Report: Parallel Randomized Nyström Approximation

Emanuele Caruso, Davide Villani

1 Introduction

The primary objective of this study is to analyze the numerical stability of the Randomized Nyström approximation for low-rank matrix approximation. Given a Symmetric Positive Semidefinite (SPSD) matrix $A \in \mathbb{R}^{n\times n}$ and a sketching matrix $\Omega \in \mathbb{R}^{n\times l}$, the Nyström approximation is defined as:

$$A_{Nyst} = (A\Omega)(\Omega^T A\Omega)^+(\Omega^T A) \tag{1}$$

where $(\cdot)^+$ denotes the pseudoinverse.

In this formulation, Ω acts as the sketching matrix, reducing the dimensionality of the problem while preserving the dominant spectral properties of A. The algorithm is divided in the computation of the matrices $C = A\Omega$, with dimensions $C \in \mathbb{R}^{n\times l}$, and $B = \Omega^T A\Omega$, with dimensions $B \in \mathbb{R}^{l\times l}$. Consequently, the approximation can be expressed more compactly as $A_{Nyst} = CB^+C^T$.

The final goal is to produce the rank-k approximation of the matrix A_{Nyst}, where the target rank k is strictly less than the sketch size l ($k < l$). Direct truncation of the dense $n \times n$ matrix is computationally infeasible; therefore, we operate on the factorized components by performing a spectral decomposition and extracting the k-dominant spectral components. The final output is constructed as a low-rank factorization:

$$[[A_{Nyst}]]_k = \hat{U}_k\Sigma_k^2\hat{U}_k^T \tag{2}$$

In this notation, $\hat{U}_k$ represents the matrix of the top-k approximate eigenvectors of A, and Σ_k^2 is the diagonal matrix containing the corresponding top-k eigenvalues. .

1.1 Stable Rank-k Computation

Computing the rank-k truncation requires careful numerical handling, particularly when the core matrix $B = \Omega^T A\Omega$ is ill-conditioned. The standard approach in the literature relies on a Cholesky factorization of the core matrix ($B = LL^T$), followed by a QR decomposition of the whitened matrix $Z = CL^{-T}$ to obtain orthogonal factors.

However, this workflow depends heavily on B being Symmetric Positive Definite (SPD). In our experimental setting, B is strictly Symmetric Positive Semidefinite (SPSD). In such regimes, the Cholesky factorization becomes numerically unstable.

To ensure robustness, we first compute the eigenvalue decomposition of the core matrix, $B = V_B\Sigma_B V_B^T$ and then project the sampling matrix C onto the subspace defined by the dominant eigenvectors of B. This yields the factorized form:

$$[[A_{Nyst}]]_k = \hat{U}_k\Sigma_k\hat{U}_k^T \tag{3}$$

2 Parallelization of Randomized Nyström

To handle large datasets efficiently, we employ a distributed memory approach using MPI. The computation of A (kernel matrix), C, and B is parallelized across P processors.

2.1 Data Distribution

2.1.1 Distributed Construction of the Kernel Matrix

Theoretically, the kernel matrix A is distributed among processors using a two-dimensional block distribution. We assume a virtual processor grid of size $\sqrt{P} \times \sqrt{P}$. The matrix is partitioned into blocks A_{ij} corresponding to the interaction between the i-th row block and the j-th column block.

The experimental validation is performed using the `YearPredictionMSD` dataset. Following the standard protocol for kernel methods in this domain, we construct a dense symmetric positive semidefinite matrix $A \in \mathbb{R}^{n \times n}$ using a Radial Basis Function (RBF) kernel. The entries of the matrix are defined as

$$A_{ij} = \exp\left(-\frac{\|x_i - x_j\|_2^2}{c^2}\right), \tag{4}$$

where c is chosen in the range $[10^4, 10^5]$.

In practice, allocating the full dense matrix $A \in \mathbb{R}^{n \times n}$ is memory-prohibitive for large n. To circumvent this issue, we distribute the underlying dataset $X \in \mathbb{R}^{n \times d}$ rather than the kernel matrix itself. The root processor partitions the dataset X into $\sqrt{P}$ horizontal slices, denoted by $X_1, X_2, \ldots, X_{\sqrt{P}}$.

To construct the local block A_{ij}, the processor P_{ij} requires the i-th subset of data (acting as rows) and the j-th subset of data (acting as columns). Consequently, processor P_{ij} receives the pair (X_i, X_j) and computes the entries of A_{ij} on the fly using the kernel function:

$$A_{ij} = \kappa(X_i, X_j). \tag{5}$$

For instance, in a configuration with $P = 9$ processors (corresponding to a 3×3 grid), the matrix is structured as

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}. \tag{6}$$

2.2 Matrix Ω Generation

To efficiently construct the Nyström approximation, we employ a random sketching matrix to project the high-dimensional target matrix onto a lower-dimensional subspace while preserving its dominant spectral features. We introduce a sketching matrix $\Omega \in \mathbb{R}^{n \times \ell}$, with $\ell \ll n$, whose entries are drawn independently from a standard normal distribution,

$$\Omega_{ij} \sim \mathcal{N}(0, 1).$$

The sketch dimension ℓ controls the trade-off between computational cost and approximation accuracy. In our algorithm, the sketching matrix is used twice: first to compute $C = A\Omega$, and then to compute $B = \Omega^T C$. To parallelize these operations, the work is distributed among P processors, such that each processor owns a sketch block $\Omega_p \in \mathbb{R}^{\ell \times \frac{m}{P}}$. Processors are indexed by their rank r and mapped to a two-dimensional process grid like:
$i = r \,//\, P, \qquad j = r \bmod P.$
With this block decomposition, each processor computes the local contributions
$C_{i,j} = A_{i,j}\Omega_j, \qquad B_{i,j} = \Omega_i^T C_{i,j}.$
Consequently, each processor requires access to two sketching blocks, corresponding to its row and column indices.

Rather than generating the full sketching matrix on a single processor and redistributing it, an approach that would incur significant communication overhead, we exploit controlled random seeding to allow each processor to independently generate its required sketching blocks while maintaining global consistency.

Algorithm 1 Distributed Gaussian Sketch Generation

Require: ℓ, seed
1: seed $\leftarrow$ **Broadcast**(seed)
2: SetRandomSeed(seed $+i$)
3: $\Omega_i \leftarrow$ **Randn**(n_{local}, ℓ)
4: SetRandomSeed(seed $+j$)
5: $\Omega_j \leftarrow$ **Randn**(n_{local}, ℓ)

By doing so the $rank_0$ processor will broadcast the same seed to everyone and each rank will now compute two Ω and whenever $i = j$ also $\Omega_i = \Omega_j$ so in the end we will only actually generate $\sqrt{P}$ different sketch matrixes already correctly distributed among all processors without the need of any communication other than the base seed.

2.3 Parallel $C = A\Omega$

The parallel computation is structured to minimize communication volume by exploiting the block-decomposition of the matrix product $C = A\Omega$. To illustrate the logic, we consider the operations performed on a 2×2 grid (which generalizes to $\sqrt{P} \times \sqrt{P}$). The workflow is formally described in the following :

Algorithm 2 Parallel Block-Matrix Logic (Visualized for 2×2 Grid)

Matrix Expansion: Decompose the global product $C = A\Omega$ into blocks:

$$\begin{pmatrix} C_1 \\ C_2 \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} \Omega_1 \\ \Omega_2 \end{pmatrix} = \begin{pmatrix} \underbrace{A_{11}\Omega_1 + A_{12}\Omega_2}_{C_{11}} & \underbrace{}_{C_{12}} \\ \underbrace{A_{21}\Omega_1 + A_{22}\Omega_2}_{C_{21}} & \underbrace{}_{C_{22}} \end{pmatrix}$$

Following this logic, the actual execution proceeds in three main phases:

2. Local Block Multiplication (C_{local}) First, every processor P_{ij} operates independently to compute its partial contribution to the matrix C. Having generated the local kernel block A_{ij} and the corresponding sketch block Ω_j (as described in the data distribution section), the processor computes the local product $C_{ij} = A_{ij}\Omega_j$. This step is computationally dense and requires no communication.

3. Reduction of C To satisfy the summation equation $C_i = \sum_j C_{ij}$, a row-wise reduction is performed. Using `MPI_Reduce` across each row communicator, the local contributions $(C_{i1}, C_{i2}, \dots)$ are summed into a single block C_i stored on the root processor of that row.

4. Gathering C for Factorization Finally, to construct the low-rank factors $\hat{U}_k$, the full matrix C is required. At this stage, C exists as distributed blocks $C_1, C_2, \dots$ residing on the first column of the processor grid. We execute an explicit **Gather** operation to collect these blocks to the root processor. This reconstructs the tall matrix $C \in \mathbb{R}^{n \times l}$, enabling the final projection step.

2.4 Distributed Computation of Core Matrix B

The core matrix is defined as $B = \Omega^T A \Omega$. Since we have already computed the distributed blocks of $C = A\Omega$, we can compute B via the contraction $B = \Omega^T C$.

To maximize parallelism, we view this operation as a sum of local contributions from the entire processor grid. The logic is visualized in Algorithm 2, which expands the matrix product for a 2×2 grid configuration.

Algorithm 3 Parallel Computation of B (Visualized for 2×2 Grid)

1: **Matrix Contraction:** The global product $B = \Omega^T C$ is decomposed into grid interactions.

$$B = \begin{pmatrix} \Omega_1^T & \Omega_2^T \end{pmatrix} \begin{pmatrix} C_{11} + C_{12} \\ C_{21} + C_{22} \end{pmatrix}$$

By distributing the multiplication, we identify the individual terms computed by each processor:

$$B = \underbrace{\Omega_1^T C_{11}}_{B_{11}} + \underbrace{\Omega_1^T C_{12}}_{B_{12}} + \underbrace{\Omega_2^T C_{21}}_{B_{21}} + \underbrace{\Omega_2^T C_{22}}_{B_{22}}$$

Thus, the global matrix is simply the sum of all local blocks: $B = \sum_{i,j} B_{ij}$.

2: **Local Computation (B_{ij})** Each processor P_{ij} computes its specific contribution term:

$$B_{ij} \leftarrow \Omega_i^T \times C_{ij}$$

3: **Global Aggregation (Allreduce)** The final matrix is obtained by summing these local terms across all processors:

$$B_{global} \leftarrow \text{ALLREDUCE}(B_{11}, B_{12}, B_{21}, B_{22})$$

2.4.1 Local Contribution (B_{ij})

Each processor P_{ij} holds a specific block of the sketching matrix, Ω_i (associated with its row index in the grid), and the block C_{ij} it computed in the previous step. By multiplying these locally, $B_{ij} = \Omega_i^T C_{ij}$, the processor effectively calculates one term of the global summation. This operation is performed independently in parallel, with no communication required.

2.4.2 The Role of Allreduce

Since the target matrix B is the sum of all these distributed B_{ij} terms ($B = \sum_{i,j} B_{ij}$), we must aggregate data from every node in the grid. The `MPI_Allreduce` operation with the `SUM` operator performs this global addition. It

collects the $l \times l$ matrices from all processors, sums them element-wise, and broadcasts the final result back to everyone. This ensures that every processor possesses the exact full matrix B_{global}, enabling the subsequent stable factorization step to proceed locally on the root processor.

3 Computing Nyström Approximation

Once the distributed matrices C and B are computed and gathered (with C residing on the root processor and B available globally), the final step is to construct the low-rank factors. This procedure is encapsulated in Algorithm 3, which translates the numerical implementation into formal logic.

Algorithm 4 Rank-k Nyström Factorization

1: **Input:** Gathered matrix $C \in \mathbb{R}^{n \times l}$, Core matrix $B \in \mathbb{R}^{l \times l}$, Rank k
2: **Output:** Approx. eigenvectors $\hat{U}_k$, eigenvalues Σ_k
3: **Step 1: Spectral Decomposition of Core**
Compute eigenvalues and eigenvectors of the symmetric matrix B:

$$\Lambda, V_B \leftarrow \text{eigh}(B)$$

4: **Step 2: Feature Projection**
Compute the approximate eigenvectors of A by projecting C onto B's basis:

$$\hat{U} \leftarrow C \cdot V_B \cdot \text{diag}(\Lambda^{-1})$$

5: **Step 3: Rank-k Truncation**
Extract the top-k components to form the final approximation factors:

$$\hat{U}_k \leftarrow \hat{U}[:, 1:k], \quad \Sigma_k \leftarrow \Lambda[1:k]$$

The computation is designed to ensure numerical robustness against the rank-deficiency of kernel matrices. The process involves critical stages:

1. Spectral Decomposition of B Instead of inverting B directly (which is unstable), we perform an eigenvalue decomposition $B = V_B \Lambda V_B^T$. Since B is constructed as $\Omega^T A \Omega$, its spectrum captures the dominant information of the full kernel matrix projected into the sketch space.

2. Constructing the Approximate Basis ($\hat{U}$) The approximate eigenvectors of A, denoted as $\hat{U}$, are derived by mapping the sampling matrix C through the inverse spectral components of B. The implementation computes $\hat{U} = CV_B\Lambda^{-1}$, effectively "whitening" the sampled columns. This projection aligns the column space of C with the principal components of the underlying manifold.

4 Error analysis

To evaluate the quality of the approximation, we use the relative error in the nuclear norm. The nuclear norm $||M||_*$ is defined as the sum of the singular values of M.

$$||M||_* = \sigma_1(M) + \cdots + \sigma_n(M) \tag{7}$$

The error metric is given by:

$$\text{Error} = \frac{||A - [[A_{Nyst}]]_k||_*}{||A||_*} \tag{8}$$

The experiments were conducted on the YearPredictionMSD dataset. Figure 1 illustrates the decay of the relative error as a function of the target rank k, across different sketching sizes $l \in \{100, 400, 700, 1000\}$. As expected, the error decreases monotonically as the approximation rank k increases. Furthermore, increasing the sketch size l consistently improves the approximation quality. For instance, at $k = 50$, the configuration with $l = 1000$ yields a lower error compared to $l = 100$, demonstrating that a larger sampling space captures more spectral information from the original matrix.

Crucially, the algorithm maintains numerical stability even for higher ranks. This robustness is attributed to our implementation choice: rather than computing the Cholesky factorization of the core matrix B (which can be unstable for SPSD matrices), we utilized an eigenvalue decomposition approach ($B = V_B \Sigma_B V_B^T$) to construct the pseudo-inverse implicitly. This ensures valid approximations even when the kernel matrix spectrum decays rapidly.

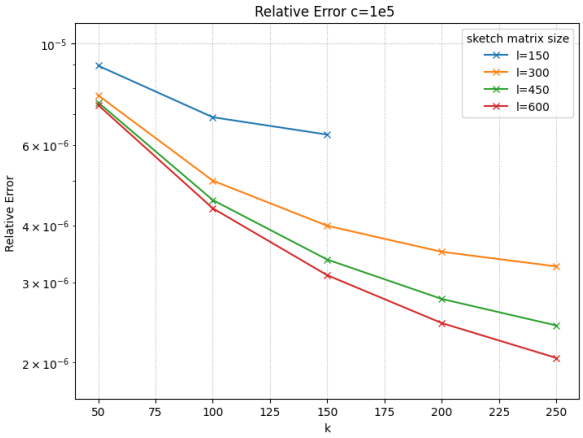

Figure 1: Relative nuclear norm error vs. target rank k for varying sketch dimensions l for $n = 15.000$.

5 Parallel & Sequential Performance

The performance of the distributed algorithm was evaluated on a remote machine using MPI up to 25 cores. We analyze both strong and weak scalability to assess the efficiency of the parallelization strategy.
It is important to note that all the results showed are relative only to the parallelized part, the computation of C and B, the k-rank approximation is not taken into consideration when timing the runs.

5.1 Strong Scaling

Strong scaling evaluates the speedup obtained by increasing the number of processors P while keeping the problem size fixed. Figure 2 reports the measured speedup for a matrix of size $n = 65{,}536$ as the number of cores increases from 1 to 25 (only perfect squares). The results shows strong scaling of the algorithm. This high parallel efficiency is primarily due to the dominance of the matrix multiplication step $C = A\Omega$, which is embarrassingly parallel and requires no inter-process communication.

Communication overhead is limited to the `MPI_Allreduce` operation used to assemble matrix B and the final `MPI_Gather` for collecting C. These costs remain negligible compared to the computational workload associated with kernel generation and matrix multiplication.

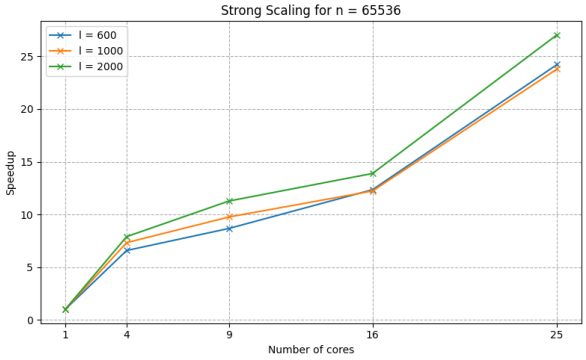

Figure 2: Strong Scaling: Speedup vs. Number of Cores for a fixed problem size $n = 65536$.

5.2 Weak Scaling

Weak scaling evaluates the system's ability to handle larger problem sizes as more computing resources are added. In Figure 3, we scale the problem size such that n increases with P.

The runtime plots show a slight increase as the number of cores grows, rather than remaining perfectly constant. This behavior is expected because while the local computation load ($A_{local} \times \Omega_{local}$) is kept roughly constant, the global reduction operations ($\sum C_{ij}$ and $\sum B_{ij}$) involve communication that scales logarithmically with P. Despite this, the runtime growth is modest (i.e. for $l = 600$, runtime increases from $\approx$ 3s to $\approx$ 5s as cores increase from 4 to 25), indicating that the algorithm effectively leverages distributed resources for large-scale datasets.

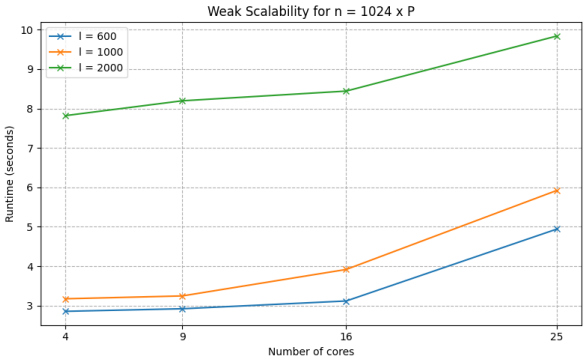

Figure 3: Weak Scalability: Runtime (seconds) vs. Number of Cores where the problem size n scales with P.