Randomized Nyström

Siyuan Cheng, Kleon Karapas

MATH-505 HPC for numerical methods and data analysis

1 Introduction

For a large symmetric positive semi-definite (SPSD) matrices $A \in \mathbb{R}^{n\times n}$, the Nyström approximation provides an efficient way to capture essential information by

$$A_{Nys} = (A\Omega)(\Omega^\top A\Omega)^\dagger(\Omega^\top A),$$

where $\Omega \in \mathbb{R}^{n\times l}$ is the randomized sketching matrix.

In this project, we explore the k-rank approximation of A_{Nys} using two sketching techniques: Gaussian and Subsampled Randomized Hadamard Transform (SRHT).

2 Preliminaries

2.1 Rank-k Truncated Randomized Nyström

Given an SPSD matrix $A \in \mathbb{R}^{n\times n}$ and a randomized sketching matrix $\Omega \in \mathbb{R}^{n\times l}$, we can derive the following transformations for the Nyström approximation

$$\begin{aligned} A_{Nys} &= (A\Omega)(\Omega^\top A\Omega)^\dagger(\Omega^\top A) \\ &\overset{\text{(a)}}{=} C(LL^\top)^\dagger C^\top \overset{\text{(b)}}{=} QR(QR)^\top \overset{\text{(c)}}{=} QU\Sigma^2U^\top Q^\top, \end{aligned}$$

where (a) computes the Cholesky decomposition $LL^\top = \Omega^\top A\Omega$, (b) uses the QR factorization of $CL^{-\top}$, and (c) performs the SVD decomposition. Furthermore, using the rank-k SVD decomposition $R = U_k\Sigma_k V_k^\top$ in the last step gives the rank-k truncated Nyström approximation

$$[A_{Nys}]_k = QU_k\Sigma_k^2U_k^\top Q^\top.$$

Details for each step are further elaborated in Algorithm 1.

Algorithm 1 Randomized Nyström with Rank-k Truncation of A_{Nys}

Input: An SPSD matrix $A \in \mathbb{R}^{n\times n}$, sketching matrix $\Omega \in \mathbb{R}^{n\times l}$
Output: Rank-k approximation $[A_{Nys}]_k$

1: Compute $C = A\Omega$
2: Compute $B = \Omega^\top C$
3: Perform Cholesky factorization $B = LL^\top$ // or eigenvalue decomposition if Cholesky fails
4: Compute $Z = CL^{-\top}$
5: Perform QR factorization $Z = QR$
6: Compute truncated rank-k SVD for $R = U_k\Sigma_k V_k^\top$
7: Compute $\hat{U}_k = QU_k$
8: Compute $[A_{Nys}]_k = \hat{U}_k\Sigma_k^2\hat{U}_k^\top$

2.2 Subspace Embedding and Sketching Matrix

Definition. *An oblivious subspace embedding OSE(n, ε, δ) is a random matrix* $\Omega \in \mathbb{R}^{l\times m}$ *such that, for any n-dimensional subspace* $\mathcal{V} \subset \mathbb{R}^m$*, the following property hold with probability at least* $1-\delta$*:*

$$(1-\varepsilon)\|x\|_2^2 \le \|\Omega x\|_2^2 \le (1+\varepsilon)\|x\|_2^2, \quad \forall x \in \mathcal{V}. \tag{1}$$

With the sketching size l, the following two types of sketching matrices [1] are considered in our project:

1. **Gaussian Matrix**: Defined by $\Omega = \frac{1}{\sqrt{l}}G$, where $G \in \mathbb{R}^{l\times m}$ is a Gaussian random matrix whose entries are independent standard normal random variables. It satisfies the property (1) for OSE(n, ε, δ) with $l = O((n + \log(1/\delta))\epsilon^{-2})$.

2. **Subsampled Randomized Hadamard Transform** (SRHT) : Defined by the matrix $\Omega = \frac{m}{\sqrt{l}}RHD$, where $D \in \mathbb{R}^{m\times m}$ is a diagonal matrix of random signs, $H \in \mathbb{R}^{m\times m}$ is the normalized Walsh-Hadamard transform, and $R \in \mathbb{R}^{l\times m}$ is formed by randomly selecting l rows from HD. It satisfies the property (1) for OSE(n, ε, δ) with $l = O(\epsilon^{-2}(n + \log(m/\delta))\log(n/\delta))$.

   Additionally, it can be represented as column blocks for parallelization:

   $$\Omega = [\Omega_1\ \Omega_2\ \cdots\ \Omega_P],$$

   where each $\Omega_i = \sqrt{\frac{m}{Pl}}D_{L_i}RHD_{R_i}$, $D_{L_i} \in \mathbb{R}^{l\times l}$ and $D_{R_i} \in \mathbb{R}^{m/P\times m/P}$ are diagonal matrices with independent random signs, $H \in \mathbb{R}^{m\times m}$ is the normalized Walsh-Hadamard matrix, and $R \in \mathbb{R}^{l\times m/P}$ is a uniform sampling matrix.

2.3 Datasets

Two categories of datasets are utilized for the numerical experiments. The first one is a synthetic dataset from [2] consists of matrices exhibiting polynomial and exponential decay:

1. Polynomial Decay: $A = \mathrm{diag}(1, \ldots, 1, 2^{-p}, \ldots, (n-R+1)^{-p}) \in \mathbb{R}^{n\times n}$, with decay parameters p set to 0.5, 1, and 2.

2. Exponential Decay: $A = \mathrm{diag}(1, \ldots, 1, 10^{-q}, 10^{-2q}, \ldots, 10^{-(n-R)q}) \in \mathbb{R}^{n\times n}$, with decay rates q of 0.1, 0.25, and 1.

These matrices are analyzed at $R = 5, 10, 20$, with n specified in subsequent sections.

The second dataset contains several kernel matrices, defined as `mnist` and `year` for the MNIST and YearPredictionMSD datasets [3, 4]:

1. The MNIST dataset is first scaled to the range [0,1], and then the radial basis function $e^{-\|x_i-x_j\|^2/c^2}$ is applied with $c = 10^2$.

2. For the YearPredictionMSD dataset, the radial basis function is applied with $c = 10^4$ and $c = 10^5$.

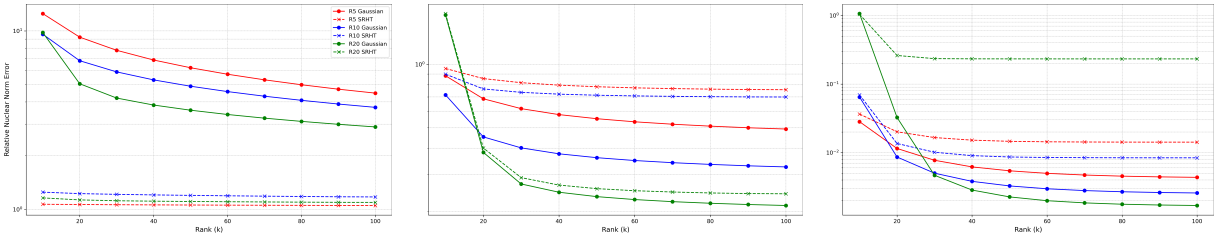
3 Numerical Stability

The metric for evaluating the numerical stability is the relative nuclear error

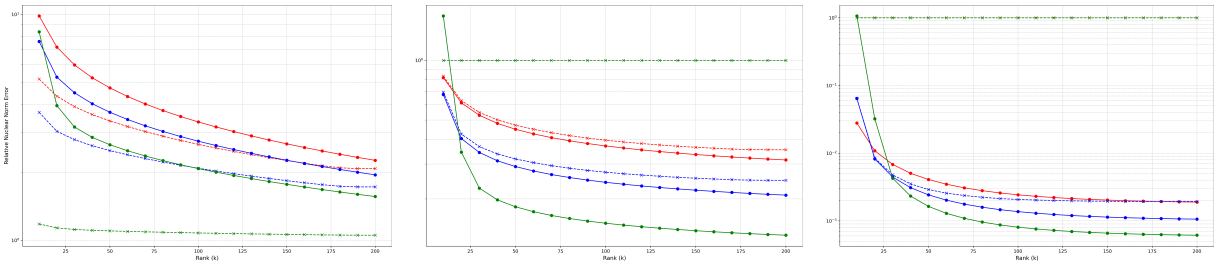
$$e = \frac{\|A - [A_{nys}]_k\|_*}{\|A\|_*},$$

where $\|A\|_* = \sigma_1(A) + \cdots + \sigma_n(A)$ is the nuclear norm.

3.1 Polynomial and Exponential Decay datasets

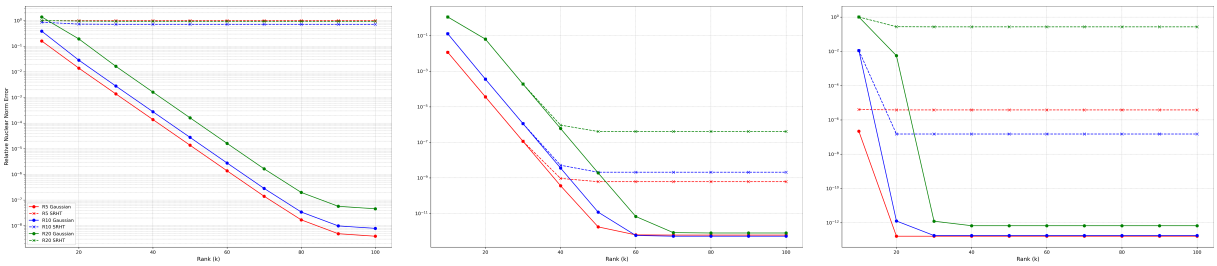


(a) `poly_decay_slow`, $l = 100$ (b) `poly_decay_med`, $l = 100$ (c) `poly_decay_fast`, $l = 100$

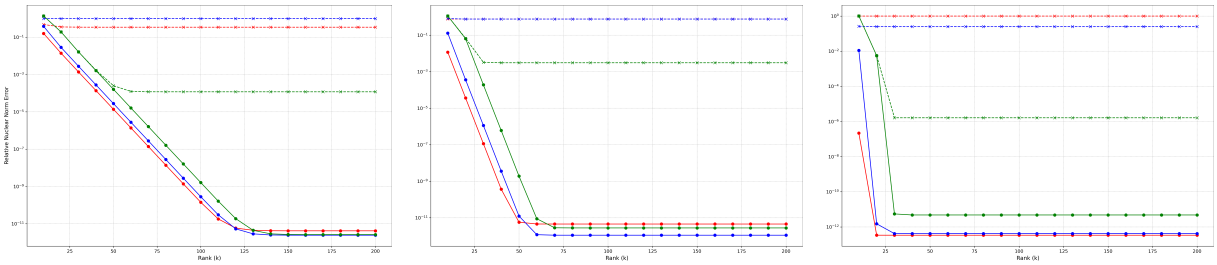


(d) `poly_decay_slow`, $l = 200$ (e) `poly_decay_med`, $l = 200$ (f) `poly_decay_fast`, $l = 200$

Figure 1: **Relative nuclear error as a function of k for the polynomial dataset.** The polynomial dataset is used with $p = 0.5$ (`slow`), $p = 1$ (`med`) and $p = 2$ (`fast`) and, $R = 5$ (`R5`), $R = 10$ (`R10`) and $R = 20$ (`R20`).



(a) `exp_decay_slow`, $l = 100$ (b) `exp_decay_med`, $l = 100$ (c) `exp_decay_fast`, $l = 100$



(d) `exp_decay_slow`, $l = 200$ (e) `exp_decay_med`, $l = 200$ (f) `exp_decay_slow`, $l = 200$

Figure 2: **Relative nuclear error as a function of k for the exponential dataset.** The exponential dataset is used with $q = 0.1$ (`slow`), $q = 0.25$ (`med`) and $q = 1$ (`fast`) and, $R = 5$ (`R5`), $R = 10$ (`R10`) and $R = 20$ (`R20`).

First, we start by noticing that as the decay coefficient (p for polynomial and q for exponential) increases, the error decreases significantly for smaller k: from the polynomial dataset, if we compare Fig. 1a with Fig. 1c an error of order 10^1 at $k = 20$ is achieved for $p = 0.1, R = 5, S =$ Gaussian, while for the same k and matrix we obtain an error the order of 10^{-2}. The same

conclusion can be made by comparing Fig. 2a, 2d and Fig. 2c, 2f for the exponential dataset. This trend reflects that faster decay rates lead to a better approximation since the matrix becomes more diagonally dominant, reducing the influence of off-diagonal terms. We also notice that increasing the *effective rank* R leads to larger errors for small k, for almost all tested matrices. Theoretically, we would expect that increasing R also requires a higher rank k to approximate the matrix effectively but we observe a different behavior. For the polynomial dataset, we see that by increasing R the error improves consistently for all the decay rates and schemes considered, while for the exponential dataset the opposite trend is observed. A possible explanation for this could be that in the polynomial case eigenvalues decay slowly (power-law behavior), meaning that when R increases, more eigenvalues are exactly 1. If we express the energy of the matrix as follows,

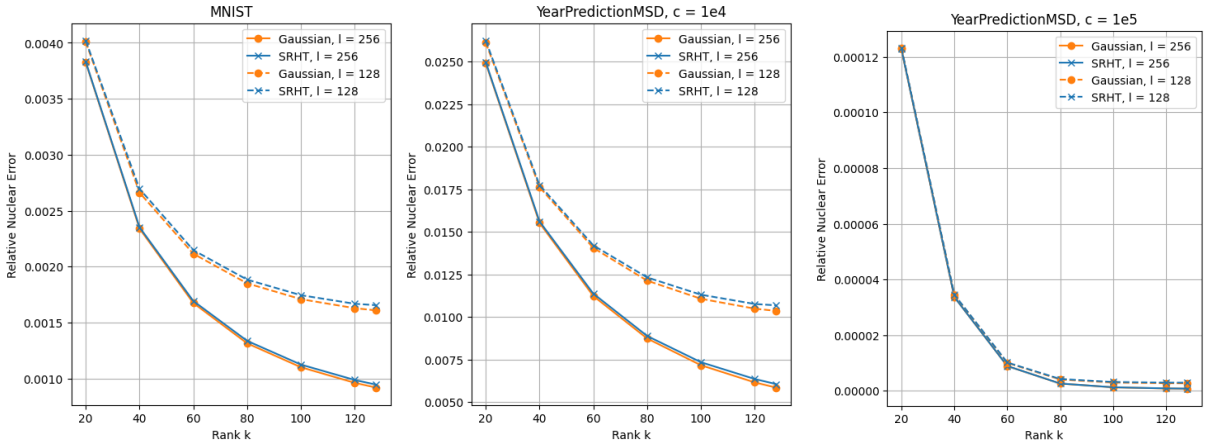
$$\text{Energy} = \sum_{i=1}^{R} 1 + \sum_{i=R+1}^{n} i^{-p}$$

we notice that when increasing R, a larger portion of the matrix's energy is concentrated in the first R eigenvalues, which allows k-rank approximations to capture more of the energy because the largest k eigenvalues dominate the spectrum. Now on the other hand, if we compare the energy contribution for the exponential decay:

$$\text{Energy} = \sum_{i=1}^{R} 1 + \sum_{i=R+1}^{n} 10^{-(i-R)q}$$

we see that for large R, the second term grows because more eigenvalues contribute non-negligible amounts, making it harder for k-rank approximations to capture enough energy. Gaussian subsampling consistently outperforms in terms of relative nuclear error for all matrices except Fig. 1a. Especially in the case of the exponential dataset, SRHT in many cases does not even converge or converges to it's best value for very small k. As expected, for the exponential dataset increasing l from 100 to 200 slightly improves the convergence of the relative error to smaller values for smaller k. On the other hand, for the polynomial dataset there is no significant improvement in the convergence, but choosing a higher l nonetheless allows us to also truncate the matrix at a higher k which usually yields a better approximation.

3.2 MNIST and YearPredictionMSD datasets



(a) Results for `mnist`. (b) Results for `year`, $c = 10^4$. (c) Results for `year`, $c = 10^5$.

Figure 3: **Relative Nuclear Error for Different k.** Results from the kernel $A \in \mathbb{R}^{4096\times4096}$ on two different datasets, with the sketching size of $l = 128, 256$.

Due to the significant computational cost of calculating the nuclear norm for large matrices, we constructed the kernel matrices using only the first 4096 rows of the MNIST and YearPredictionMSD datasets (see Section 2.3) for stability analysis. The results for corresponding `mnist` and `year` are shown in Figure 3.

We observe a decrease in the relative nuclear error as the rank k increases. This is because increasing rank k allows more information from A_{nys} to be retained, which leads to a better approximation of A. This enhancement comes from utilizing more components of the singular value decomposition to represent A, capturing more of its inherent structure.

Similarly, we note that the error decreases as the sketch size l increases. According to the theoretical conditions discussed in Section 2.2, larger sketch sizes l can facilitate Ω being an OSE with a smaller ε, which indicates the potential enhancement of the approximation quality. In the cases shown in Figures 3a and 3b, the Gaussian method performs slightly better than the SRHT under identical settings, demonstrating lower error values at equivalent ranks and sketch sizes. This performance difference could possibly stem from SRHT requiring a larger asymptotic bound for l to fulfill the OSE conditions. For instance, with $\delta = 0.01$ and $l = 128$, Ω meets the OSE criteria for the column space of `mnist` with $\varepsilon = 5$ for Gaussian and $\varepsilon = 10$ for SRHT. Meanwhile, at $l = 256$, these values improve to $\varepsilon = 2$ for Gaussian and $\varepsilon = 5$ for SRHT, indicating a better preservation of the subspace structure.

However, the benefits of increasing the sketch size l and the distinctions between the Gaussian and SRHT methods are less noticeable in Figure 3c. This kernel matrix, with its elements uniformly close to 1, is easily captured by randomized sketching. Both methods adequately approximate the matrix at smaller sketch sizes, making further increases in l ineffective at reducing error significantly.

4 Parallelization

Given an SPSD matrix $A \in \mathbb{R}^{n\times n}$, this section describes the parallelization techniques of the randomized Nyström Algorithm 1. We first apply a 2-D block parallelization structure to compute $C = A\Omega$. For example, when the number of processors $P = 4$, the formula for computing C can be written as

$$C = A\Omega = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} \Omega_1 \\ \Omega_2 \end{pmatrix} = \begin{pmatrix} \sum_j A_{1j}\Omega_j \\ \sum_j A_{2j}\Omega_j \end{pmatrix} = \begin{pmatrix} C_1 \\ C_2 \end{pmatrix}, \tag{2}$$

where each processor stores a block $A_{ij} \in \mathbb{R}^{n/\sqrt{P}\times n/\sqrt{P}}$, the sketching matrix Ω is segmented into row blocks $\Omega_j \in \mathbb{R}^{n/\sqrt{P}\times l}$, distributed among processors corresponding to blocks A_{ij}.

Once the row blocks C_j are computed, we proceed to calculate $B = \Omega^T A\Omega$ by aggregating the local results as follows

$$B = \Omega^\top A\Omega = \Omega C = \begin{pmatrix} \Omega_1^\top & \Omega_2^\top \end{pmatrix} \begin{pmatrix} C_1 \\ C_2 \end{pmatrix} = \sum_{j=1}^{\sqrt{P}} \Omega_j^\top C_j.$$

After this step, we can perform 1D parallelization to fully utilize the computation resources. With the Cholesky factorization $LL^\top = B$, the computation of $Z = CL^{-\top}$ can be distributed as

$$Z = \begin{pmatrix} Z_1 \\ Z_2 \\ Z_3 \\ Z_4 \end{pmatrix} = \begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix} L^{-\top} = \begin{pmatrix} C_1L^{-\top} \\ C_2L^{-\top} \\ C_3L^{-\top} \\ C_4L^{-\top} \end{pmatrix}.$$

Then we can perform TSQR on Z and compute the low-rank approximation of Nyström matrix. It should be noted that, due to constraints from the 2D-block distribution and TSQR, the number

of processors must be a square number that is also a power of two. The implementation details for parallel Nyström are described in Algorithm 2.

Algorithm 2 Parallel Nyström Algorithm with Rank-k Truncation of A_{Nys}

Require: $A \in \mathbb{R}^{n\times n}$, 2D-block-distributed over P processors, where P is a square number and a power of two.

function ParNyström(A)
  I = MyProcID() // from 1 to P
  Get communicators comm_row and comm_col by comm.Split( )
  Ir = MyRowID(), Ic = MyColID() // from 1 to $\sqrt{P}$
  comm_col.Scatterv(A), store as $Acol_I$
  comm_row.Scatterv($Acol_I$), store as A_I
  Ω_I = Generate_sketching(n, l, type, comm) // detailed in Algorithm 3
  Local computation $T_I = A_I\Omega_I$
  comm_col.Allreduce(T_I), store as C_I
  comm_row.Gather(C_I), store as C in row roots
  if $Ir = Ic$ **then**
    $B_I = \Omega_I^\top C_I$
  end if
  comm.Reduce(B_I), store as B in root
  Cholesky decomposition $LL^\top = C$ // LDL decomposition when Cholesky fails
  comm.Scatterv(C), store as C_I
  $Z_I = C_I L^{-\top}$ // solve the system without computing the inverse
  comm.Gather(Z_I), store as Z in root processor
  Q, R = TSQR(Z, comm) // use previously implemented TSQR
  Truncated SVD $U_k\Sigma_k V_k^\top = R$, compute $\hat{U}_k = QU_k$, $[A_{Nys}]_k = \hat{U}_k\Sigma_k^2\hat{U}_k^\top$
end function

For the 2D-block distribution of $A\Omega$, `MPI_Comm_split` is used to split the communicator into rows and columns. `MPI_Scatterv` and `MPI_Gather` are performed to distribute and collect the local matrices. Particularly, `MPI_Allreduce` is used to compute the row blocks C_I and align them with corresponding Ω_I. TSQR algorithm is applied with the binary tree structure described in Project 1. Additionally, the generation of local sketching matrix Ω_I is detailed in Algorithm 3.

Algorithm 3 Local Sketching Generator

Require: Dimension n, l, type of sketching matrix, communicator from Parallel Nyström

function Generate_sketching(n, l, comm)
  Split communicator and get I = MyProcID(), Ir = MyRowID(), Ic = MyColID() // Same as in 2
  for $i = 0$ **to** $\sqrt{P}$ **do**
    Generate R on root // only for block SRHT
    comm.Bcast(R)
    if Ic = i **and** Ir = 0 **then**
      Generate D_{L_I}, D_{R_I} // or Ω_I=Gaussian($n/\sqrt{P}$, l) for Gaussian sketching
    end if
    comm_row.Bcast(D_{L_I}, D_{R_I}) // or comm_row.Bcast(Ω_I)
  end for
end function

For Gaussian sketching matrices, a Gaussian random matrix is generated on each row root processor. These matrices are then synchronized across all processors within the same column using `MPI_Bcast`.

In the case of block SRHT sketching, the sampling indices for R is generated on the root processor and broadcast to all processors. Following this, diagonal elements for D_{L_I} and D_{R_I} are generated and broadcast in a manner analogous to the Gaussian sketching matrix. Each processor then can apply these components on the local A_I in the main algorithm.

5 Runtime Analysis

In our runtime analysis, both sequential and parallel approaches consistently utilize a matrix size of $n = 8192$ across all datasets detailed in Section 2.3. This size was selected as the maximum feasible value on the Helvetios cluster that prevents memory overflow while still meeting the operational and implementation requirements.

The computational experiments for sequential algorithm were conducted on a laptop with an Apple M2 Max chip and 32GB RAM. The following software and library versions were used:

	Python	NumPy	SciPy	mpi4py	Scikit-learn	PyTorch
Version	3.10.16	2.2.1	1.14.1	4.0.1	1.6.0	2.5.1

5.1 Sequential Runtime

Partial results for the sequential runtime on the synthetic dataset are presented in Figures 4 and 5, each showing three examples. Results for MNIST and YearPredictionMSD datasets are presented in 6. Additional results are available in Appendix A.

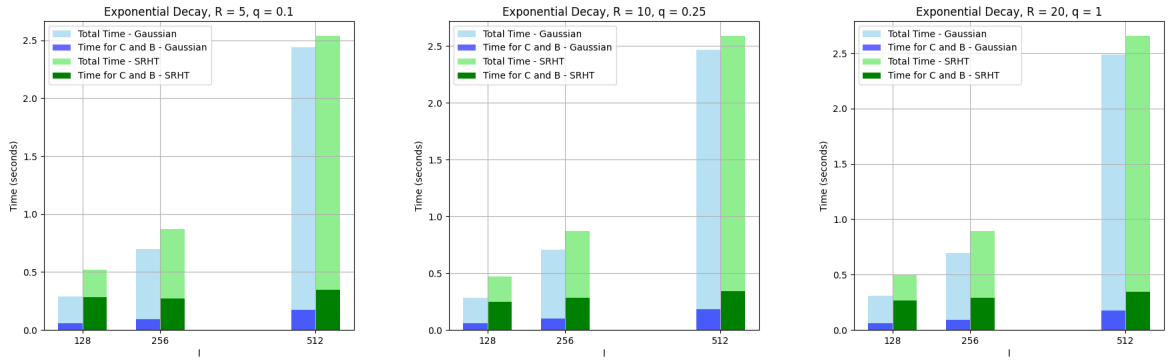


Figure 4: **Sequential runtime for different l on exponential decay matrices.** Each bar shows the total runtime for Nyström along with the time for computing $C = A\Omega$ and $B = \Omega^\top C$.

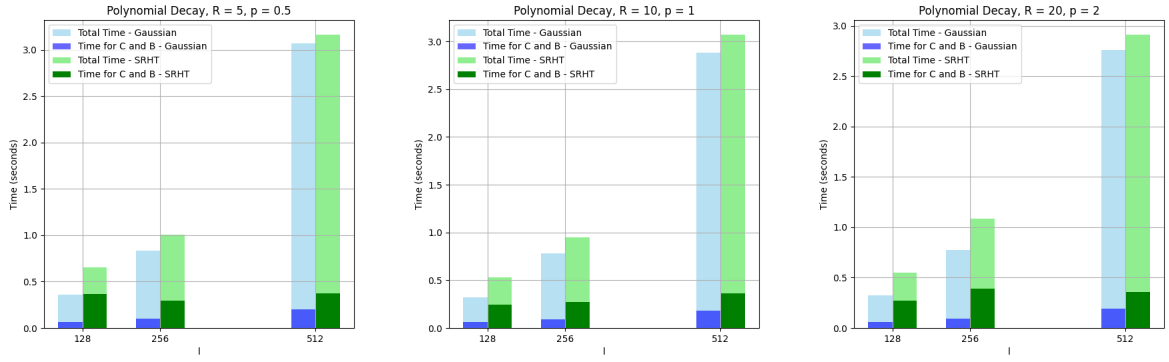
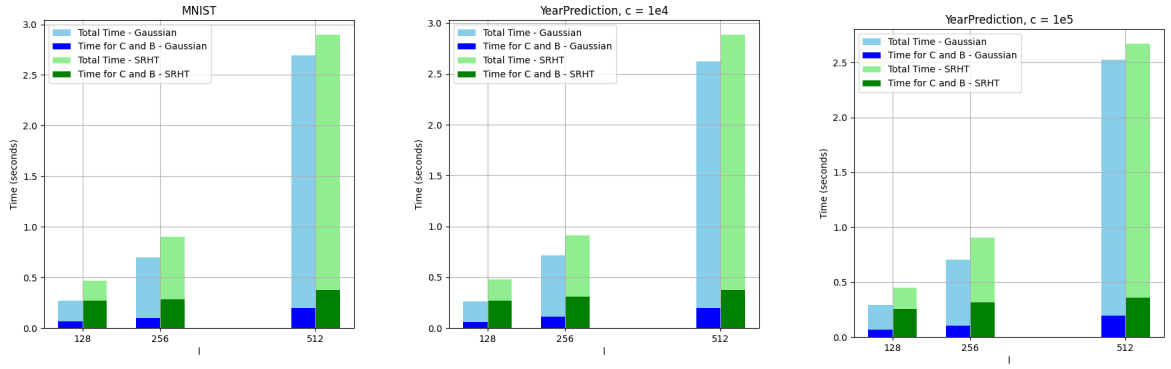


Figure 5: **Sequential runtime for different l on polynomial decay matrices.** Each bar shows the total runtime for Nyström along with the time for computing $C = A\Omega$ and $B = \Omega^\top C$.

Ideally, the SRHT should outperform the Gaussian sketching in terms of runtime, as SRHT operates in $O(n^2 \log n)$ flops for computing ΩA due to the efficiency of the Hadamard transform, while Gaussian requires $O(n^2 l)$ flops for matrix multiplication. However, practical implementations of SRHT in Python are hampered by the lack of direct support for matrix-based Hadamard



(a) Runtime for `mnist`. (b) Runtime for `year`, $c = 10^4$. (c) Runtime for `year`, $c = 10^5$.

Figure 6: **Sequential Runtime for Different l.** Each bar shows the total runtime for the Nyström approximation along with the time for computing $C = A\Omega$ and $B = \Omega^\top C$.

transforms. Initially, we attempted to use PyTorch's vector-based `hadamard_transform`, which requires multiple tensor-to-array conversions and looping over vectors, significantly increasing the computational overhead. This inefficiency led us to opt for explicit multiplication. Despite being theoretically slower, it was proved to be faster than the Hadamard-based approach in Python implementation.

Moreover, this explains why in the Gaussian approach, the computation time for $\Omega^\top A\Omega$ increases with l, while such variation is less evident in SRHT. In SRHT, the generation of the sketching matrix involves the matrix multiplication HD (if forming Ω explicitly) or DA (if applying the Hadamard transform directly), where D is a diagonal matrix of random signs. Theoretically, multiplying by D should be efficient, as it requires only flipping the signs of the columns in H or the rows in A. However, practical implementations in Python, even with vector broadcasting mechanisms, still approximate the complexity of matrix multiplication. Consequently, the computation of $\Omega^\top A\Omega$ in our implementation remains dominated by $O(n^3)$ complexities, which may diminish the visibility of changes attributable to varying l. Therefore, we observe that the computation times for $\Omega^\top A\Omega$ with SRHT consistently exceed those with Gaussian, and this gap decreases as l increases. This pattern is also shown in the results of our parallel runtimes.

Additionally, it is evident that the runtime for calculating $\Omega^\top A\Omega$ is relatively small due to the modest dimension sizes involved. Most of the computational time is concentrated on the $Z = CL^{-\top}$ step. This component has significant enhancements in parallel implementations.

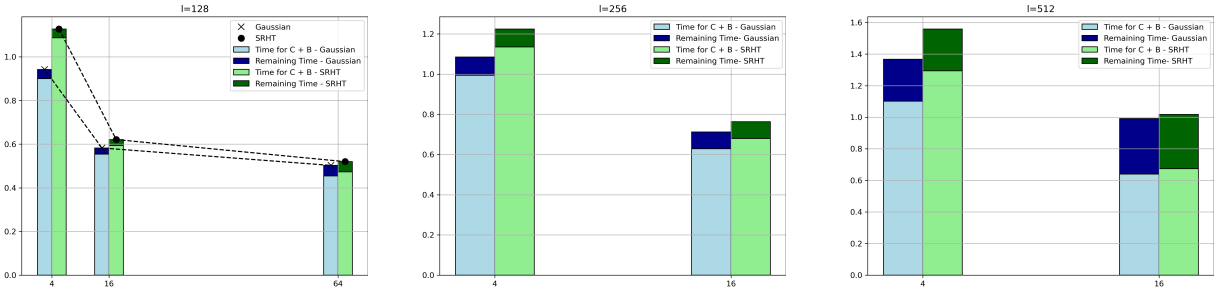
5.2 Parallel Performance

The parallel experiments on matrices of size $n = 8192$ were conducted on the Helvetios cluster using 4, 16, and 64 processors. Due to constraints from the matrix dimensions and TSQR, only tests involving $l = 128$ were conducted on 64 processors.

5.2.1 Polynomial and exponential decay dataset

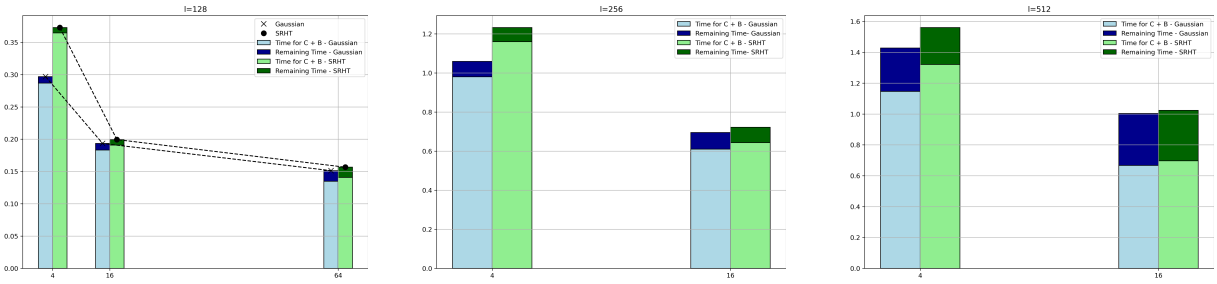
Partial results for parallel runtimes are shown in Figures 7 and 12. Examining Fig. 7a and Fig. 12a, we clearly see that for both datasets the runtime decreases as the number of processors increases from 4 to 16 and 64: the parallelization is therefore effective. However, the reduction in runtime is not linear with the increase of processors, especially noticeable for $l = 256$ (Figs. 7b, 7c) and $l = 512$ (Figs. 12c, 8c). This is due to the communication overhead between processors, which becomes more significant as the number of processors increases, and is amplified by the

fact that the dimension of the matrices is small ($n = 8192$) because of the constraints on memory previously mentioned. Here we see that there is a price to be paid for the better convergence results previously analyzed in Section 3.1 of the Gaussian methods, as we see that for both matrices and all the values of l, the runtimes are longer.



(a) Results for $l = 128$. (b) Results for $l = 256$. (c) Results for $l = 512$.

Figure 7: **Parallel runtime for `poly_decay_fast_R20`.** Each bar shows the total runtime and the dark part represents the runtime for computing $C = A\Omega$ and $B = \Omega^\top C$.

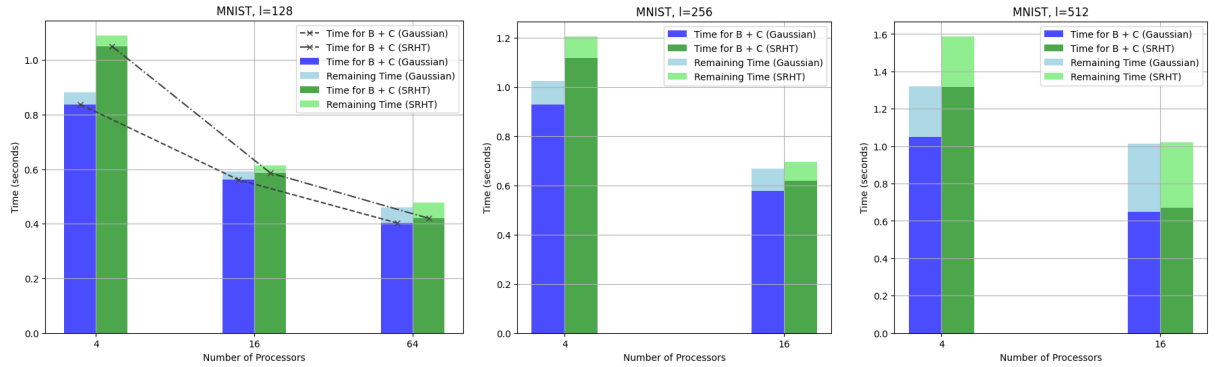


(a) Results for $l = 128$. (b) Results for $l = 256$. (c) Results for $l = 512$.

Figure 8: **Parallel runtime for `exp_decay_fast_R20`.** Each bar shows the total runtime and the dark part represents the runtime for computing $C = A\Omega$ and $B = \Omega^\top C$.

5.2.2 MNIST and YearPredictionMSD datasets

The parallel runtimes for `mnist` is depicted in Figures 14, results for `year` can be found in Appendix B.



(a) Results for $l = 128$. (b) Results for $l = 256$. (c) Results for $l = 512$.

Figure 9: **Parallel runtime for `mnist`.** Tests are conducted on $P = 4, 16$ and 64 due to implementation requirements. Each bar shows the total runtime and the dark part represents the runtime for computing $C = A\Omega$ and $B = \Omega^\top C$.

The results show a decrease in runtime with an increasing number of processors, confirming the capability of the parallel algorithm to effectively distribute computations across multiple processors. However, the effectiveness varies significantly with the size of l.

For $l = 128$, the parallel implementation shows minimal improvement over sequential execution. This is primarily due to the small computational workload associated with smaller values of l and n, where the communication costs of the parallel approach do not justify the overhead. Conversely, for $l = 512$, the benefits of parallelization become more apparent. With 4 processors, a 2x speedup is achieved, and with 16 processors, the speedup increases to 3x. The reduction in runtime is predominantly due to the decreased execution time for computing $Z = CL^{-\top}$, as shown by the recorded runtimes of each step.

Inheriting the implementation approach from the sequential method, the computation time for $\Omega^\top A\Omega$ using SRHT exceeds that of the Gaussian method. However, unlike the sequential runtime, the parallel runtime is predominantly dominated by the computation of $\Omega^\top A\Omega$. Even with increasing P, this portion of the runtime remains significantly higher than its sequential counterpart. This persistent overhead is largely attributed to the communication costs introduced by the 2D-block distribution and the use of `MPI_Allreduce` in our implementation. These factors suggest that the expected advantages of block parallelized Gaussian and SRHT are not fully achieved, indicating potential improvement in the implementation.

Additionally, the relative nuclear errors under the same settings as described in Section 3.2 are calculated. The results presented in Appendix C are very close to the corresponding results in Figure 3, suggesting that our parallel implementation maintains numerical stability.

6 Conclusion

In this project, we explore truncated randomized Nyström methods using Gaussian sketching and (block) SRHT. Both sketching methods achieve effective approximations in terms of relative nuclear error, with Gaussian observed to perform slightly better under the same dimensions. Despite not fully achieving SRHT's theoretical speed advantages, our implementation of parallel code with 2D-block partitioning demonstrates scalability with the increase in processors, proving the feasibility of computational optimization through parallel processing.

Acknowledgment

Generative AI was used in this report to assist in correcting grammar, improving the clarity of English, generating LaTeX formatting for figures and tables, and helping to debug.

References

[1] Lecture slides. *Lecture 7: Introduction to randomization and sketching techniques*. Math-505 HPC for numerical methods and data analysis.

[2] Joel Tropp et al. "Fixed-Rank Approximation of a Positive-Semidefinite Matrix from Streaming Data". In: (June 2017). DOI: 10.48550/arXiv.1706.05736.

[3] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[4] Thierry Bertin-Mahieux et al. "The Million Song Dataset." In: Jan. 2011, pp. 591–596.

A Plots for Sequential Runtime

Additional results for sequential runtime on polynomial decay matrices and exponential decay matrices are displayed in this section.

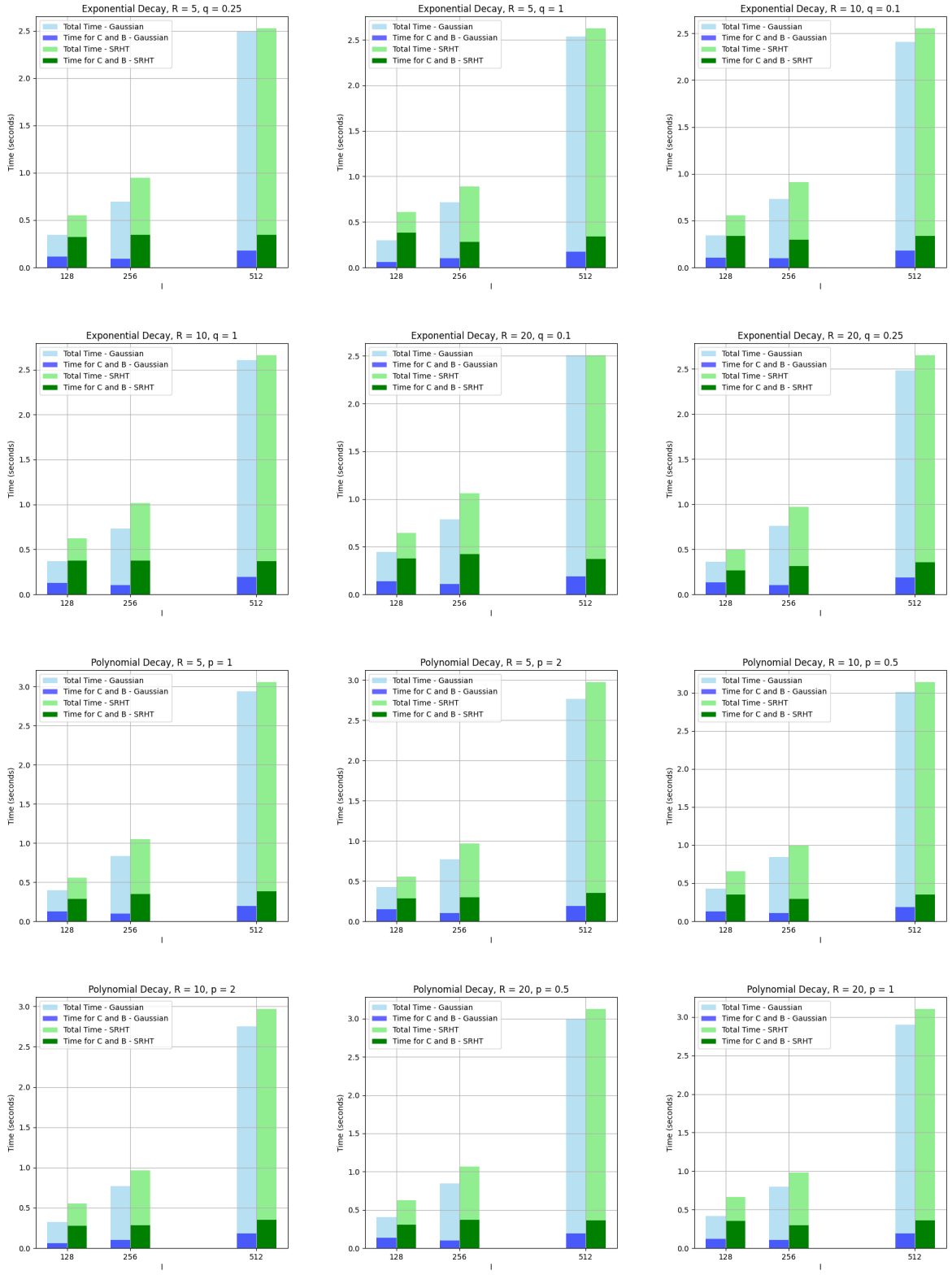
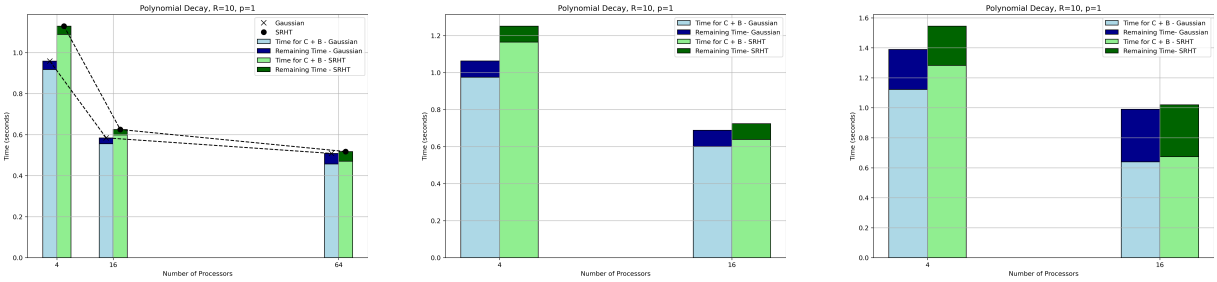


Figure 10: **Sequential runtime on exponential/polynomial decay matrices.**

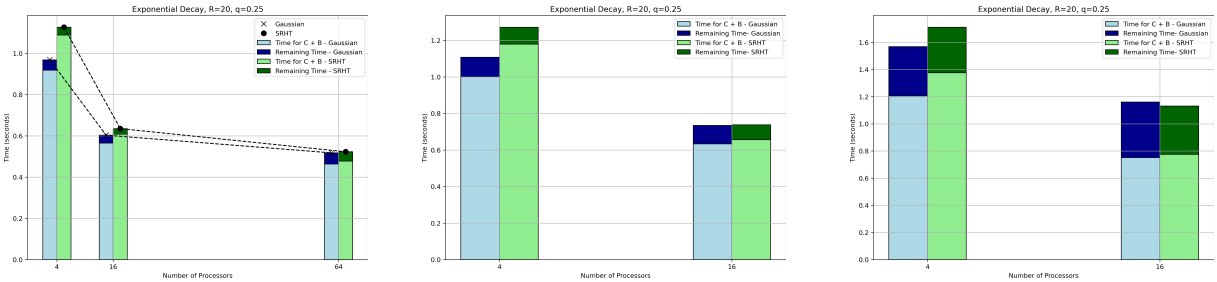
B Plots for parallel runtime

Additional results for parallel runtimes are shown in this section.



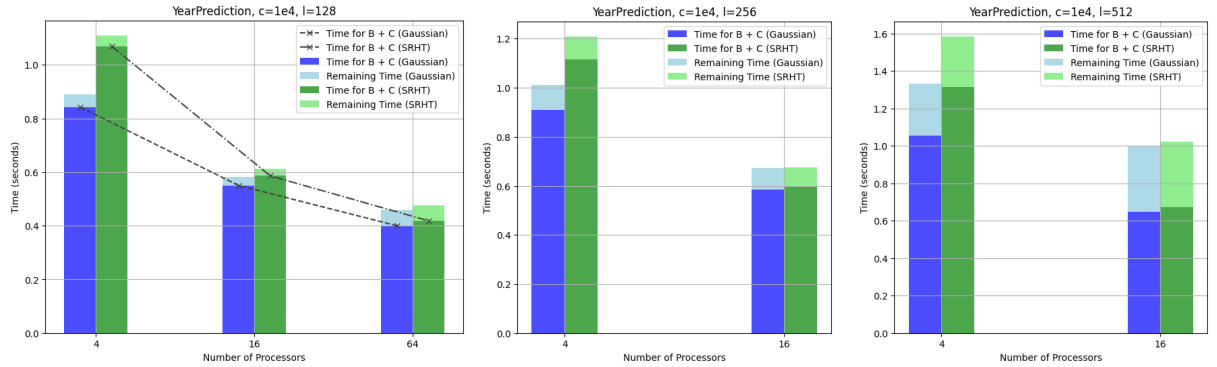
(a) Results for $l = 128$. (b) Results for $l = 256$. (c) Results for $l = 512$.

Figure 11: **Parallel runtimes for `poly_decay_med_R10`.** Each bar shows the total runtime and the dark part represents the runtime for computing $C = A\Omega$ and $B = \Omega^\top C$.



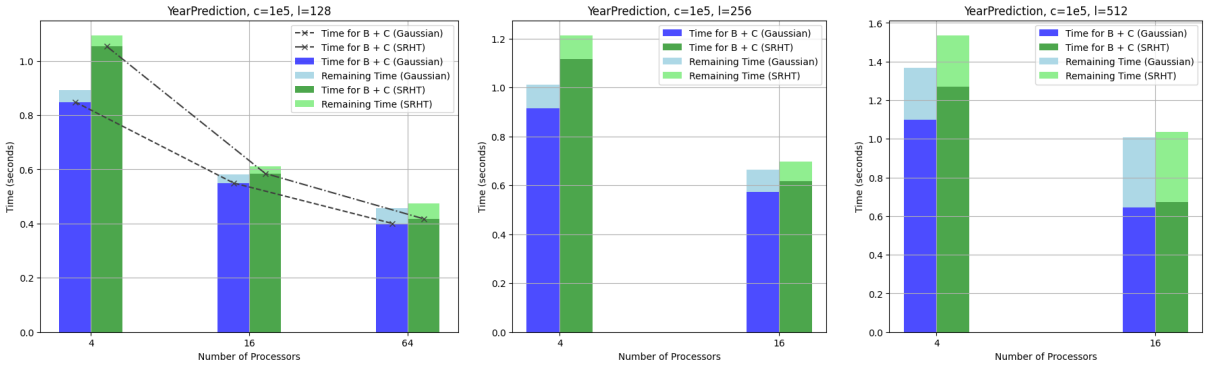
(a) Results for $l = 128$. (b) Results for $l = 256$. (c) Results for $l = 512$.

Figure 12: **Parallel runtimes for `exp_decay_med_R20`.** Each bar shows the total runtime and the dark part represents the runtime for computing $C = A\Omega$ and $B = \Omega^\top C$.



(a) Results for $l = 128$. (b) Results for $l = 256$. (c) Results for $l = 512$.

Figure 13: **Parallel runtimes for `year` with $c = 10^4$.**



(a) Results for $l = 128$. (b) Results for $l = 256$. (c) Results for $l = 512$.

Figure 14: **Parallel runtimes for `year` with $c = 10^5$.**

C Numerical stability for parallel Nyström

Numerical stability for parallel Nyström under the same setting as described in Section 3.2 is investigated in this section. Unlike the previous section, here we do not employ truncation but instead utilize the full Nyström approximation.

	$P = 4$, $l = 128$	$P = 4$, $l = 256$	$P = 16$, $l = 128$	$P = 16$, $l = 256$
Gaussian	1.62e-3	6.34e-4	1.61e-3	6.56e-4
Block SRHT	1.66e-3	7.14e-4	1.67e-3	7.16e-4

Table 1: **Nuclear error for parallel Nyström on `mnist`.** The results display a consistent pattern with those from the sequential approach, as shown in Figure 3. This demonstrates the numerical stability of the parallel algorithm.

	$P = 4$, $l = 128$	$P = 4$, $l = 256$	$P = 16$, $l = 128$	$P = 16$, $l = 256$
Gaussian	1.05e-2	4.12e-3	1.05e-2	4.06e-3
Block SRHT	1.10e-2	4.67e-3	1.04e-2	4.25e-3

Table 2: **Relative nuclear error for parallel Nyström on `mnist`.**

	$P = 4$, $l = 128$	$P = 4$, $l = 256$	$P = 16$, $l = 128$	$P = 16$, $l = 256$
Gaussian	2.75e-6	4.26e-7	2.68e-6	4.17e-7
Block SRHT	2.78e-6	1.57e-6	2.63e-6	4.46e-7

Table 3: **Relative nuclear error for parallel Nyström on `year1`.**