consider the datasets MNIST [3] and YearPredictionMSD [2]. Using the radial basis function $\exp\left(\frac{-\|x_1-x_j\|^2}{\sigma^2}\right)$ we popolate a dense matrix of size $n \times n$ using the first n data points in the respective datasets. With regards to σ, we set it to 100, for MNIST, and to 10^5 and 10^6, for YearPredictionMSD. We define A_{MNIST}, $A_{\mathrm{Year\_a}}$ and $A_{\mathrm{Year\_b}}$, respectively.

4.2 First results

We start by executing an array of tests varying ℓ, k and the sketching matrix. We decided to execute each test only once, in order to be able to test for more different values of ℓ and k and for both the sketching matrices. The results are collected in Figure 1.

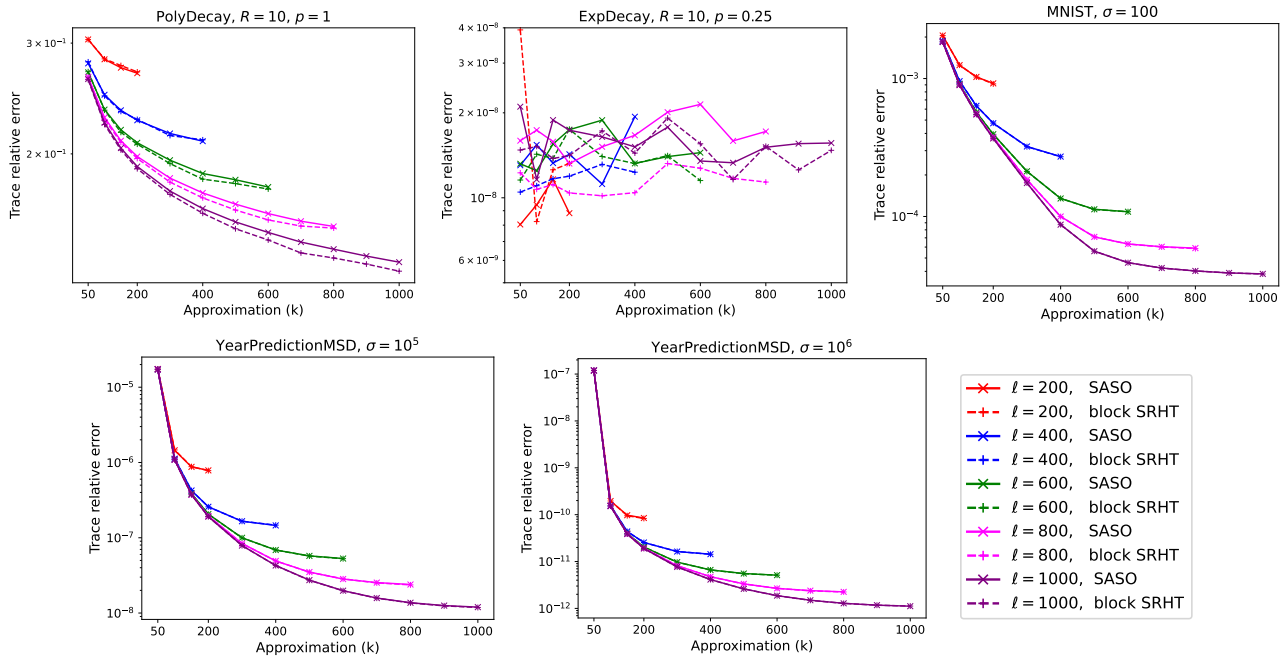


Figure 1

Except for A_{Exp}, our results display similar trends to the ones reported in [1]. For A_{MNIST} and $A_{\mathrm{Year\_a}}$ we can even directly compare the numerical results that prove to be very close (in spite of using a different matrix size). The matrix $A_{\mathrm{Year\_b}}$ provides the best results, $A_{\mathrm{Year\_a}}$ still achieves low trace relative errors. The error increases when considering A_{MNIST} and becomes very high when considering A_{Poly}.

In general, the relative error decreases when increasing either ℓ or k. As a matter of fact, a bigger ℓ allows us to get a better approximation of the original matrix and thus impact positively the relative error. Of course, an higher rank approximation improves the error.

Another aspect that can be considered in this plots is the difference of the error between the sketching matrices. In our case, it is close to none, since in A_{MNIST}, $A_{\text{Year\_a}}$, $A_{\text{Year\_b}}$ the graphs overlap almost perfectly. In A_{Poly}, we can notice a small difference, that is probably caused by an higher "volatility" due to the high trace relative error.

With regards to A_{Exp}, the relative error is quite low, although it follows a seemingly chaotic distribution. This is due to how A_{Exp} is defined (the entries on the diagonal go exponentially to 0). Since the singular values of A_{Exp} become very close to 0 very quickly, increasing k basically means that we want to approximate more 0s. For this reason we don't expect to see an improvement when increasing k.

4.3 Optimal ratio ℓ/k

Next, we inspect how the ratio ℓ/k affects the relative error. For this we considered the matrices $A_{\text{Year\_a}}$, A_{MNIST}, and A_{Poly}. In Figure 2 we see the relative error compared to the ratio ℓ/k for the different test matrices, sketching matrices (here we included also gaussian sketching), and multiple values of k. Each test has been executed once.

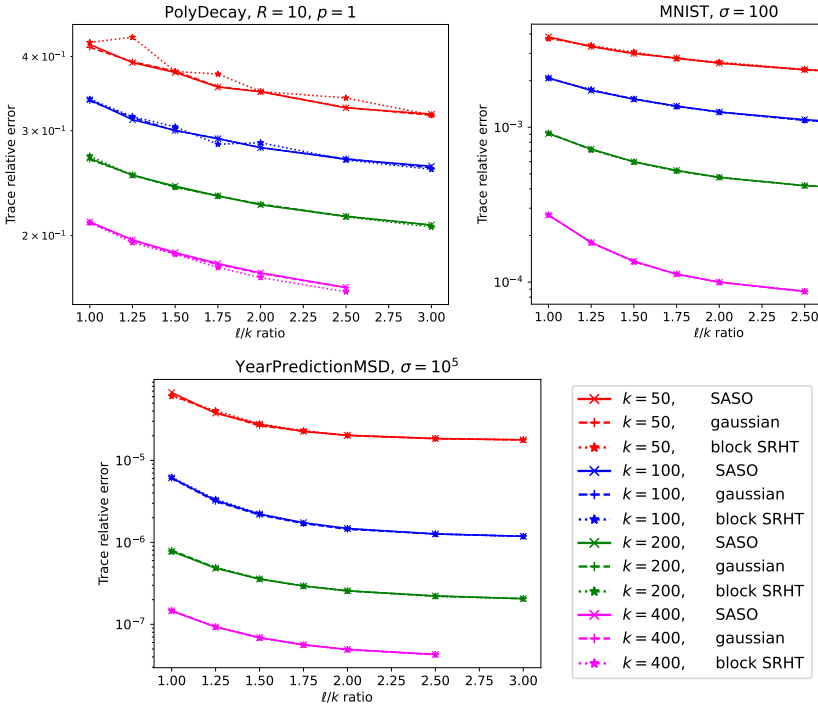


Figure 2

The type of sketching does not seem have a strong influence on the result. However, it seems clear that, while the approximation improves when increasing the ratio ℓ/k, the marginal gain decreases. As a result, choosing ℓ too large with respect to k might not make sense. In light of this observation, we decided to fix $\ell = 2k$ for the following numerical experiments.

4.4 Optimal number of non-zeros in SASO

Now we turn our attention to the numerical results obtained when using the SASO. This sketching operator allows us to choose the number of non-zero elements t in each row of Ω. We examine the relative trace error for different values of t. As described in Section 4.3, we fix $\ell = 2k$. We collect the data in Figure 3. Each value is obtained as an average of three tests.

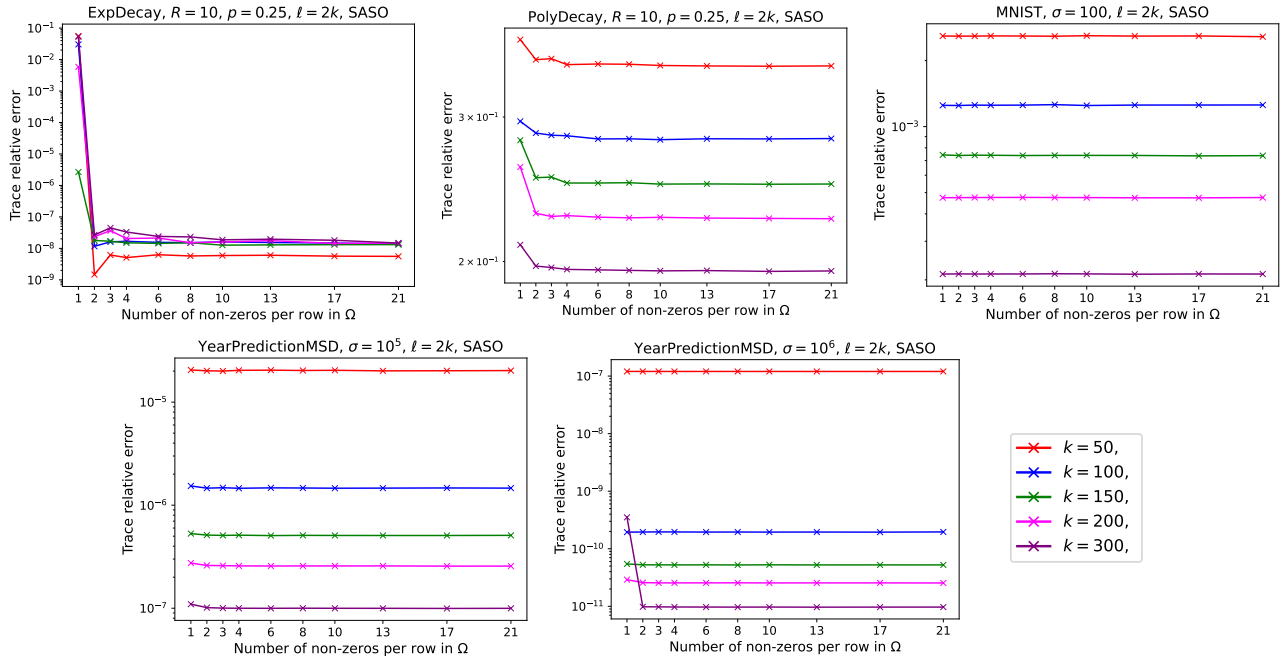


Figure 3

The datasets A_{Poly} and A_{Exp} show improvements when increasing t from very small values, to bigger ones, then the error quickly flattens out. For the other data sets we see no large improvements when increasing t. Only for $A_{\text{Year\_b}}$ there seems to be an outlier when $t = 1$ and $k = 300$. We reran the tests for this entry to make sure that this was not a statistical outlier, but a feature of the graph.

From our observation it is clear that, as mentioned in [4, Appendix A.2], eight non-zero elements per row are enough to get good performance. For this reason, we fixed $t = 8$ in all other tests with SASO.

4.5 Approximation of A's leading singular values

We conclude our numerical investigation by discussing if the Nyström algorithm is appropriate for approximating the leading k singular values of a matrix A. We compare the singular values obtained via Nyström algorithm (Σ_k^2) with the exact singular values of A obtained via SVD. We consider $k = 50,\ 100,\ 150$, with $\ell = 2k$ (as mentioned in Section 4.3), and we run 3 tests per setting per matrix using block SRHT sketching, for a total of 45 tests. In Figure 4, we plot the ratios between the approximated singular values and the exact ones.

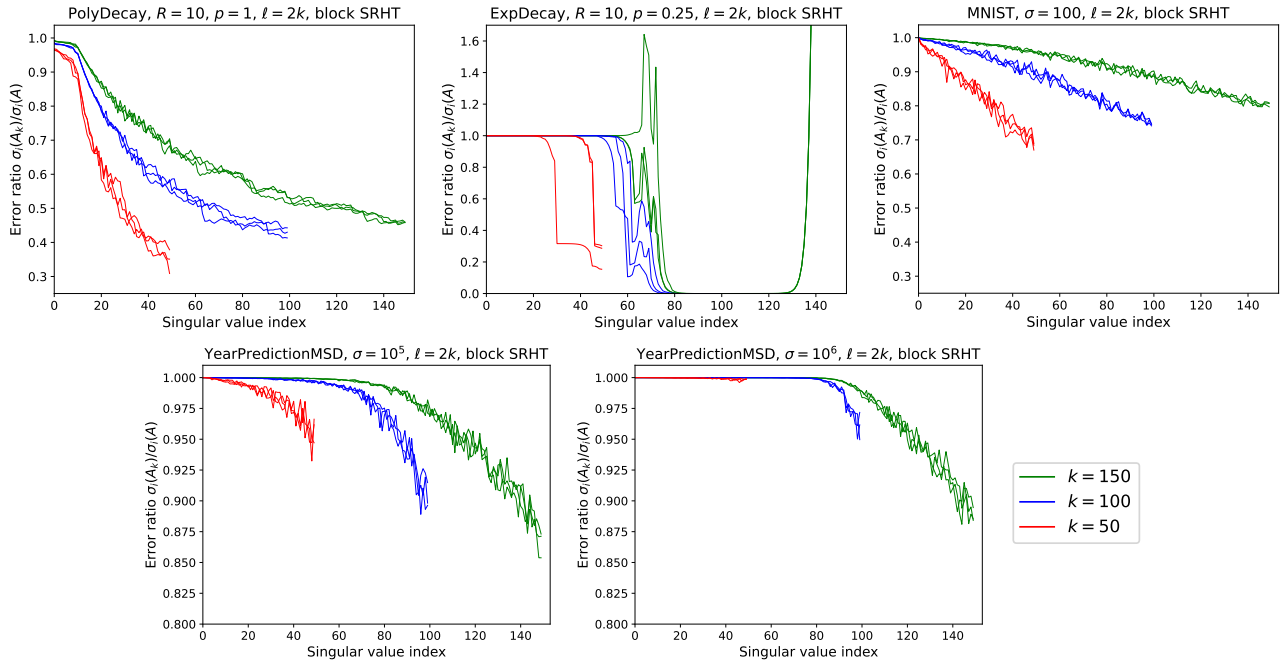


Figure 4

With regards to A_{Poly} and A_{Exp}, we notice how the Nyström algorithm does a good job in approximating the first singular values. By definition, the singular values of A_{Poly} and A_{Exp} decrease quite quickly after the 10-th. Thus it is very positive to observe an initial horizontal trait very close to 1 in the two graphs. Indeed, the first 10 singular values are the most interesting ones, as most of the others are very close to 0 anyway. In addition, we explain the chaotic behaviour in A_{Exp} by noticing that for this matrix the singular values decrease towards zero extremely fast. Therefore, computing the ratio will introduce the numerical "noise" we see in the plot.

With regards to A_{MINST}, we see how the quality of the approximation of the singular values decrease quite steadily the more the index increases. In $A_{\text{Year\_a}}$, and $A_{\text{Year\_b}}$, we notice a much better approximation of the singular values and a slower decrease in the quality, especially for $A_{\text{Year\_b}}$.

5 Runtime performance testing

In this section, we investigate the runtime performance of our parallel algorithm in comparison to the sequential one. We have executed an array of tests varying ℓ, k and the sketching matrix, both in sequential and parallel. Each setting has been executed 5 times, for a total of 800 tests, in Figure 5 we report the average runtimes for each setting.

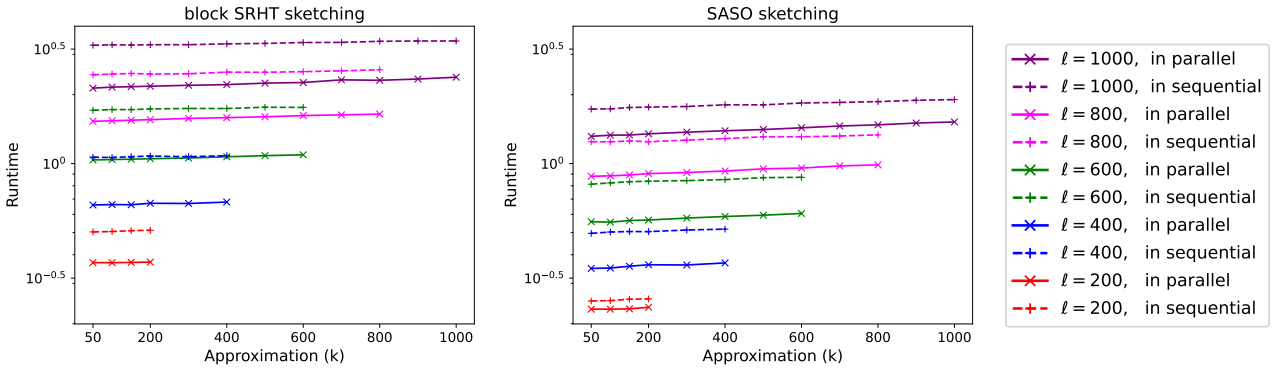


Figure 5

From our result, we notice how our parallel implementation is on average 45% faster than the sequential one, proving that we did a good job avoiding communication. On the other hand, we notice how our implementation of block SRHT is on average 40% slower than our implementation of SASO. It is probably possible to improve the performance for both the sketching matrices. In Appendix B, we discuss how the sketching with block SRHT could be implemented using a different method. Similarly, [4, Appendix A.2.1] explains how SASO can be efficiently applied to a matrix.

Another element we find important to notice is the relation between ℓ, k and runtime performance. In general, we can say that the runtimes increase both when increasing ℓ or k. However, ℓ seems to impact the runtimes much more. Indeed, the impact of k is almost negligible, if compared with the impact of ℓ. This translates in the almost horizontal traits in Figure 5. For this reason, in Figure 6, we plot the runtimes as a function of ℓ in logarithmic scale, fixing $k = 200$. From this last plot we can empirically infer that the runtimes are roughy proportional to ℓ.

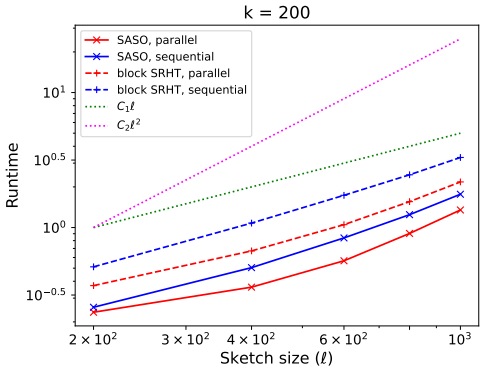


Figure 6

Appendix

A Extending our parallelization approach to odd powers of 2

In this section, we want to explore how our parallelization approach can be adapted when s is an odd power of 2. The results of this section were not implemented in Python. For the sake of simplicity, the analysis is done assuming to have only 8 processors ($s = 8$), nevertheless, the same reasoning can be generalized to any odd power of 2. We define $z = \frac{\log_2 s - 1}{2}$

We start by computing $C = A\Omega$. As in Section 3.2, we want to execute the multiplication block-wise as

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \\ A_{20} & A_{21} \\ A_{30} & A_{31} \end{pmatrix} \begin{pmatrix} \Omega_0 \\ \Omega_1 \end{pmatrix} = \begin{pmatrix} A_{00}\Omega_0 + A_{01}\Omega_1 \\ A_{10}\Omega_0 + A_{11}\Omega_1 \\ A_{20}\Omega_0 + A_{21}\Omega_1 \\ A_{30}\Omega_0 + A_{31}\Omega_1 \end{pmatrix}. \tag{4}$$

The processors are organized into a rectangular grid with dimensions $2^{z+1} \times 2^z$. Row and column indices for each processor are computed using the formulas: $row = rank \% 2^{z+1}$ and $col = rank / 2^{z+1}$, where $\%$ is the euclidean remainder operator. An example of conversion from $rank$ to row and col is the following.

0	4
1	5
2	6
3	7

$\longrightarrow$

0,0	0,1
1,0	1,1
2,0	2,1
3,0	3,1

We then distribute the matrix A as in Section 3.2. As a result, each processor owns a rectangular block of A,

$$A_{row\,col} \in \mathbb{R}^{\frac{n}{2^{z+1}} \times \frac{n}{2^z}}.$$

To execute multiplication 4, each processor executes one multiplication $A_{row\,col}\Omega_{col}$, generating Ω_{col} as explained in Section 3.3.

The results are then summed together row-wise using a `Reduce`. We use $row//2$ as root, where $//$ is the euclidean division. The blocks of C are stored in the processors as follows.

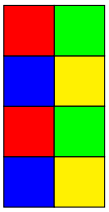
×	
×	
	×
	×

As in Section 3.2, we want to have the matrix C distributed among the processors using a one-dimensional block distribution, in order to execute the multiplication $B = \Omega^\top A\Omega = \Omega^\top C$ block-wise, as

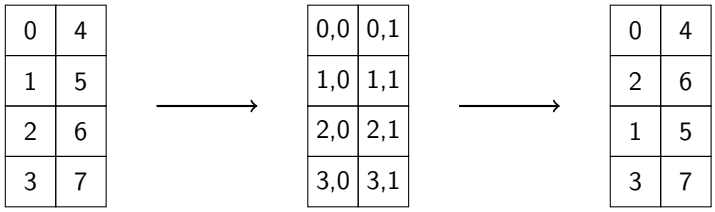
$$\begin{pmatrix} \tilde{\Omega}_0^\top & \tilde{\Omega}_1^\top & \tilde{\Omega}_2^\top & \tilde{\Omega}_3^\top & \tilde{\Omega}_4^\top & \tilde{\Omega}_5^\top & \tilde{\Omega}_6^\top & \tilde{\Omega}_7^\top \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \\ C_7 \end{pmatrix} = \tilde{\Omega}_0^\top C_0 + \tilde{\Omega}_1^\top C_1 + \tilde{\Omega}_2^\top C_2 + \tilde{\Omega}_3^\top C_3 + \tilde{\Omega}_4^\top C_4 + \tilde{\Omega}_5^\top C_5 + \tilde{\Omega}_6^\top C_6 + \tilde{\Omega}_7^\top C_7$$

where $\Omega_0 = \begin{pmatrix} \tilde{\Omega}_0 \\ \tilde{\Omega}_1 \\ \tilde{\Omega}_2 \\ \tilde{\Omega}_3 \end{pmatrix}$ and $\Omega_1 = \begin{pmatrix} \tilde{\Omega}_4 \\ \tilde{\Omega}_5 \\ \tilde{\Omega}_6 \\ \tilde{\Omega}_7 \end{pmatrix}$.

We want to scatter the blocks of C obtained in the previous step among the processors in the same column, however now there are two processors per column that own a block of C. The solution is to scatter among the processors that share the same row-parity. In the following drawing, we scatter among processors with the same color.



Thanks to this distribution, since Ω_{col} is already stored in the processors in the col-th column, we don't need to further move other matrices among the processors. However, the blocks of C are not stored in the processors in the correct order with respect to the rank. In other words, the processor $rank$ does not own the $rank$-th block of C. To solve this problem we need to "re-rank" the processors, in MPI4py this can be done for example by splitting the processors using 0 as color and using the new rank as key. The new rank can be computed as $newrank = col \cdot 2^{z+1} + row//2 + (row\%2) \cdot 2^z$. The following graphic shows the re-ranking progress.



After re-ranking, each processor executes one multiplication $\tilde{\Omega}_{newrank}^{\top} C_{newrank}$. The results are then summed all together using a `Reduce`, having processor 0 as root.

From this point, the algorithm can be executed as it is, just by replacing the old communicator and the old rank with the ones we have just defined.

B Fast implementation of block SRHT

In this section, we discuss an alternative method for executing the multiplication $A_{row\,col}\Omega_{col}$ in **Step 1** in Section 3.2, when using block SRHT. Instead of physically building the sketching matrix, we rely on the nice properties of the Fast Walsh–Hadamard transform. While we haven't implemented this method in Python, we believe it could allow a quite significant speed up. According to our tests, only building the sketch matrix takes up 30-40% of the total runtime.

We start by writing down the product we want to execute. For the sake of simplicity, we call the two blocks A_{block} and Ω_{block}, respectively. Here, $n = \text{ROWS}(A_{\text{block}})$, differently from the rest of the report.

$$A_{\text{block}}\Omega_{\text{block}} = \sqrt{\frac{n}{\ell}} A_{\text{block}} \tilde{D} H R D\,.$$

Using Python notation, we can execute this multiplication row by row as

$$(A_{\text{block}}\Omega_{\text{block}})[i,:] = \sqrt{\frac{n}{\ell}} \left(A_{\text{block}}[i,:]\tilde{D}\right) HRD,$$

where we recognise $\left(A_{\text{block}}[i,:]\tilde{D}\right) H$ as Fast Walsh–Hadamard transform of $A_{\text{block}}[i,:]\tilde{D}$. This can be computed very efficiently using Algorithm 5, thanks to a butterfly scheme. The algorithm has a time complexity of $O(n \log n)$.

After computing Fast Walsh–Hadamard transform, we can sample ℓ entries using H and flip some signs using D. This has to be repeated for every row of A_{block}. Note that it is not necessary to store the Hadamard transform of the entire matrix. Algorithm 6 formalizes this process. The variables `general_random_seed` and `col_random_seed` are the same as in Section 3.3.

After evaluating the time complexity of each step, we obtain a total time complexity of $O(n^2 \log n)$, that is in most cases better than $O(n^2 \ell)$ that we had before. The second multiplication in **Step 1** in Section 3.2, can be implemented following a similar reasoning.

Algorithm 5 Fast implementation of the Hadamard transform

function FAST_HT(v) ▷ v is a row vector
  $n \leftarrow \text{LENGTH}(v)$
  $h \leftarrow 1$
  while $h < n$ **do**
    for $i \leftarrow 0$ **to** $n - 2h$ **with step** $2h$ **do**
      for $j \leftarrow i$ **to** $i + h - 1$ **do**
        $a \leftarrow v[j]$
        $b \leftarrow v[j + h]$
        $v[j] \leftarrow a + b$
        $v[j + h] \leftarrow a - b$
      end for
    end for
    $v \leftarrow v/\sqrt{2}$
    $h \leftarrow 2h$
  end while
  return v
end function

Algorithm 6 Block SRHT sketching block generation

Input: Matrix A_{block}, integer ℓ, random seeds `general_random_seed`, `col_random_seed`
Output: res $= A_{\text{block}}\Omega_{\text{block}}$

$n \leftarrow \text{ROWS}(A_{\text{block}})$ ▷ Note that here n has a different meaning than in the rest of the report

Initialize random number generator `general_rng` with `general_random_seed`
Initialize random number generator `col_rng` with `col_random_seed`

Use `general_rng` to sample ℓ indices $h[i]$ with replacement in $1...n$
Use `col_rng` to sample $\tilde{d} \in \mathbb{R}^n$ and $d \in \mathbb{R}^\ell$ with independent Rademacher entries

Allocate res $\in \mathbb{R}^{n \times \ell}$
for $i \leftarrow 0$ **to** $n - 1$ **do**
  temp $\leftarrow$ FAST_HT($A_{\text{block}}[i, :] \cdot \tilde{d}$) ▷ Where $\cdot$ indicate the element-wise vector-vector multiplication
  res$[i, :] \leftarrow \sqrt{\frac{n}{\ell}}$(temp$[h] \cdot d$) ▷ Where $(\text{temp}[h])_j \equiv \text{temp}[h_j]\ \forall j$
end for

References

[1] O. Balabanov, M. Beaupere, L. Grigori, and V. Lederer, *Block subsampled randomized hadamard transform for low-rank approximation on distributed architectures*, 2022.

[2] T. Bertin-Mahieux, D. Ellis, B. Whitman, and P. Lamere, *The million song dataset.*, Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011), (2011), pp. 591–596.

[3] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-based learning applied to document recognition*, Proceedings of the IEEE, 86 (1998), pp. 2278–2324.

[4] R. Murray, J. Demmel, M. W. Mahoney, N. B. Erichson, M. Melnichenko, O. A. Malik, L. Grigori, P. Luszczek, M. Dereziński, M. E. Lopes, T. Liang, H. Luo, and J. Dongarra, *Randomized numerical linear algebra : A perspective on the field with an eye to software*, 2023.

[5] J. A. Tropp, A. Yurtsever, M. Udell, and V. Cevher, *Fixed-rank approximation of a positive-semidefinite matrix from streaming data*, Advances in Neural Information Processing Systems, 30 (2017).

[6] D. P. Woodruff, *Sketching as a tool for numerical linear algebra*, CoRR, abs/1411.4357 (2014).