Università degli Studi di Padova

Dipartimento di Fisica e Astronomia Galileo Galilei

Direct QR factorizations for tall-and-skinny matrices in MapReduce architectures

Austin R. Benson, David F. Gleich, James Demmel

IEE International Conference on Big Data (2013)

Management and Analysis of Physics Datasets – B

Prof. Jacopo Pazzini

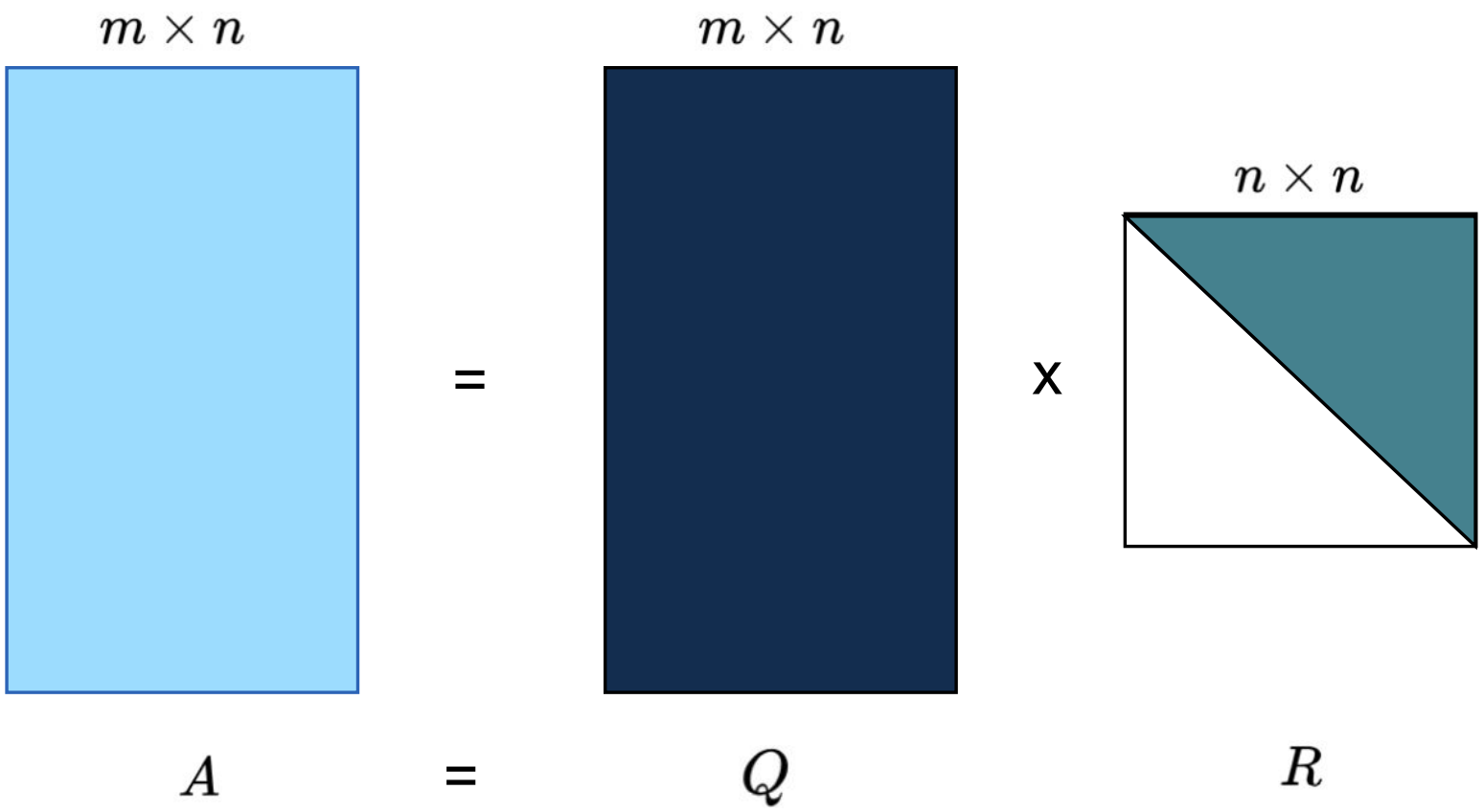
Riccardo Corte
Alessandro Miotto
Lorenzo Rizzi

Introduction: QR decomposition

QR Decomposition expresses a matrix A as A=QR, with Q orthogonal and R upper triangular.

It is key in solving linear systems, least squares, and numerical methods

When a matrix has many more rows than columns, it is called tall-and-skinny. Fast and accurate QR decomposition is crucial in data science, where datasets often contain many records but relatively few features.

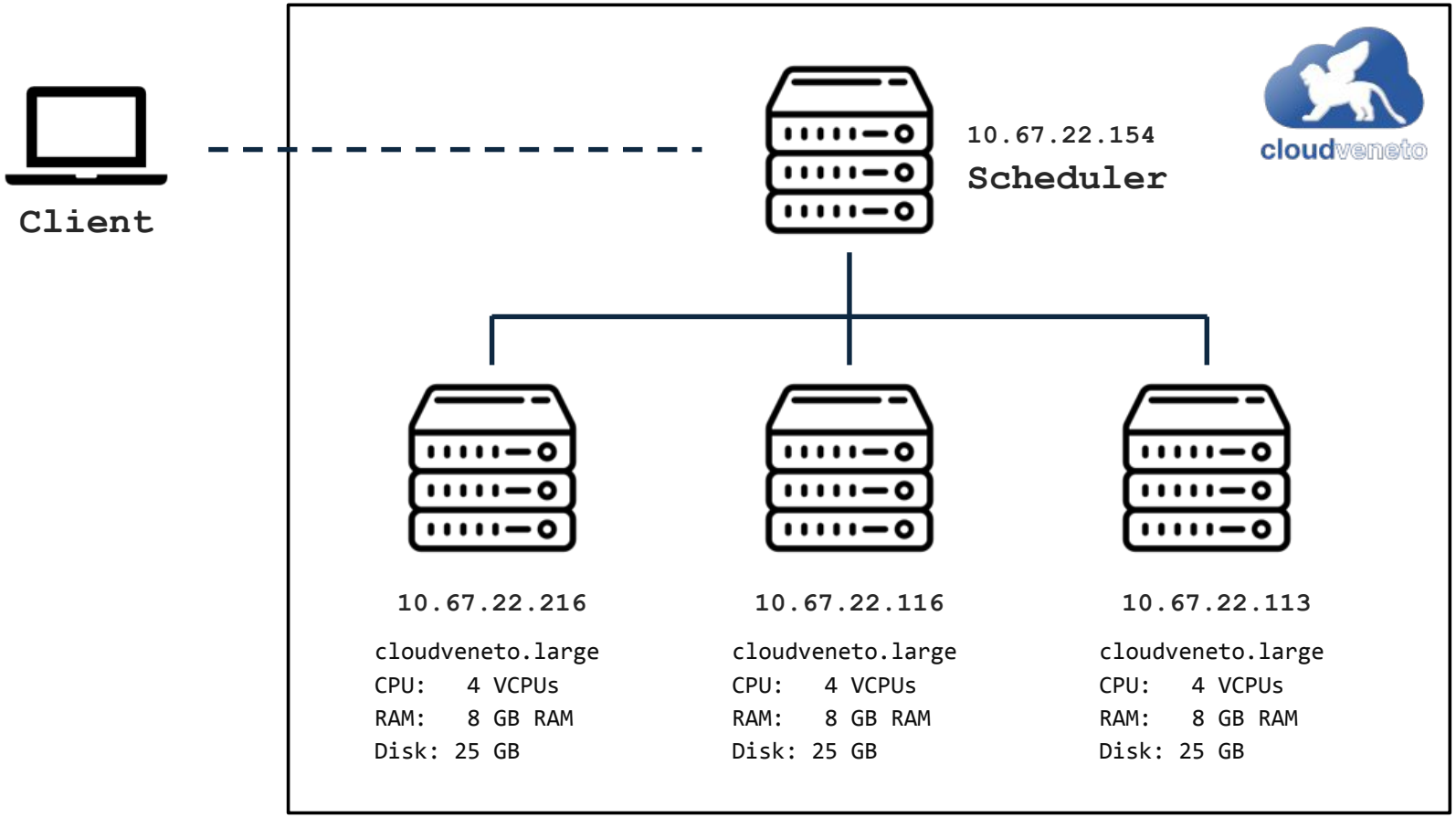


A tall-and-skinny matrix ($m \gg n$)

An orthogonal matrix ($Q^T Q = 1$)

An upper triangular matrix

Distributed system setup: CloudVeneto



We will implement three algorithms* (from least to most accurate) for performing QR decomposition in parallel on distributed systems using Dask

dask

(*) Direct QR factorizations for tall-and-skinny matrices in MapReduce architectures
(Austin R. Benson, David F. Gleich, James Demmel)

METHOD 1:

Cholesky TSQR

./Cholesky.ipynb

Cholesky TSQR

Let B be a symmetric matrix. Its Cholesky decomposition reads

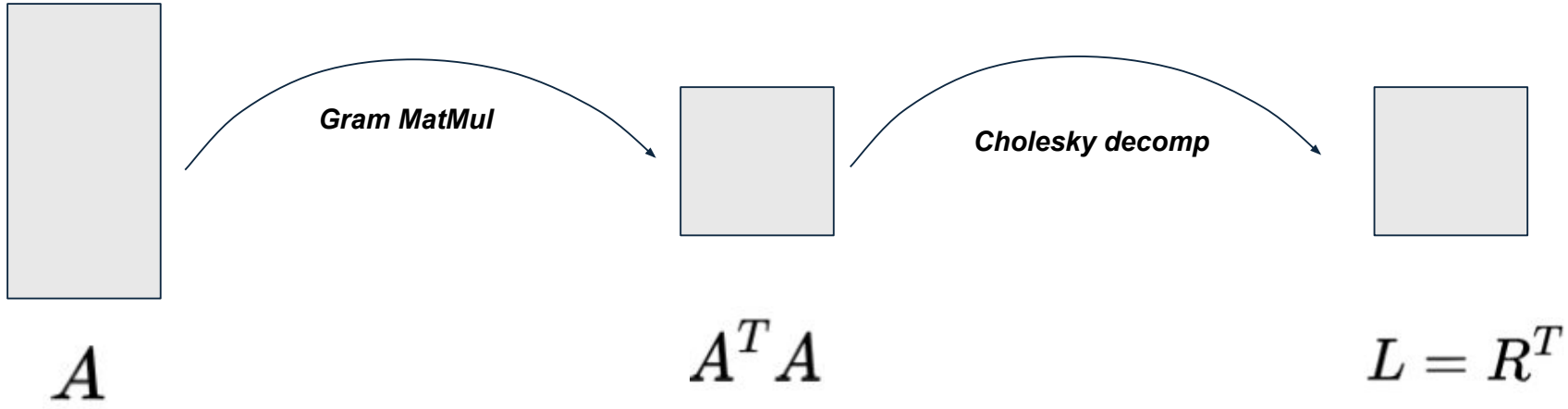
$$B = LL^T$$

Let's get back to QR. If A is a generic matrix, its QR decomposition is

$$A = QR$$

Define the Gram matrix ($G = A^T A$) so that:

$$G = A^T A = (R^T Q^T Q R) = R^T R = (R^T)(R^T)^T = LL^T$$



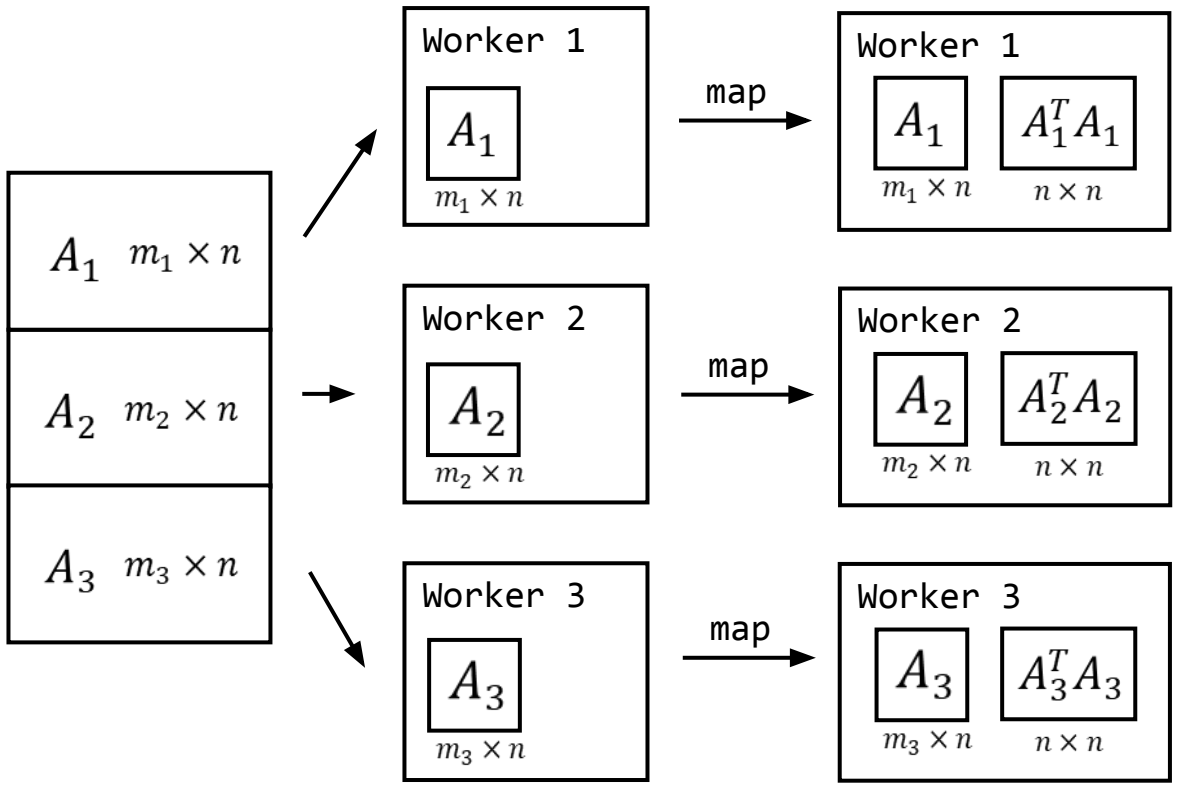
Once we have R, compute $Q = AR^{-1}$

Cholesky TSQR

Let's go parallel

```
./functions.py

def cholesky_tsqr(X_da : dask.array.Array):

    chunks_delayed = [dask.delayed(gramMatMul)(chunk)
                      for chunk in X_da.to_delayed().ravel()]

    Gram_global_delayed = dask.delayed(sum)(chunks_delayed)
    R = dask.delayed(np.linalg.cholesky)(Gram_global_delayed)

    R_inv = dask.delayed(Inverse)(R)

    Q = X_da.map_blocks(MatMul, R_inv, dtype=X_da.dtype)

    return Q, R
```

Step 1: Gram matrix

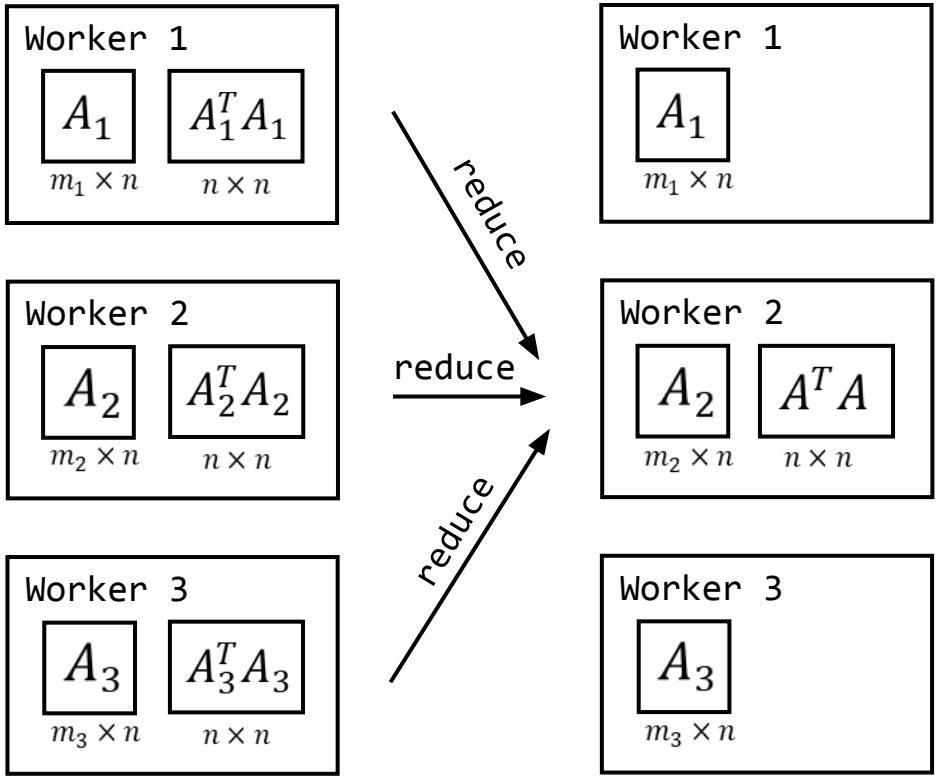


Cholesky TSQR

./functions.py

```python
def cholesky_tsqr(X_da : dask.array.Array):

    chunks_delayed = [dask.delayed(gramMatMul)(chunk)
                      for chunk in X_da.to_delayed().ravel()]

    Gram_global_delayed = dask.delayed(sum)(chunks_delayed)
    R = dask.delayed(np.linalg.cholesky)(Gram_global_delayed)

    R_inv = dask.delayed(Inverse)(R)

    Q = X_da.map_blocks(MatMul, R_inv, dtype=X_da.dtype)

    return Q, R
```

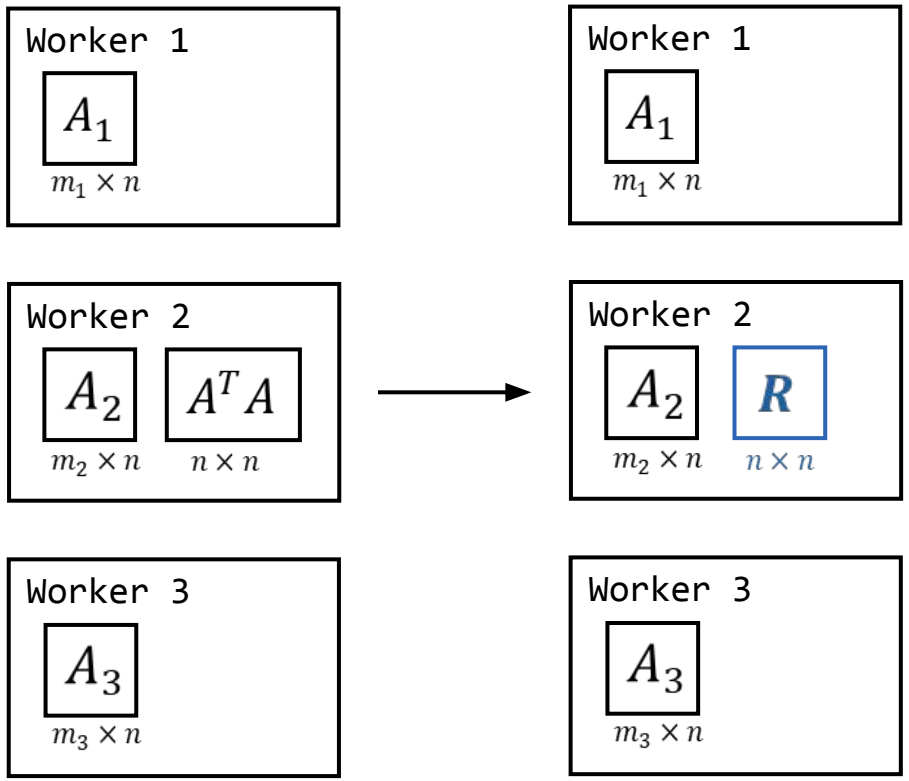
Step 1: Gram matrix



Cholesky TSQR

```
./functions.py

def cholesky_tsqr(X_da : dask.array.Array):

    chunks_delayed = [dask.delayed(gramMatMul)(chunk)
                      for chunk in X_da.to_delayed().ravel()]

    Gram_global_delayed = dask.delayed(sum)(chunks_delayed)
    R = dask.delayed(np.linalg.cholesky)(Gram_global_delayed)

    R_inv = dask.delayed(Inverse)(R)

    Q = X_da.map_blocks(MatMul, R_inv, dtype=X_da.dtype)

    return Q, R
```

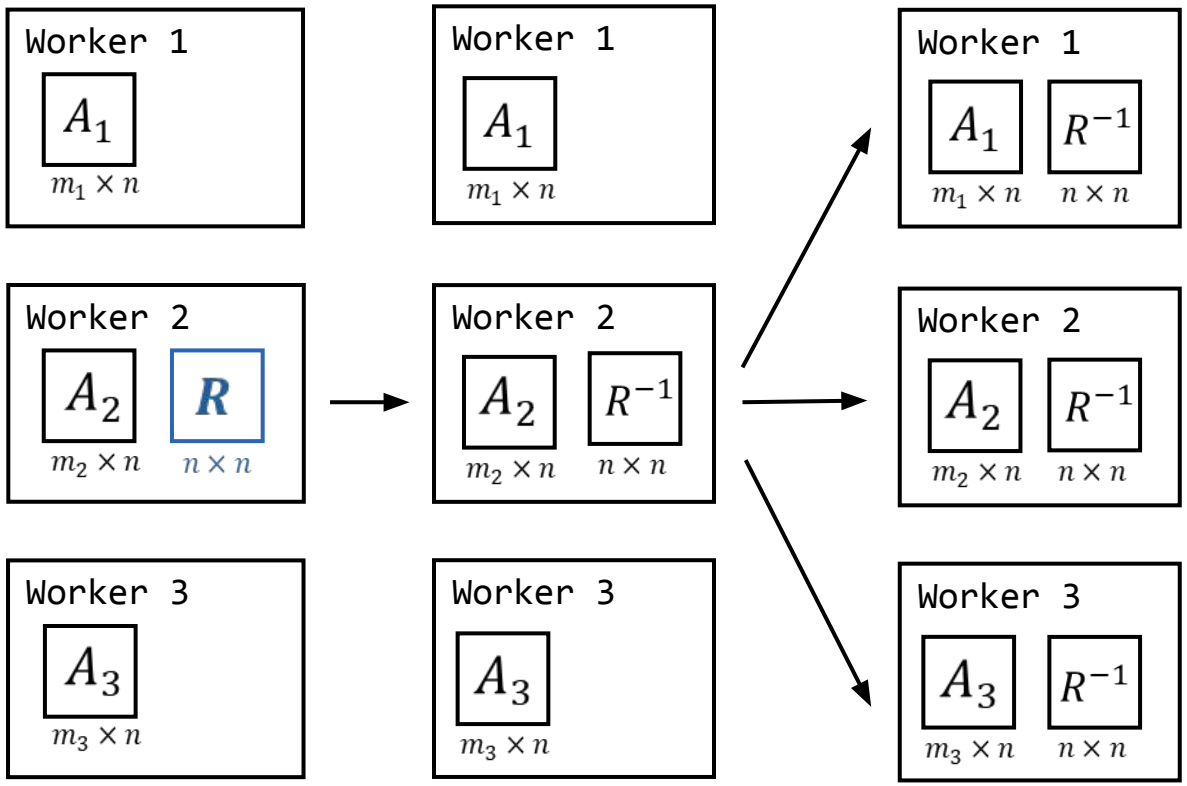
Step 2: Cholesky decomposition



Cholesky TSQR

```
./functions.py

def cholesky_tsqr(X_da : dask.array.Array):

    chunks_delayed = [dask.delayed(gramMatMul)(chunk)
                      for chunk in X_da.to_delayed().ravel()]

    Gram_global_delayed = dask.delayed(sum)(chunks_delayed)
    R = dask.delayed(np.linalg.cholesky)(Gram_global_delayed)

    R_inv = dask.delayed(Inverse)(R)

    Q = X_da.map_blocks(MatMul, R_inv, dtype=X_da.dtype)

    return Q, R
```

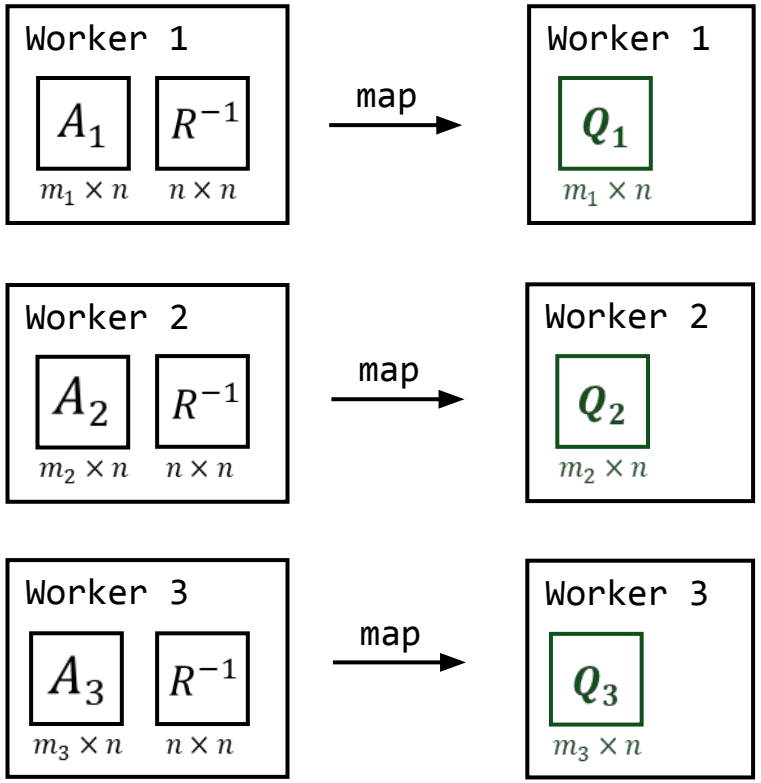
Step 2: Recovering Q



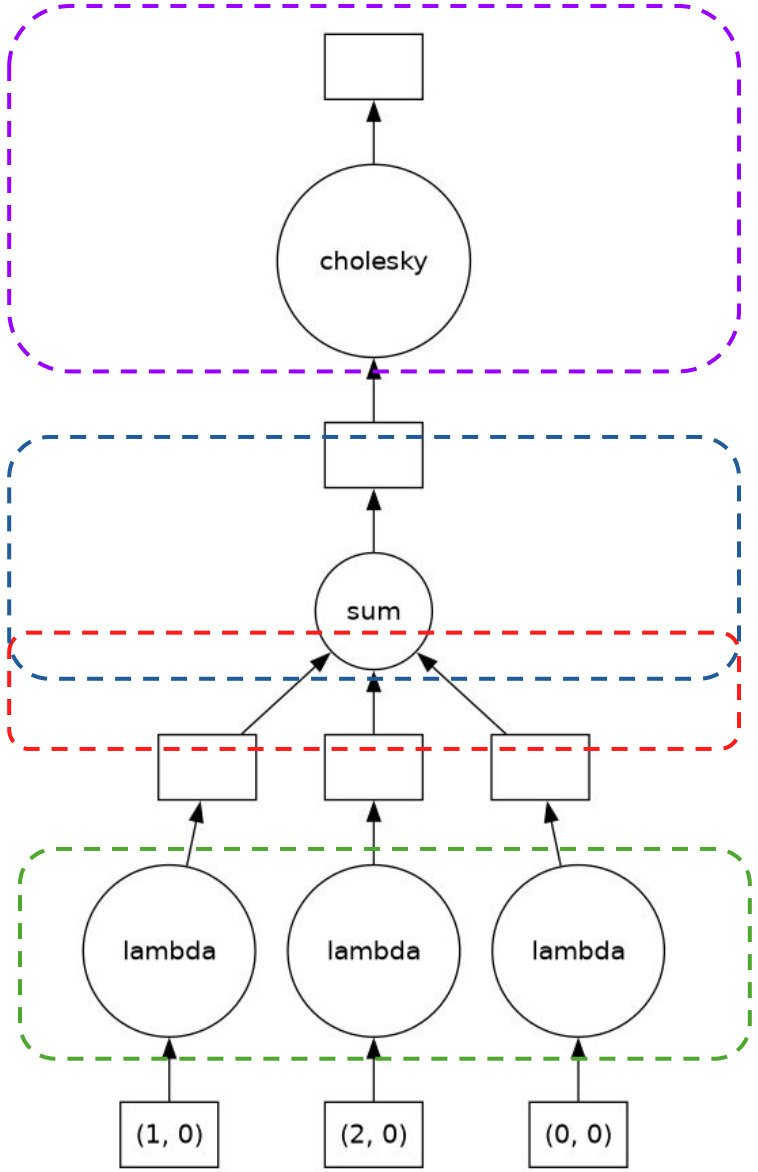
Cholesky TSQR

```
./functions.py

def cholesky_tsqr(X_da : dask.array.Array):

    chunks_delayed = [dask.delayed(gramMatMul)(chunk)
                      for chunk in X_da.to_delayed().ravel()]

    Gram_global_delayed = dask.delayed(sum)(chunks_delayed)
    R = dask.delayed(np.linalg.cholesky)(Gram_global_delayed)

    R_inv = dask.delayed(Inverse)(R)

    Q = X_da.map_blocks(MatMul, R_inv, dtype=X_da.dtype)

    return Q, R
```

Step 2: Recovering Q



Cholesky TSQR

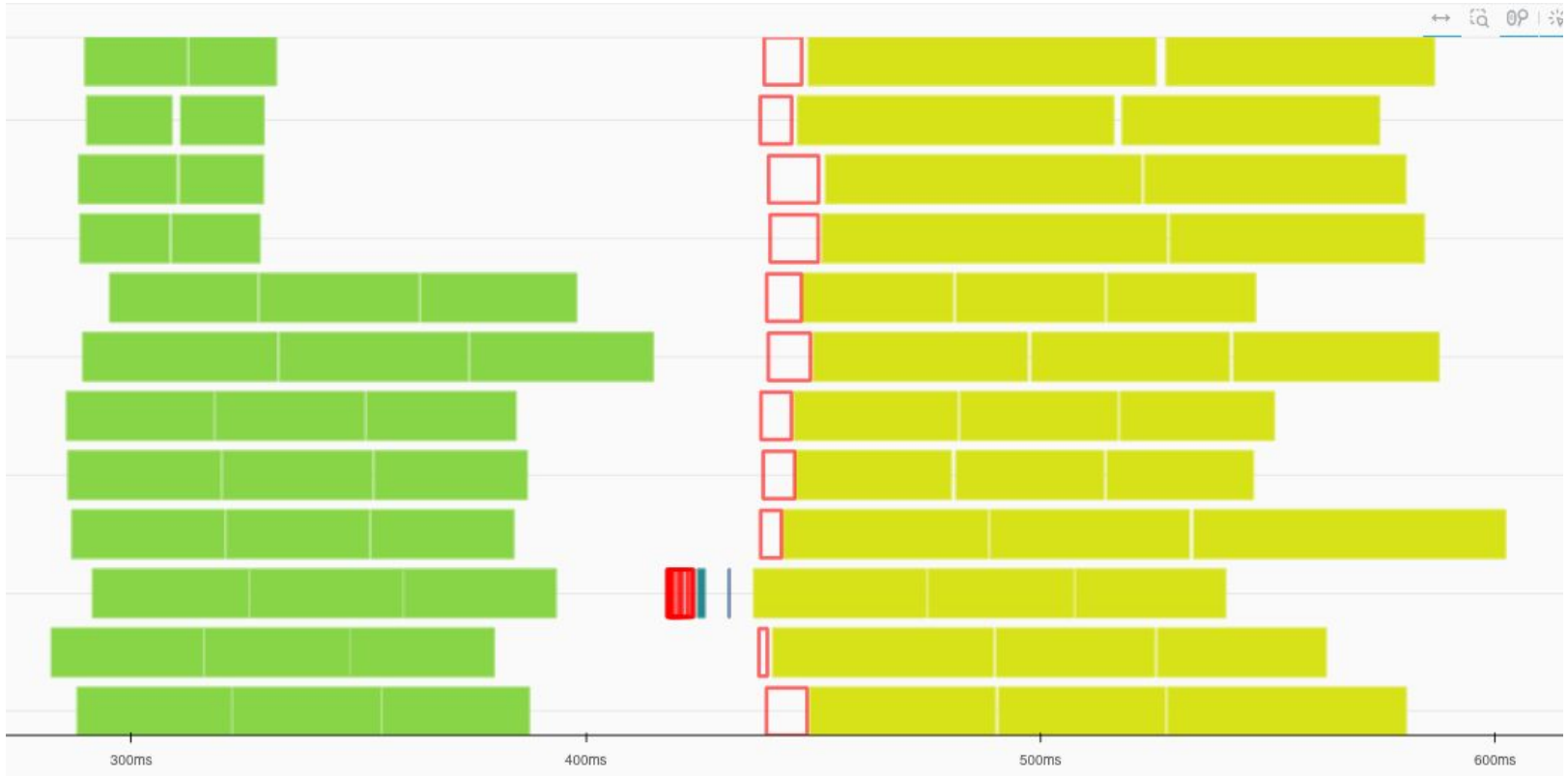


Computation of *R*

First and second step

Computation of *Q*

Third step



HIGGS dataset, 12 workers (only 3 displayed in the DAG out of simplicity)

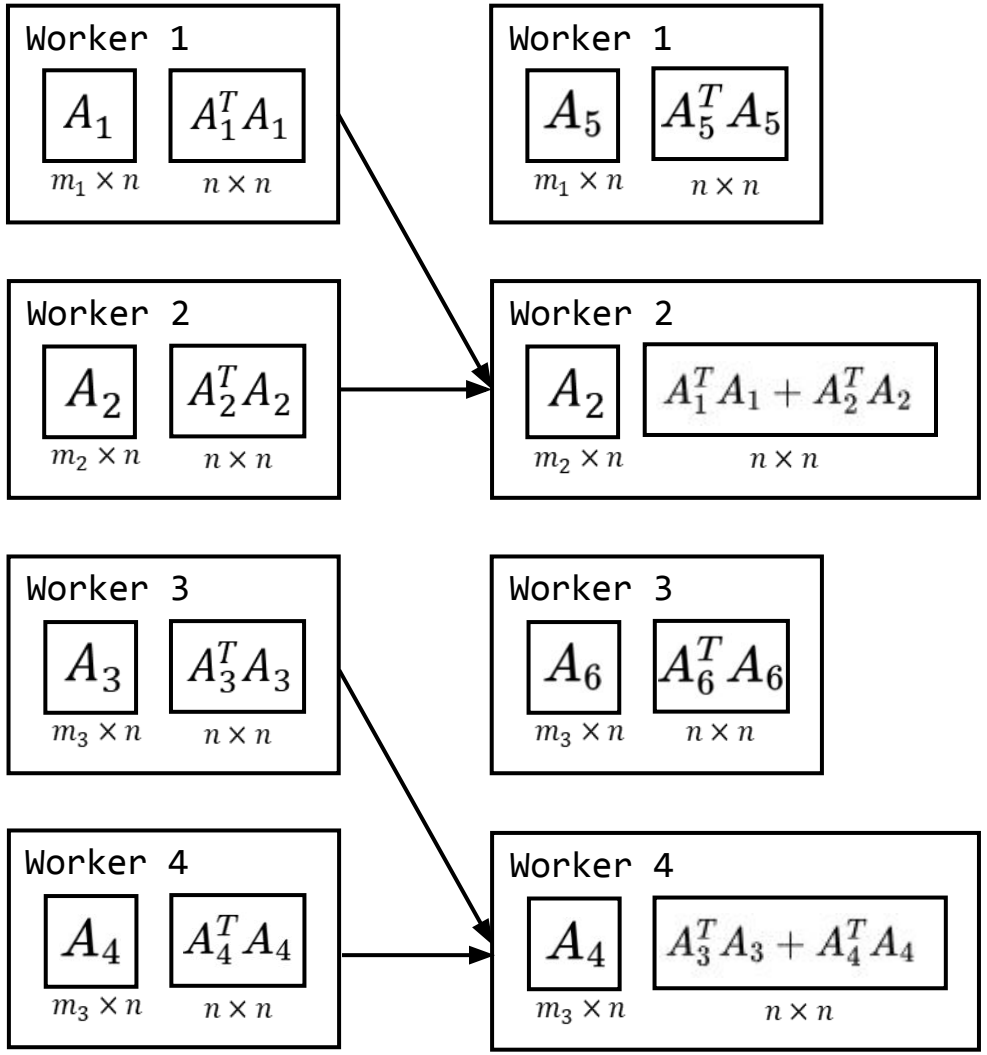
A different version: pair reduction

As of now, one worker only gather all the Gram matrices and perform the sum (serial)

Let's parallelize the sum too with a pairwise-reduction technique

./functions.py

```
def cholesky_tsqr(X_da : dask.array.Array):
    ...
    while len(chunks_delayed) > 1:
        new_level = []
        for i in range(0, len(chunks_delayed), 2):
            if i + 1 < len(chunks_delayed):
             new_level.append(dask.delayed(sum)(chunks_delayed[i],
chunks_delayed[i+1]))
            else:
                new_level.append(chunks_delayed[i])
        chunks_delayed = new_level
    ...
    return Q, R
```



Cholesky TSQR

Computation of *R*

Computation of *Q*

- Local Gram matrix $A_p^T A_p$
- Block-wise MatMul $Q = AR^{-1}$
- Transfer (of Gram, R_inv)
- Pairwise sum

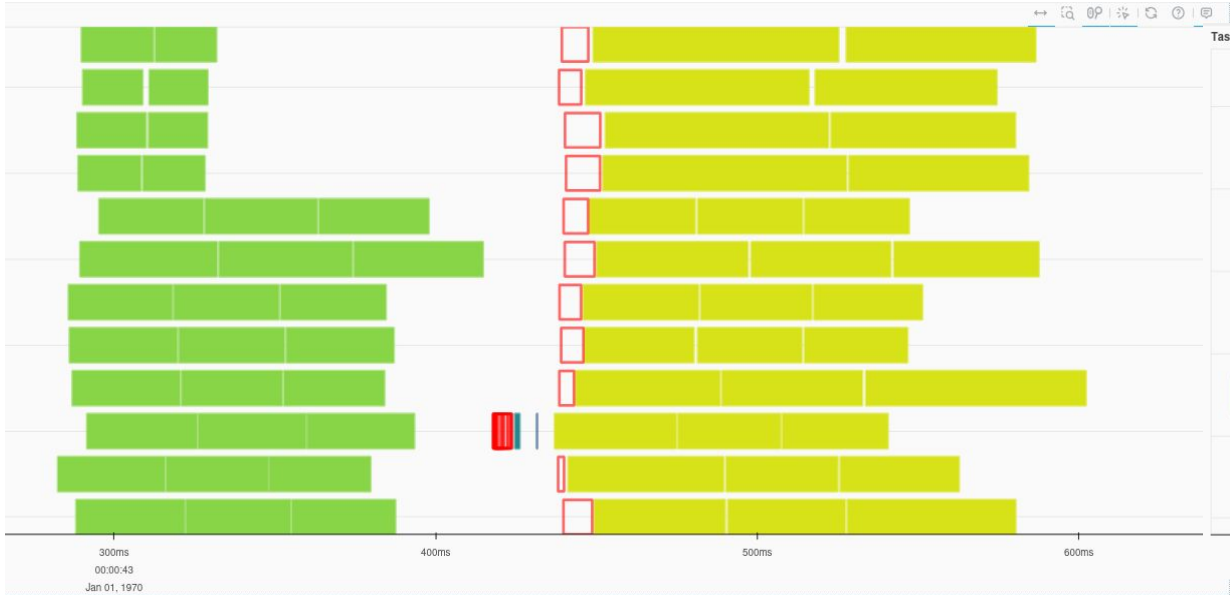
HIGGS dataset,
12 workers

Cholesky TSQR

Comparing the two methods, the version **without pairwise reduction** proves slightly faster than the alternative.

Why? Adding multiple pairwise summations **increases orchestration overhead.**

It's more efficient to transfer all the Gram matrices and aggregate them on a single worker, **since these matrices are very small** (typical in tall and skinny matrices).
Had the Gram matrices been larger (n larger) data transfers would have become a significant bottleneck, and the pairwise reduction approach might have been preferable.



HIGGS dataset,
12 workers

METHOD 2:

Indirect TSQR

./Indirect.ipynb

Indirect TSQR method

For an input matrix $A \in \mathbb{R}^{m \times n}$ **we perform partitioned QR factorizations:**

$$A = \begin{bmatrix} A_1^T & A_2^T & \cdots & A_p^T \end{bmatrix}^T, \quad A_j \in \mathbb{R}^{m_j \times n}$$

$$A_j = Q_j^{(1)} R_j, \quad Q_j^{(1)} \in \mathbb{R}^{m_j \times n}, \; R_j \in \mathbb{R}^{n \times n}.$$

Secondly, the global R matrix is obtained by stacking the shattered partitions:

$$R_{\text{stack}} = \begin{bmatrix} R_1 \\ R_2 \\ \vdots \\ R_p \end{bmatrix} \in \mathbb{R}^{pn \times n}$$

$$R_{\text{stack}} = \tilde{Q}\, R, \quad \tilde{Q} \in \mathbb{R}^{pn \times n}, \; R \in \mathbb{R}^{n \times n}$$

Ultimately the global Q is recovered indirectly by means of the inverse of R:

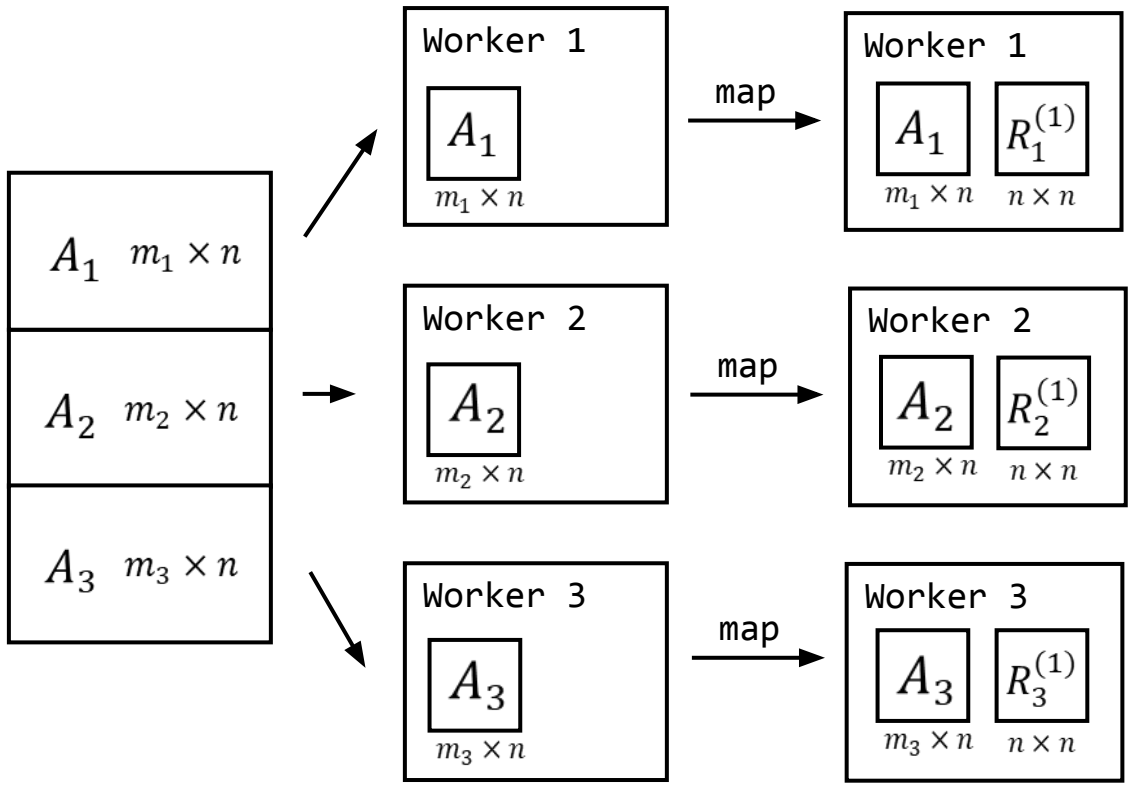
$$Q = AR^{-1}$$

Indirect TSQR

./functions.py

```python
def indirect_tsqr(X_da):
    n_cols = X_da.shape[1]
    R_blocks = X_da.map_blocks(compute_R,
                               dtype=X_da.dtype,
                               chunks=(n_cols, n_cols))

    R_stack = R_blocks.persist()

    _, R = np.linalg.qr(R_stack)

    I = np.eye(n_cols, dtype=X_da.dtype)
    R_inv = solve_triangular(R, I, lower=False)

    Q_da = X_da @ R_inv

    return Q_da, R
```

Step 1: Local QR

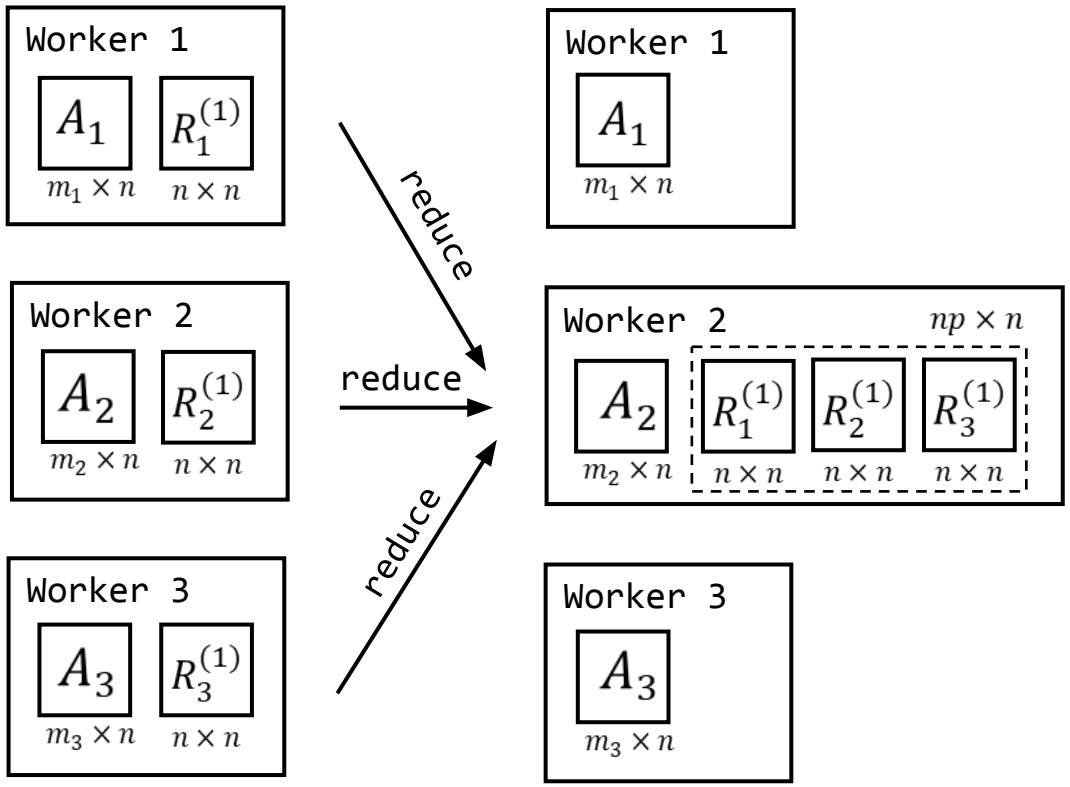


Indirect TSQR

./functions.py

```python
def indirect_tsqr(X_da):
    n_cols = X_da.shape[1]
    R_blocks = X_da.map_blocks(compute_R,
                               dtype=X_da.dtype,
                               chunks=(n_cols, n_cols))

    R_stack = R_blocks.persist()

    _, R = np.linalg.qr(R_stack)

    I = np.eye(n_cols, dtype=X_da.dtype)
    R_inv = solve_triangular(R, I, lower=False)

    Q_da = X_da @ R_inv

    return Q_da, R
```

Step 2: Global QR

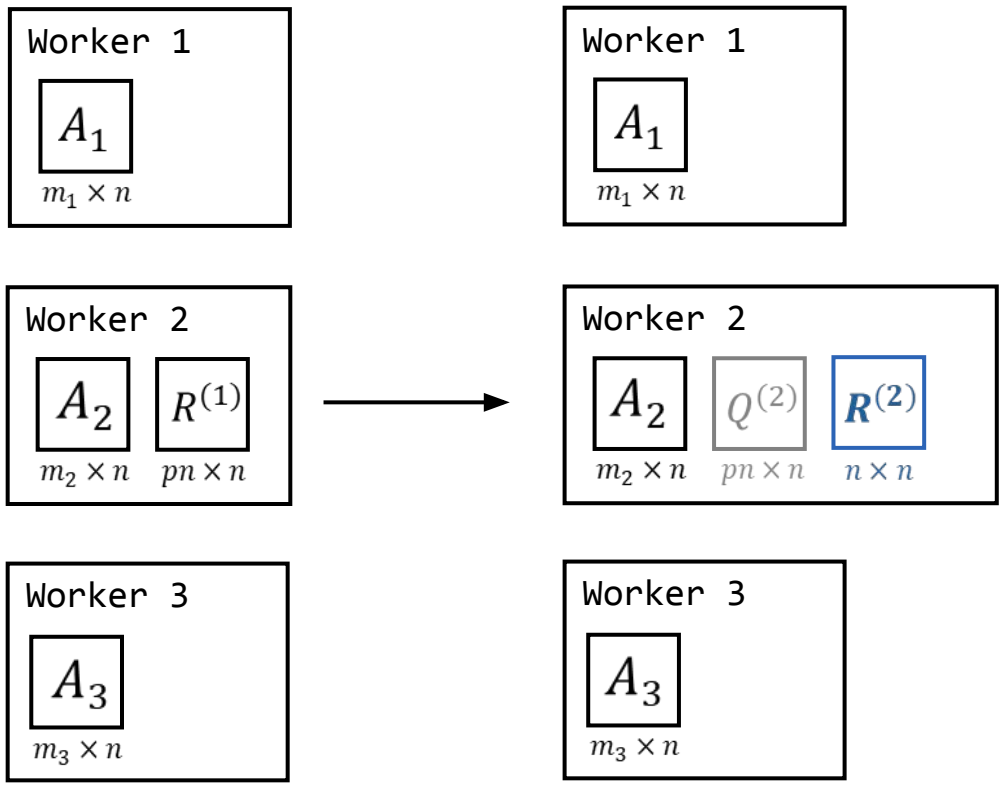


Indirect TSQR

./functions.py

```python
def indirect_tsqr(X_da):
    n_cols = X_da.shape[1]
    R_blocks = X_da.map_blocks(compute_R,
                               dtype=X_da.dtype,
                               chunks=(n_cols, n_cols))

    R_stack = R_blocks.persist()

    _, R = np.linalg.qr(R_stack)

    I = np.eye(n_cols, dtype=X_da.dtype)
    R_inv = solve_triangular(R, I, lower=False)

    Q_da = X_da @ R_inv

    return Q_da, R
```

Step 2: Global QR

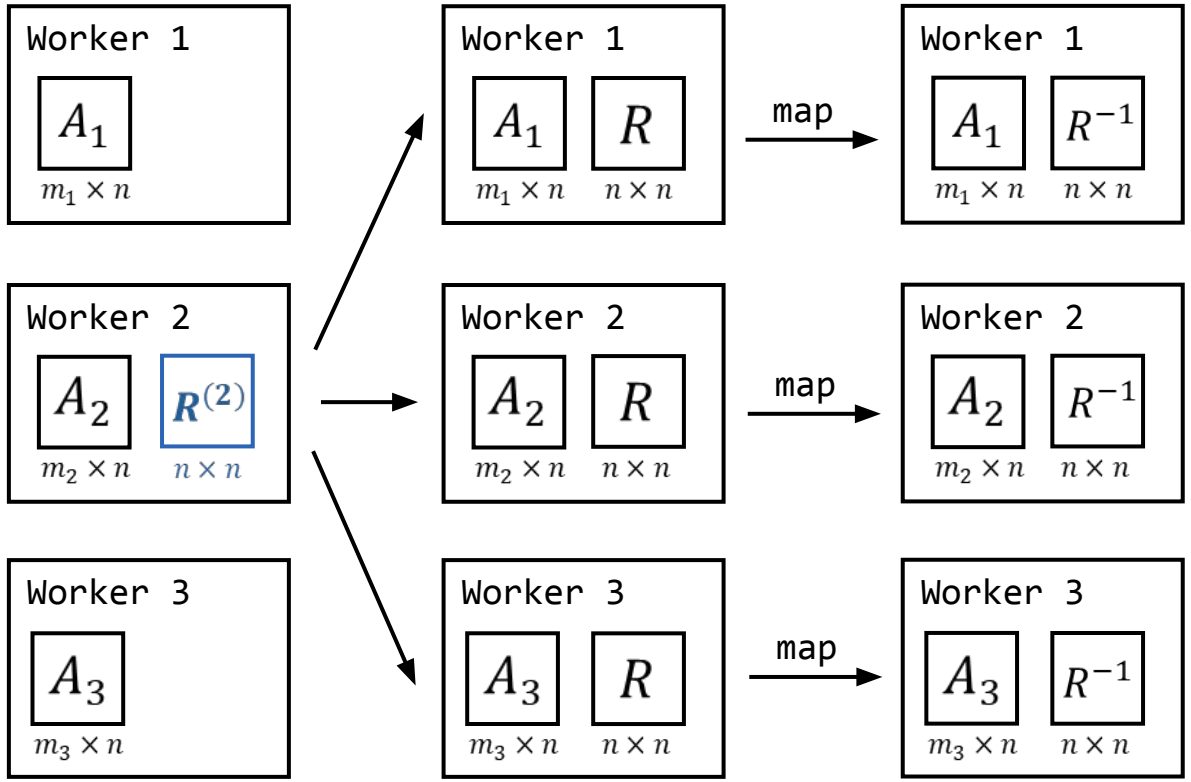


Indirect TSQR

./functions.py

```python
def indirect_tsqr(X_da):
    n_cols = X_da.shape[1]
    R_blocks = X_da.map_blocks(compute_R,
                               dtype=X_da.dtype,
                               chunks=(n_cols, n_cols))

    R_stack = R_blocks.persist()

    _, R = np.linalg.qr(R_stack)

    I = np.eye(n_cols, dtype=X_da.dtype)
    R_inv = solve_triangular(R, I, lower=False)

    Q_da = X_da @ R_inv

    return Q_da, R
```

Step 3: Recovering Q

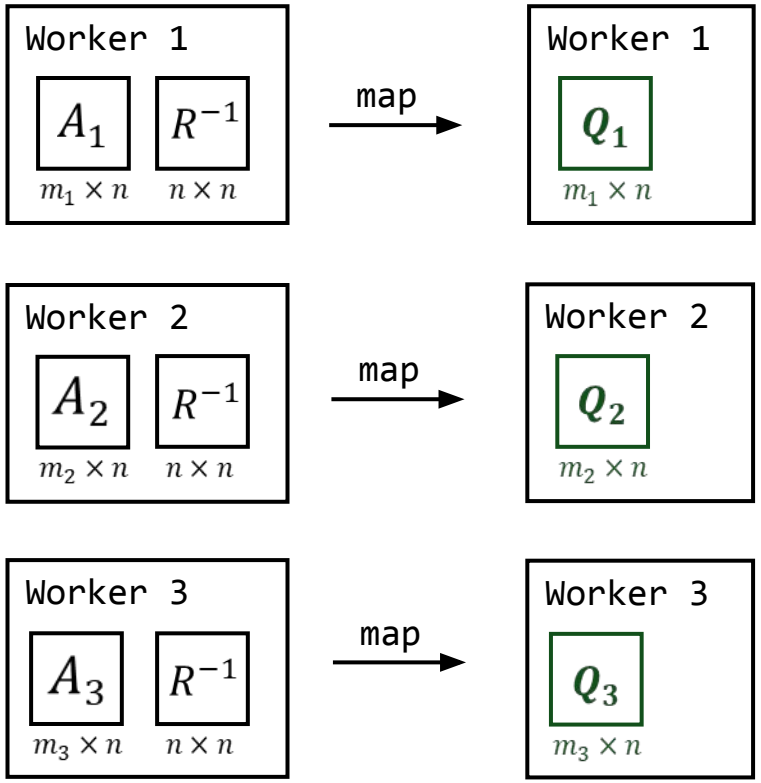


Indirect TSQR

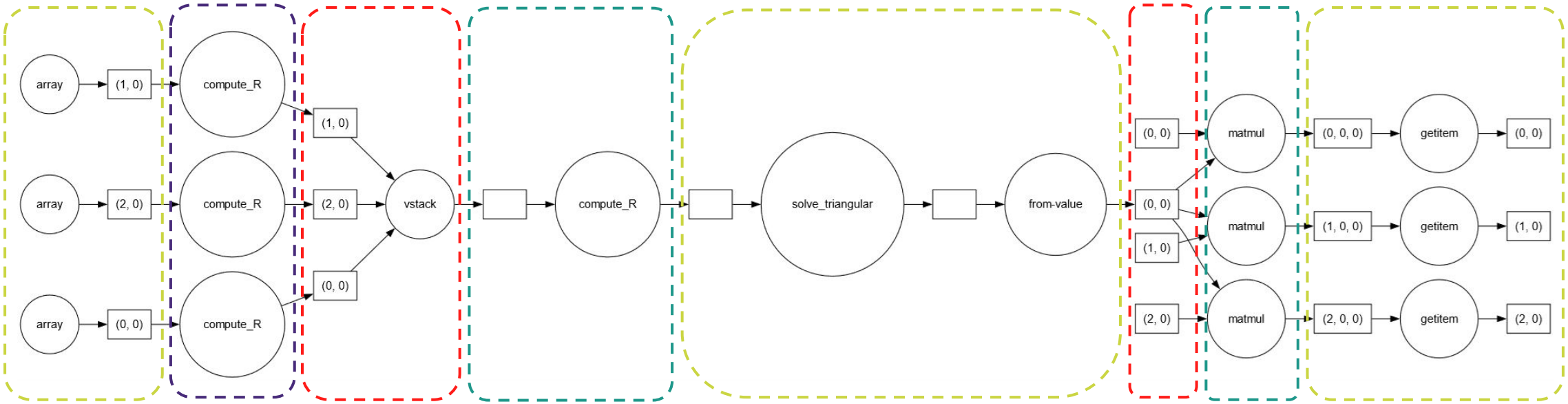
./functions.py

```python
def indirect_tsqr(X_da):
    n_cols = X_da.shape[1]
    R_blocks = X_da.map_blocks(compute_R,
                               dtype=X_da.dtype,
                               chunks=(n_cols, n_cols))

    R_stack = R_blocks.persist()

    _, R = np.linalg.qr(R_stack)

    I = np.eye(n_cols, dtype=X_da.dtype)
    R_inv = solve_triangular(R, I, lower=False)

    Q_da = X_da @ R_inv

    return Q_da, R
```

Step 3: Recovering Q



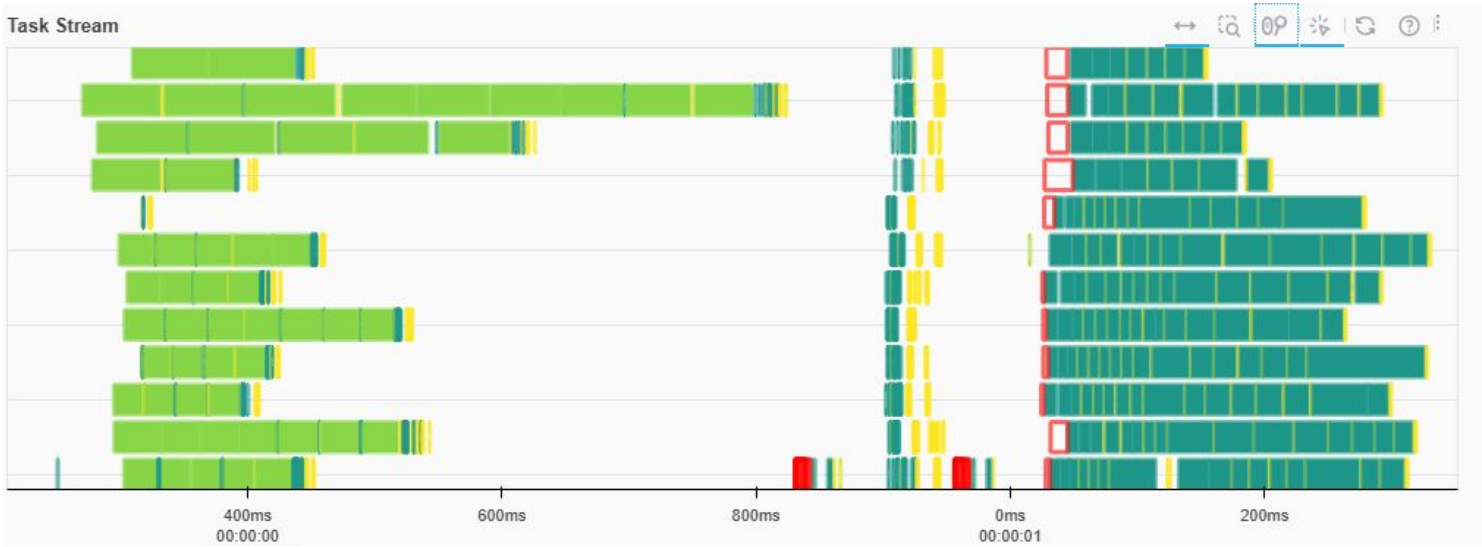
Indirect TSQR



Step 1

Step 2

Step 3



Higgs dataset

Variations: Global QR

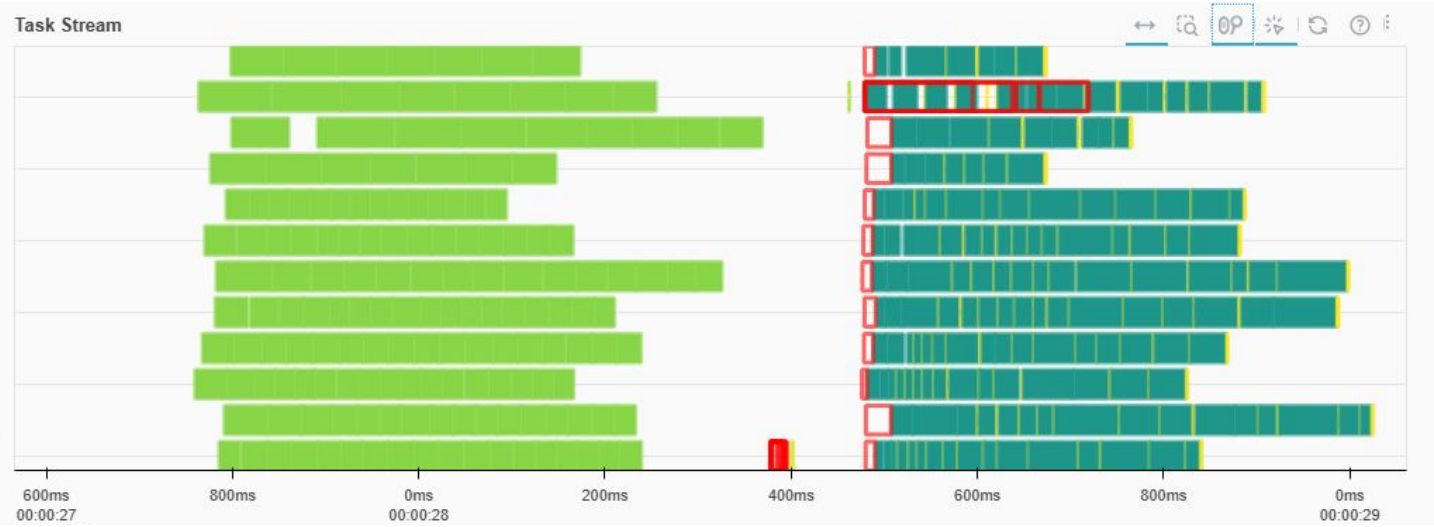
./functions.py

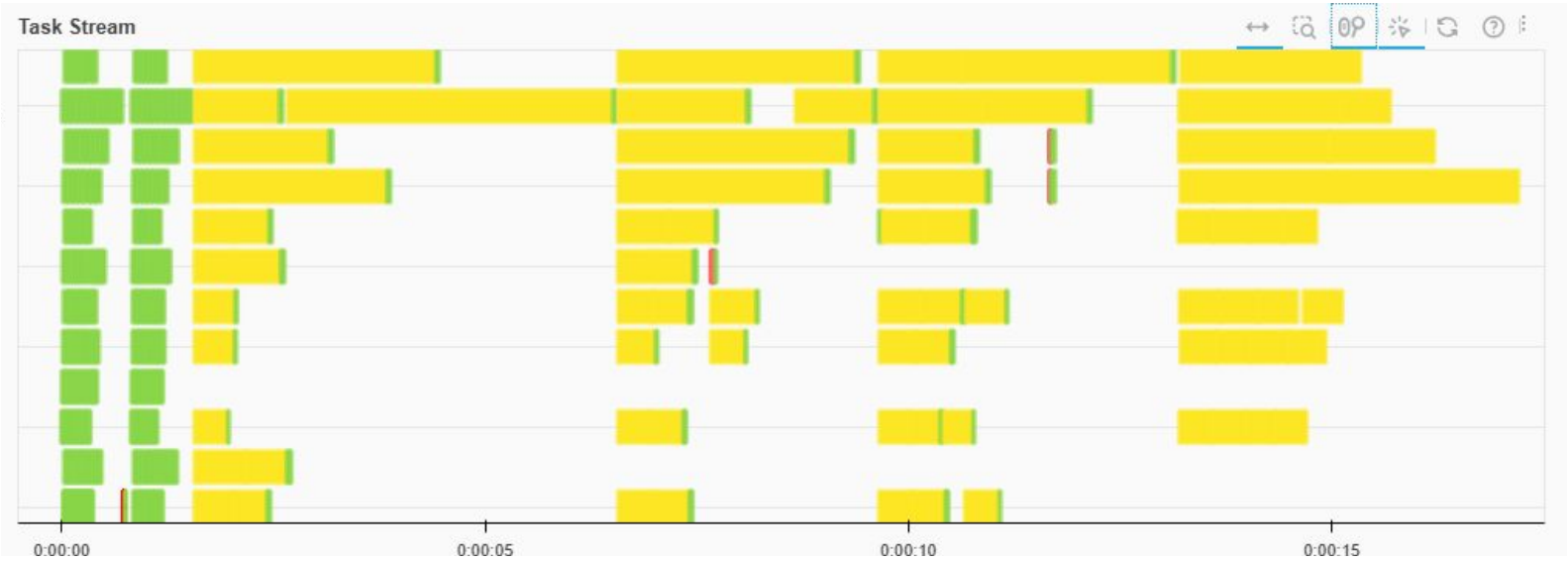
```python
def indirect_tsqr(X_da):
    n_cols = X_da.shape[1]
    R_blocks = X_da.map_blocks(compute_R,
                               dtype=X_da.dtype,
                               chunks=(n_cols, n_cols))

    R_stack = R_blocks.compute()

    _, R = np.linalg.qr(R_stack)

    I = np.eye(n_cols, dtype=X_da.dtype)
    R_inv = solve_triangular(R, I, lower=False)

    Q_da = X_da @ R_inv

    return Q_da, R
```

./functions.py

```python
def indirect_parallel_delayed(X_da):

    n_cols = X_da.shape[1]
    R_blocks = X_da.map_blocks(compute_R,
                               dtype=X_da.dtype,
                               chunks=(n_cols, n_cols))

    R_list = list(R_blocks.to_delayed().ravel())
    R_stack = dask.delayed(np.vstack)(R_list)

    R_delayed = dask.delayed(compute_R)(R_stack)
    I = np.eye(n_cols, dtype=X_da.dtype)

    R_inv_delayed = dask.delayed(solve_triangular)
                                 (R_delayed, I,
                                  lower=False)
    R_inv_da = da.from_delayed(R_inv_delayed,
                               shape=(n_cols, n_cols),
                               dtype=X_da.dtype)
    Q_da = X_da @ R_inv_da

    return Q_da, R_delayed
```

Variations: Global QR





METHOD 3:

Direct TSQR

./Direct.ipynb

Direct TSQR

Input: $A = (A_1 \quad A_2 \quad \dots \quad A_p)^T \qquad A_j \in \mathbb{R}^{m_j \times n}$

Step 1: Local QR	$A_j \mapsto \left(Q_j^{(1)}, R_j^{(1)}\right)$	$R_j^{(1)} \in \mathbb{R}^{n \times n}$ $Q_j^{(1)} \in \mathbb{R}^{m_j \times n}$
Step 2: Global QR	$R^{(1)} = \left(R_1^{(1)} \quad R_2^{(1)} \quad \dots \quad R_p^{(1)}\right)^T$ $R^{(1)} \mapsto Q^{(2)} R^{(2)}$	$R^{(1)} \in \mathbb{R}^{pn \times n}$ $R^{(2)} \in \mathbb{R}^{n \times n}$ $Q^{(2)} \in \mathbb{R}^{pn \times n}$
Step 3: Assembling Q	$Q^{(2)} = (Q_1 \quad Q_2 \quad \dots \quad Q_p)^T$ $Q_j = Q_j^{(1)} Q_j^{(2)}$	$Q_j^{(2)} \in \mathbb{R}^{n \times n}$ $Q_j \in \mathbb{R}^{m_j \times n}$

Output: $Q = (Q_1 \quad Q_2 \quad \dots \quad Q_p)^T \qquad Q \in \mathbb{R}^{m \times n}$

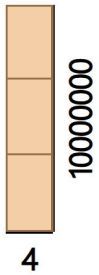
$R = R^{(2)} \qquad R \in \mathbb{R}^{n \times n}$

Direct TSQR

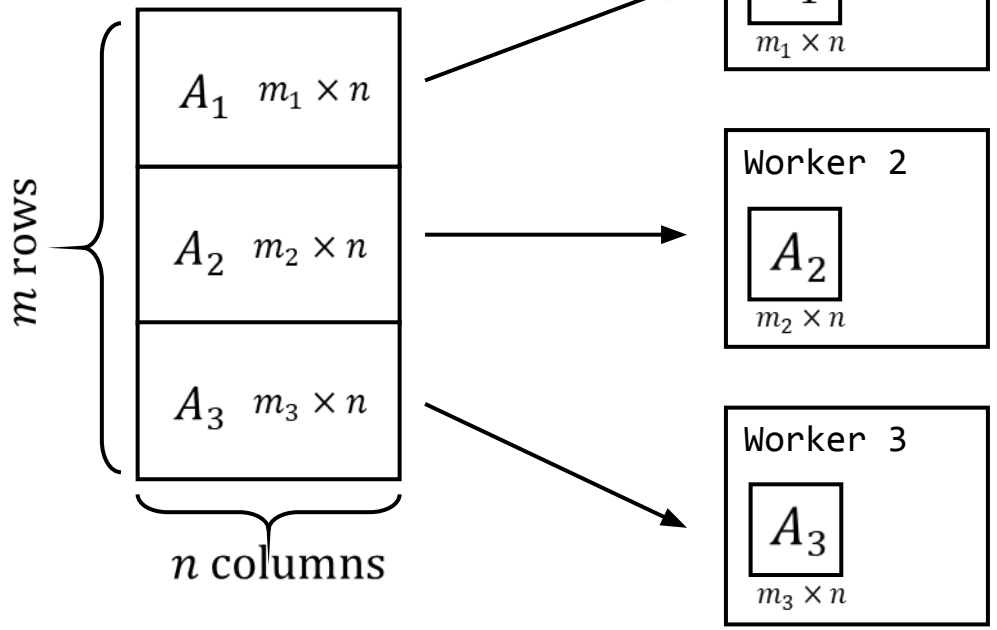
./functions.py

```python
def direct_tsqr(A : da.Array):

    n, row_chunks = A.shape[1], A.chunks[0]
    p = len(row_chunks)
    A_blocks = A.to_delayed().ravel().tolist()

    QR1 = [delayed(qr)(block) for block in A_blocks]
    Q1s = [qr[0] for qr in QR1]
    R1s = [qr[1] for qr in QR1]

    R1 = delayed(np.vstack)(R1s)
    QR2 = delayed(qr)(R1)
    Q2, R2 = QR2[0], QR2[1]
    Q2s = [Q2[i*n:(i+1)*n, :] for i in range(p)]

    Q_blocks = [da.from_delayed(
                delayed(lambda Q1, Q2: Q1 @ Q2)(Q1, Q2),
                shape=(row_chunks[i], n), dtype=A.dtype)
                for i, (Q1, Q2) in enumerate(zip(Q1s, Q2s))]
    Q = da.concatenate(Q_blocks)
    return Q, da.from_delayed(R2, (n, n), dtype=A.dtype)
```

	Array	Chunk
Bytes	305.18 MiB	101.73 MiB
Shape	(10000000, 4)	(3333334, 4)
Dask graph	3 chunks in 1 graph layer	
Data type	float64 numpy.ndarray	



row_chunks = $[m_1, m_2, m_3]$
p = 3

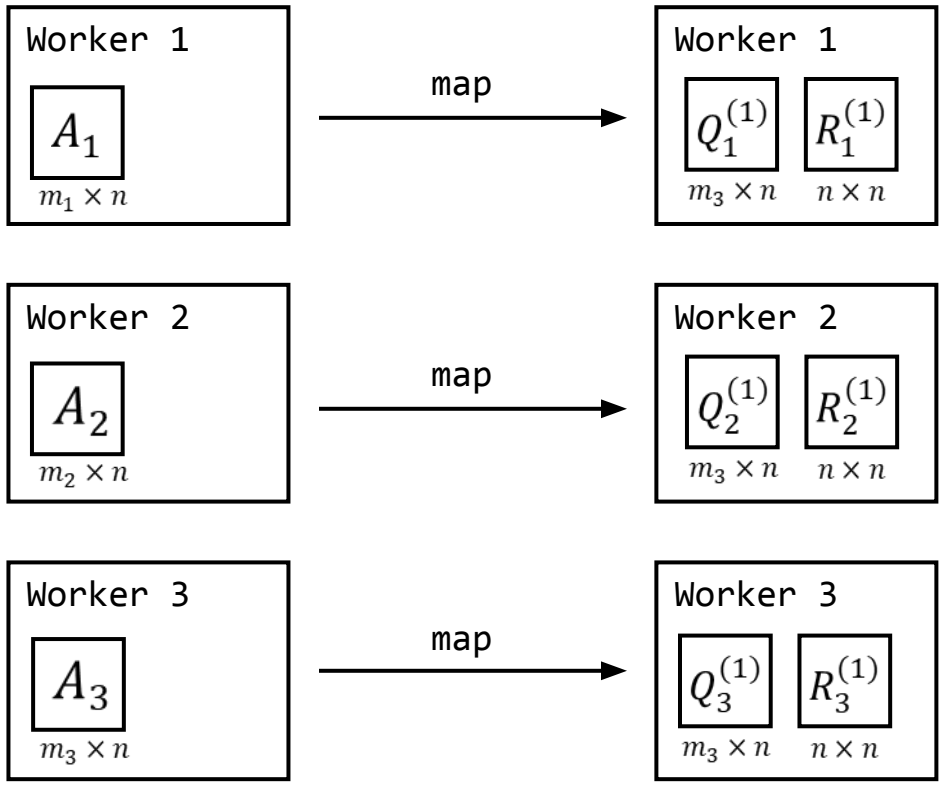


Direct TSQR

./functions.py

```python
def direct_tsqr(A : da.Array):

    n, row_chunks = A.shape[1], A.chunks[0]
    p = len(row_chunks)
    A_blocks = A.to_delayed().ravel().tolist()

    QR1 = [delayed(qr)(block) for block in A_blocks]
    Q1s = [qr[0] for qr in QR1]
    R1s = [qr[1] for qr in QR1]

    R1 = delayed(np.vstack)(R1s)
    QR2 = delayed(qr)(R1)
    Q2, R2 = QR2[0], QR2[1]
    Q2s = [Q2[i*n:(i+1)*n, :] for i in range(p)]

    Q_blocks = [da.from_delayed(
                delayed(lambda Q1, Q2: Q1 @ Q2)(Q1, Q2),
                shape=(row_chunks[i], n), dtype=A.dtype)
                for i, (Q1, Q2) in enumerate(zip(Q1s, Q2s))]
    Q = da.concatenate(Q_blocks)
    return Q, da.from_delayed(R2, (n, n), dtype=A.dtype)
```

Step 1: Local QR

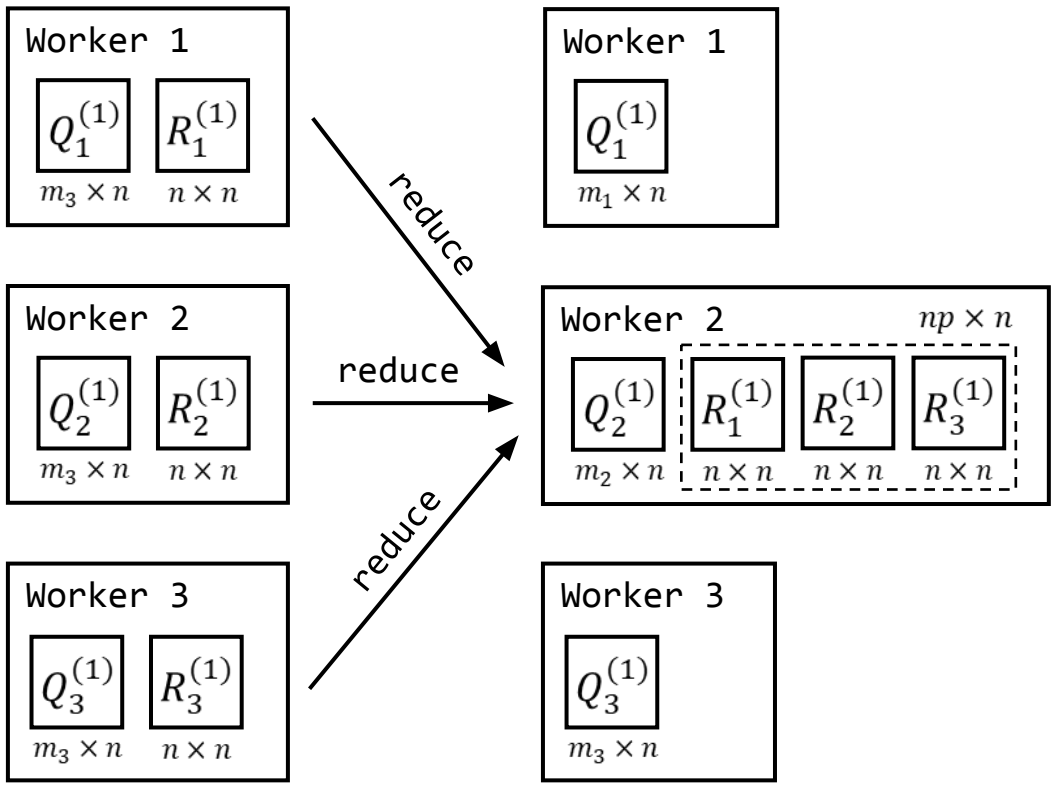


Direct TSQR

./functions.py

```python
def direct_tsqr(A : da.Array):

    n, row_chunks = A.shape[1], A.chunks[0]
    p = len(row_chunks)
    A_blocks = A.to_delayed().ravel().tolist()

    QR1 = [delayed(qr)(block) for block in A_blocks]
    Q1s = [qr[0] for qr in QR1]
    R1s = [qr[1] for qr in QR1]

    R1 = delayed(np.vstack)(R1s)
    QR2 = delayed(qr)(R1)
    Q2, R2 = QR2[0], QR2[1]
    Q2s = [Q2[i*n:(i+1)*n, :] for i in range(p)]

    Q_blocks = [da.from_delayed(
                delayed(lambda Q1, Q2: Q1 @ Q2)(Q1, Q2),
                shape=(row_chunks[i], n), dtype=A.dtype)
                for i, (Q1, Q2) in enumerate(zip(Q1s, Q2s))]
    Q = da.concatenate(Q_blocks)
    return Q, da.from_delayed(R2, (n, n), dtype=A.dtype)
```

Step 2: Global QR

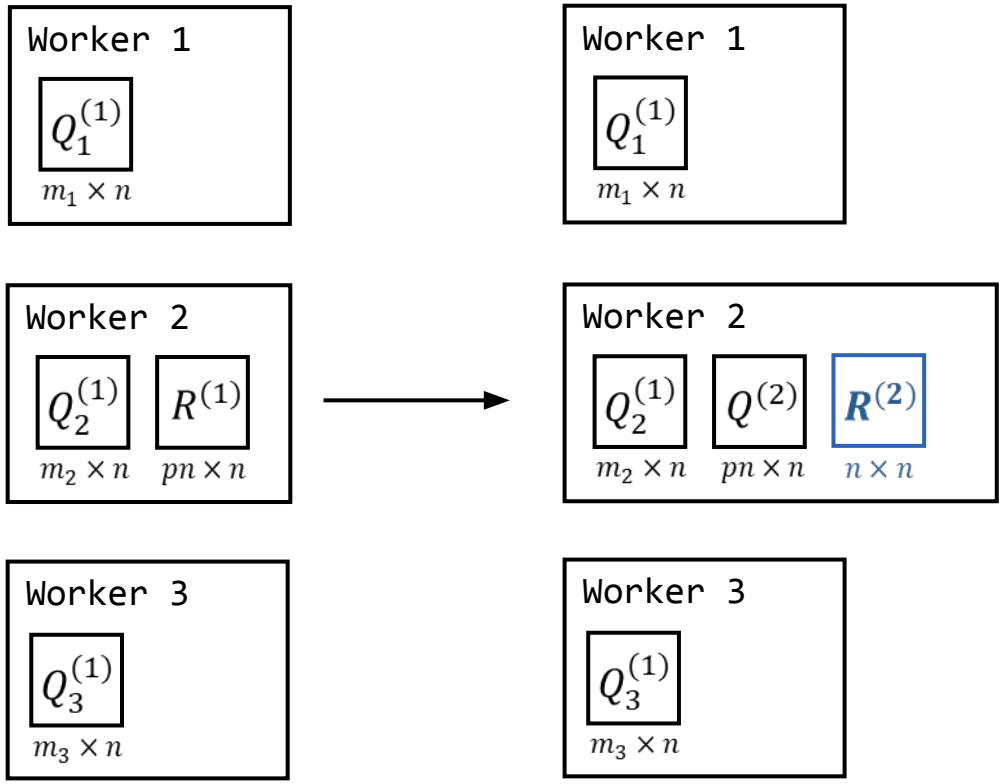


Direct TSQR

./functions.py

```python
def direct_tsqr(A : da.Array):

    n, row_chunks = A.shape[1], A.chunks[0]
    p = len(row_chunks)
    A_blocks = A.to_delayed().ravel().tolist()

    QR1 = [delayed(qr)(block) for block in A_blocks]
    Q1s = [qr[0] for qr in QR1]
    R1s = [qr[1] for qr in QR1]

    R1 = delayed(np.vstack)(R1s)
    QR2 = delayed(qr)(R1)
    Q2, R2 = QR2[0], QR2[1]
    Q2s = [Q2[i*n:(i+1)*n, :] for i in range(p)]

    Q_blocks = [da.from_delayed(
                delayed(lambda Q1, Q2: Q1 @ Q2)(Q1, Q2),
                shape=(row_chunks[i], n), dtype=A.dtype)
                for i, (Q1, Q2) in enumerate(zip(Q1s, Q2s))]
    Q = da.concatenate(Q_blocks)
    return Q, da.from_delayed(R2, (n, n), dtype=A.dtype)
```

Step 2: Global QR

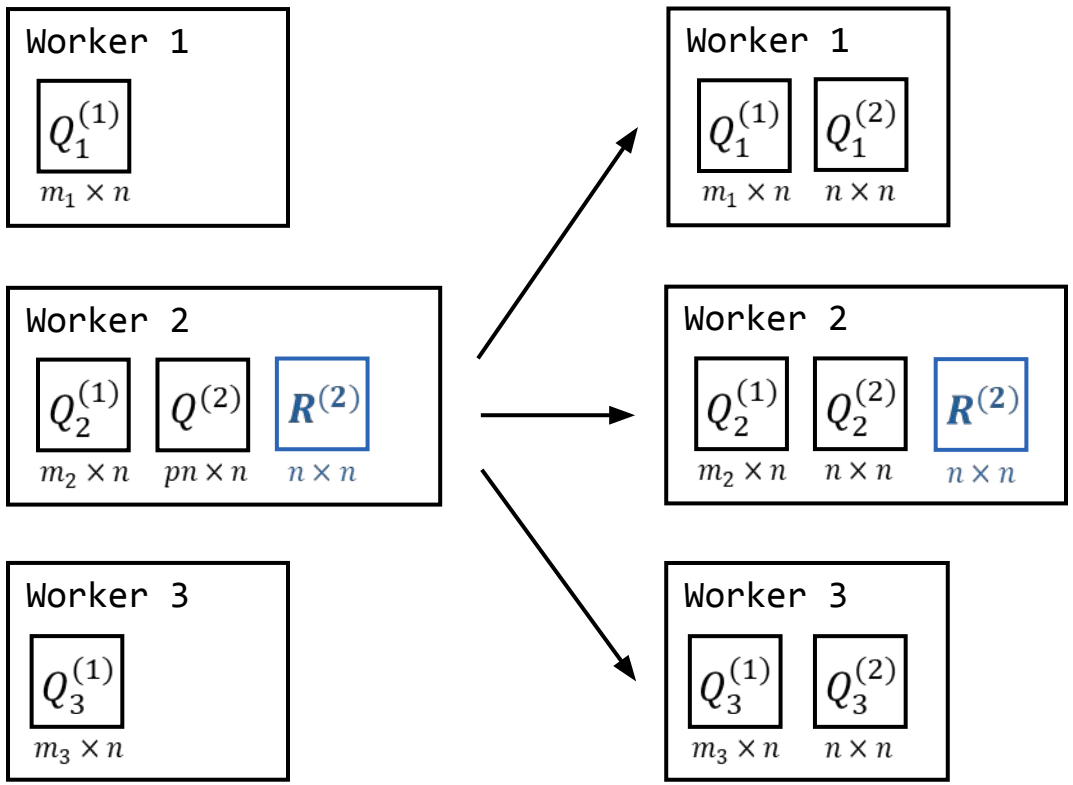


Direct TSQR

./functions.py

```python
def direct_tsqr(A : da.Array):

    n, row_chunks = A.shape[1], A.chunks[0]
    p = len(row_chunks)
    A_blocks = A.to_delayed().ravel().tolist()

    QR1 = [delayed(qr)(block) for block in A_blocks]
    Q1s = [qr[0] for qr in QR1]
    R1s = [qr[1] for qr in QR1]

    R1 = delayed(np.vstack)(R1s)
    QR2 = delayed(qr)(R1)
    Q2, R2 = QR2[0], QR2[1]
    Q2s = [Q2[i*n:(i+1)*n, :] for i in range(p)]

    Q_blocks = [da.from_delayed(
                delayed(lambda Q1, Q2: Q1 @ Q2)(Q1, Q2),
                shape=(row_chunks[i], n), dtype=A.dtype)
                for i, (Q1, Q2) in enumerate(zip(Q1s, Q2s))]
    Q = da.concatenate(Q_blocks)
    return Q, da.from_delayed(R2, (n, n), dtype=A.dtype)
```

Step 3: Assembling Q

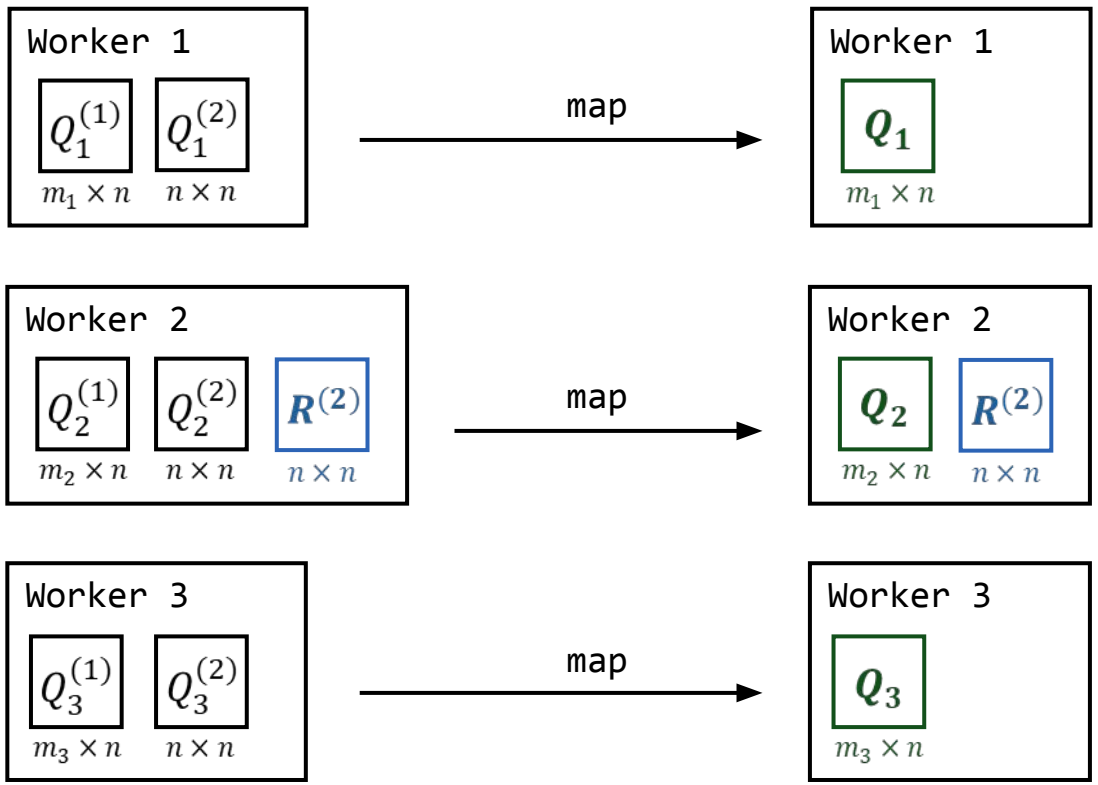


Direct TSQR

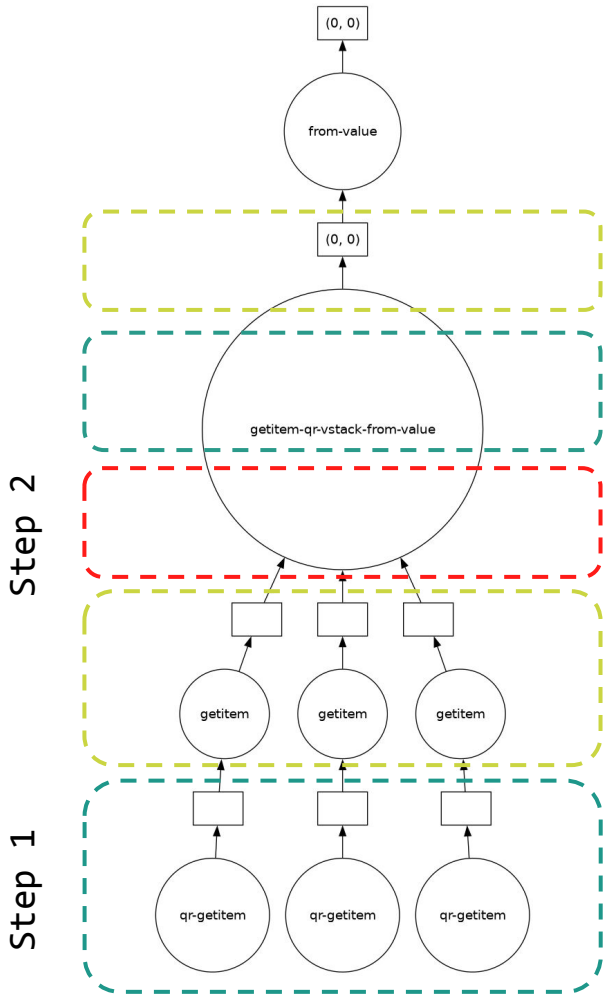
./functions.py

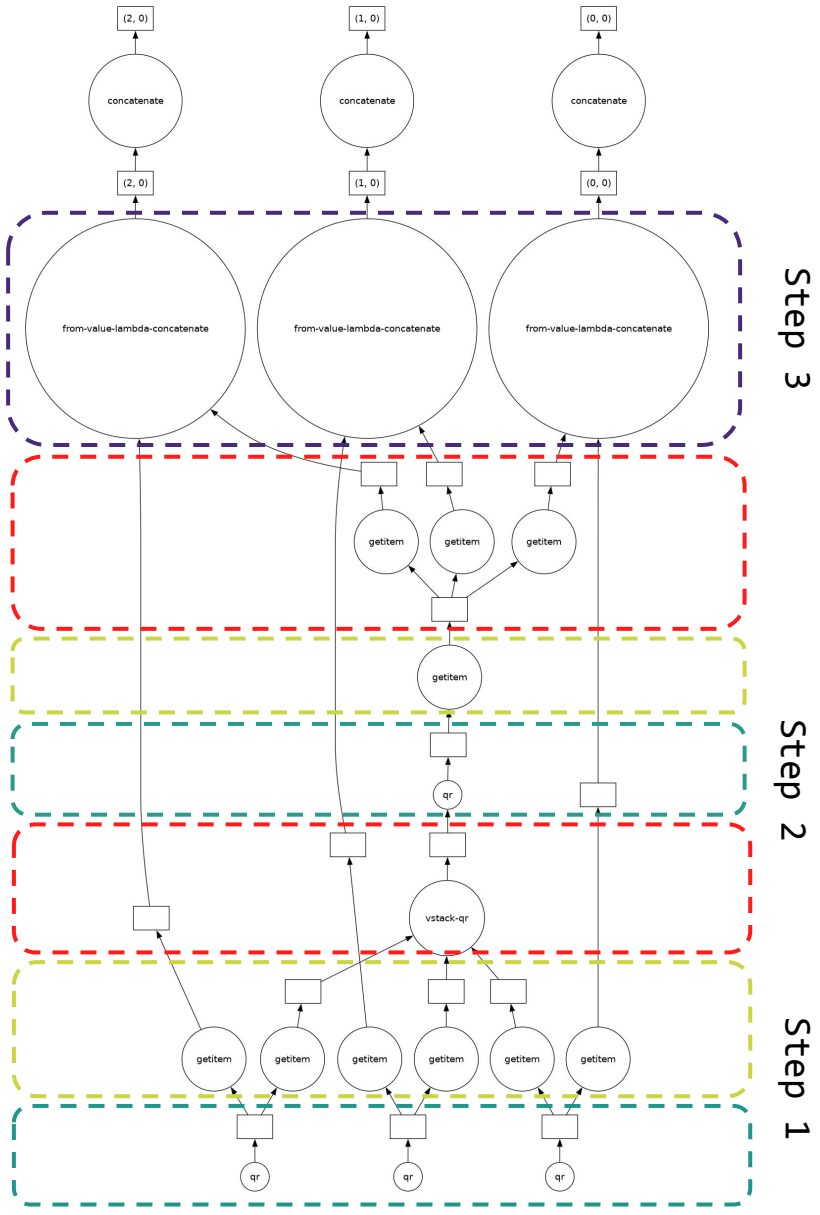
```python
def direct_tsqr(A : da.Array):

    n, row_chunks = A.shape[1], A.chunks[0]
    p = len(row_chunks)
    A_blocks = A.to_delayed().ravel().tolist()

    QR1 = [delayed(qr)(block) for block in A_blocks]
    Q1s = [qr[0] for qr in QR1]
    R1s = [qr[1] for qr in QR1]

    R1 = delayed(np.vstack)(R1s)
    QR2 = delayed(qr)(R1)
    Q2, R2 = QR2[0], QR2[1]
    Q2s = [Q2[i*n:(i+1)*n, :] for i in range(p)]

    Q_blocks = [da.from_delayed(
                delayed(lambda Q1, Q2: Q1 @ Q2)(Q1, Q2),
                shape=(row_chunks[i], n), dtype=A.dtype)
                for i, (Q1, Q2) in enumerate(zip(Q1s, Q2s))]
    Q = da.concatenate(Q_blocks)
    return Q, da.from_delayed(R2, (n, n), dtype=A.dtype)
```

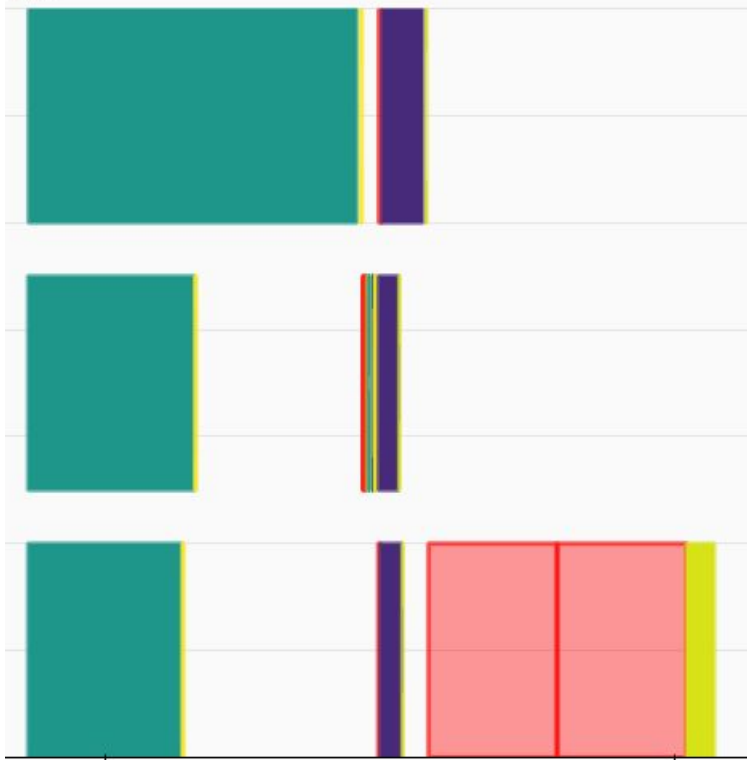
Step 3: Assembling Q



Direct TSQR







Synthetic dataset

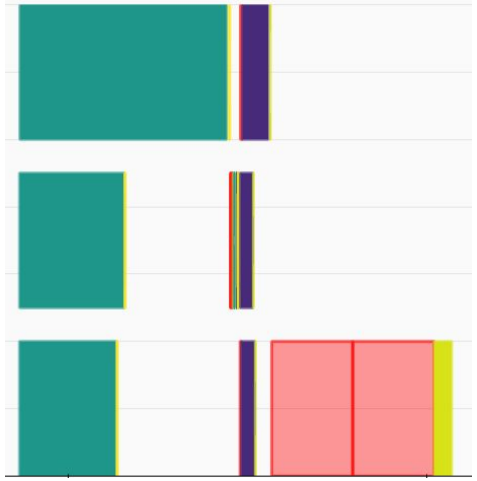
Direct TSQR

Our implementation

`direct_tsqr`

`(m, n) = (1e7, 4)`

`Time: 2.21 s`

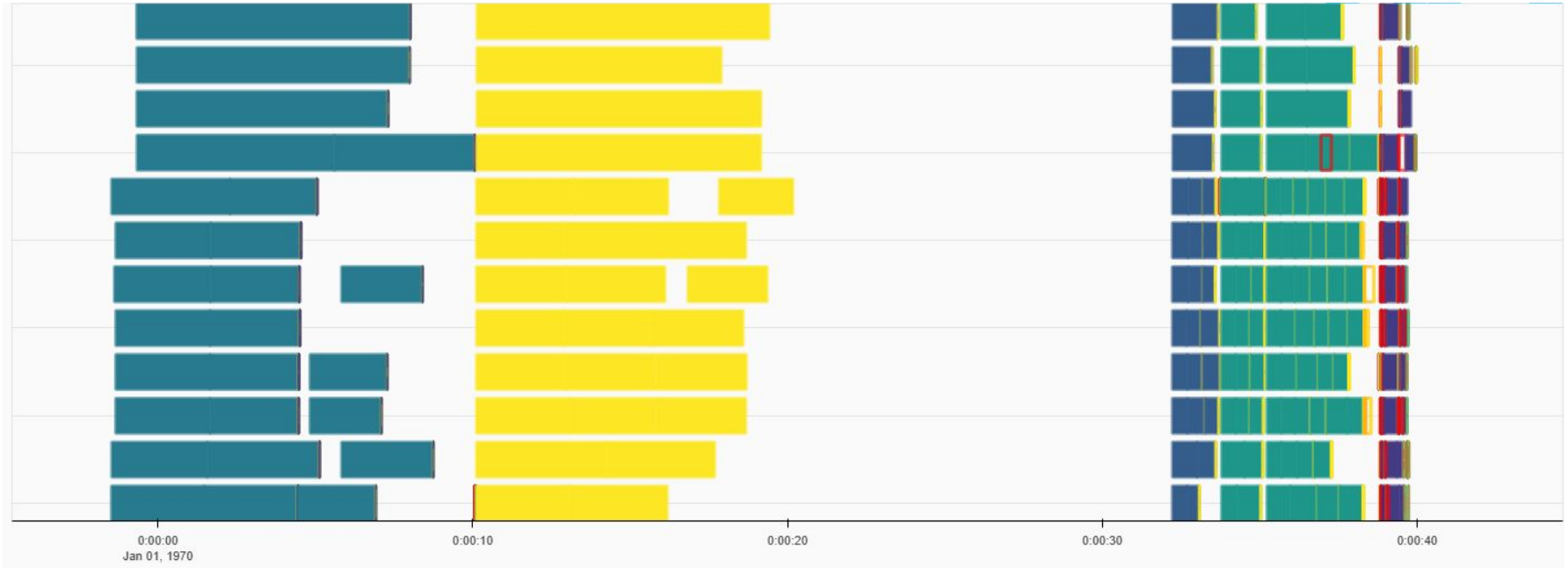


Dask's implementation

`dask.array.linalg.tsqr`

`(m, n) = (1e7, 4)`

`Time: 2.34 s`



Higgs dataset

Both Dask's and our methods are computed and then compared with a delayed `np.isclose()` to avoid transferring Q to a single worker

Direct TSQR - Singular Value Decomposition

```
./functions.py

SVD = delayed(svd)(R2)
U_R, S, Vt = SVD[0], SVD[1], SVD[2]

U_blocks = [da.from_delayed(delayed(lambda Q1, Q2, U_R:
            Q1 @ Q2 @ U_R)(Q1, Q2, U_R),
            shape=(row_chunks[i], n), dtype=A.dtype)
            for i, (Q1, Q2) in enumerate(zip(Q1s, Q2s))]
U = da.concatenate(U_blocks)
return U, da.from_delayed(S, (n,), dtype=A.dtype), \
       da.from_delayed(Vt, (n, n), dtype=A.dtype)
```

Extension to SVD

$$A = QR = Q(U_R \Sigma_R V_R^T) =$$
$$= (QU_R)\Sigma_R V_R^T$$

$$\boxed{\begin{aligned} U &= QU_R \\ V^T &= V_R^T \\ \Sigma &= \Sigma_R \end{aligned}}$$

Our implementation

`direct_tsqr`

`(m, n) = (1e7, 4)`

`Time: 2.14 s`

Dask's implementation

`dask.array.linalg.tsqr`

`(m, n) = (1e7, 4)`

`Time: 2.23 s`

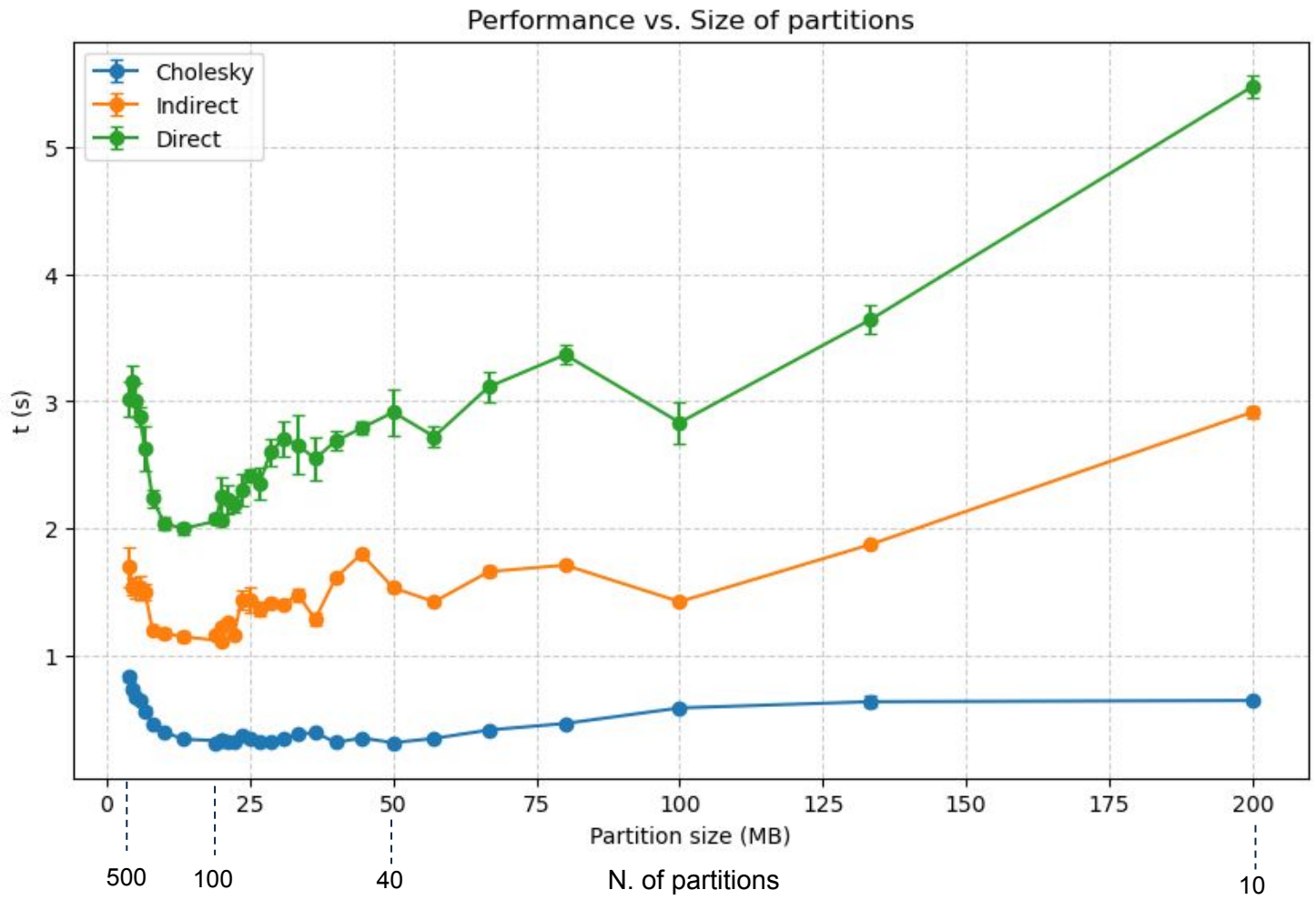
BENCHMARK 1:

Partition number

*This benchmark analyzes how the **number of partitions affects the performance** and accuracy of the TSQR algorithms, showing that optimal partitioning improves speed and excessive partitioning increases overhead, and highlighting the **trade-offs** between speed and numerical stability for each method.*

`./Benchmark_partitions.ipynb`

Benchmark 1: Partition number



First observation:

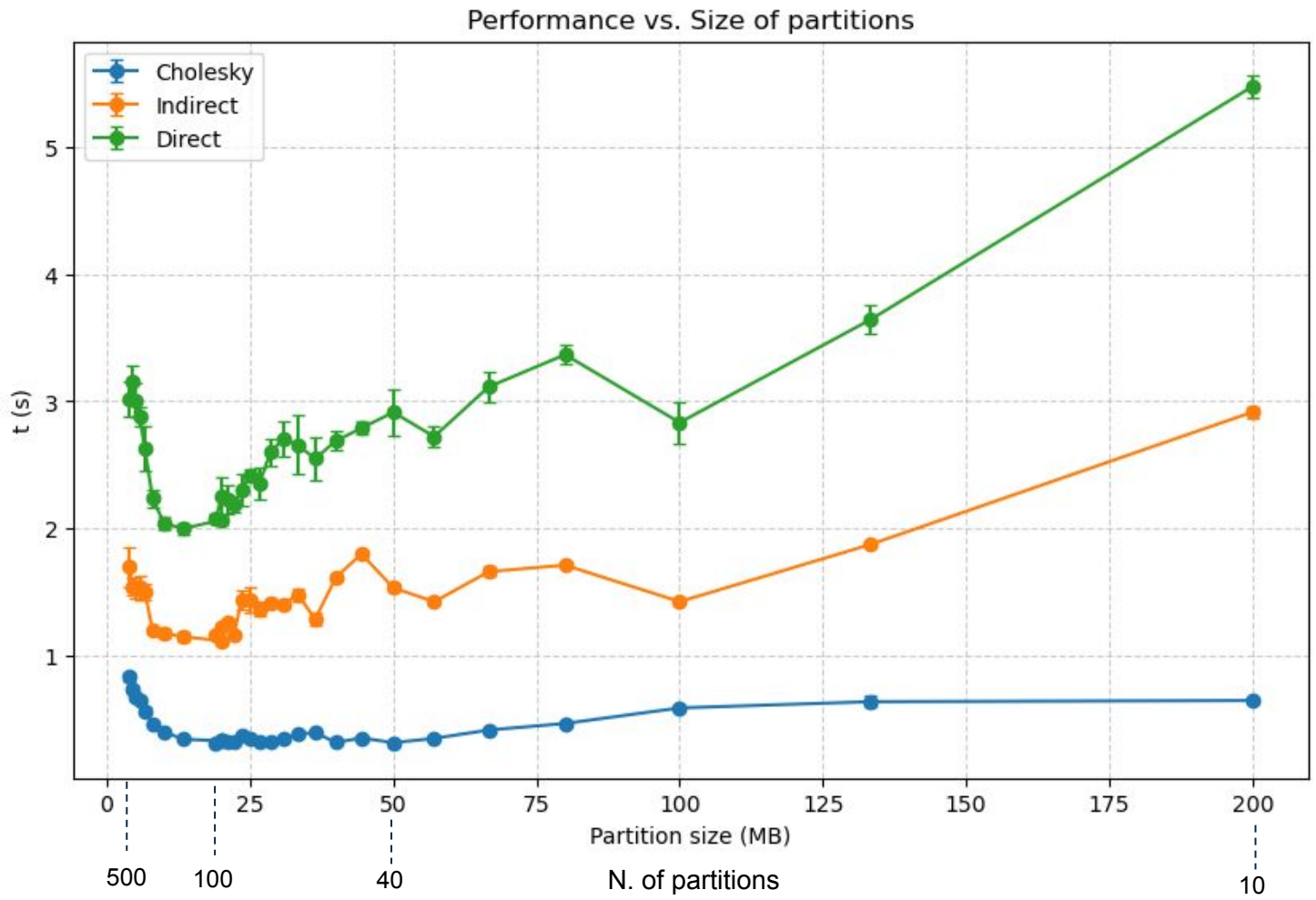
- Too **few partitions** (e.g., 10 partitions for 12 workers) make each chunk large (and can leave some workers idle.)
- Too **many partitions** create very small chunks, making computations easy, but the orchestration overhead increases, overall reducing performance.

↓

A sweet spot can be found

HIGGS dataset (2 GB), 12 workers. Repeat the algorithms 25 times each, collect the median and the MAD (median absolute deviation)

Benchmark 1: Partition number



Second observation:

It seems like **Cholesky decomposition** is faster than other state-of-the-art methods …

Although faster, **Cholesky** suffers from instability and inaccuracy. Specifically:

$$||Q^TQ - I||_2 \approx 136.80 \quad \text{[Cholesky]}$$

$$||Q^TQ - I||_2 \approx 3.7 \cdot 10^{-15} \quad \text{[Direct]}$$

$$||A - QR||_2 \approx 5 \quad \text{[Cholesky]}$$

$$||A - QR||_2 \approx 1 \cdot 10^{-15} \quad \text{[Direct]}$$

HIGGS dataset (2 GB), 12 workers

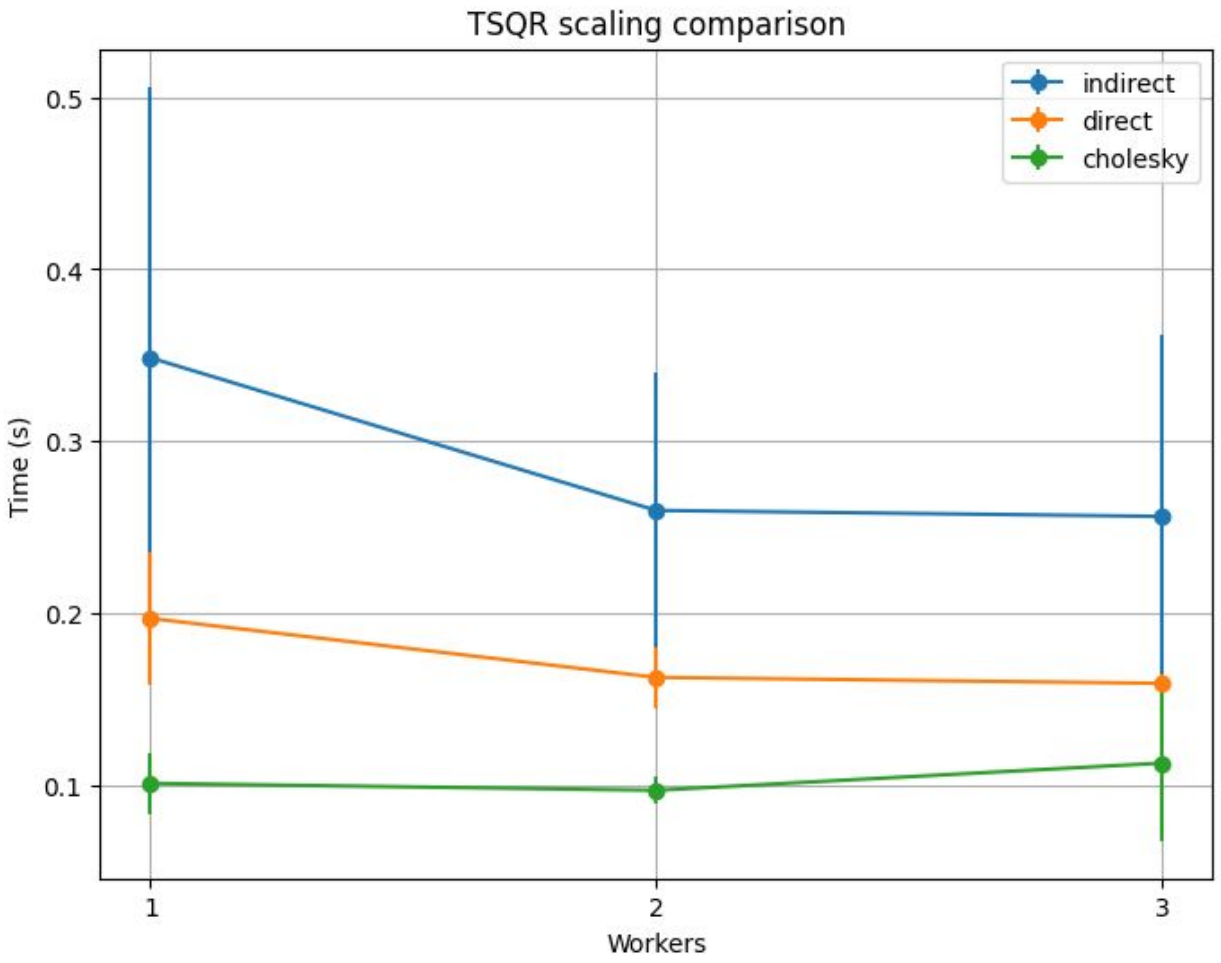
BENCHMARK 2:

Workers number

*This notebook benchmarks how varying the **number of Dask workers** affects the **performance** of the three TSQR algorithms on fixed-size datasets, showing that increasing workers improves speed for large datasets due to better parallelization*

`./Benchmark_workers.ipynb`

Benchmark 2: Workers number




```
cluster.scale(target)
```

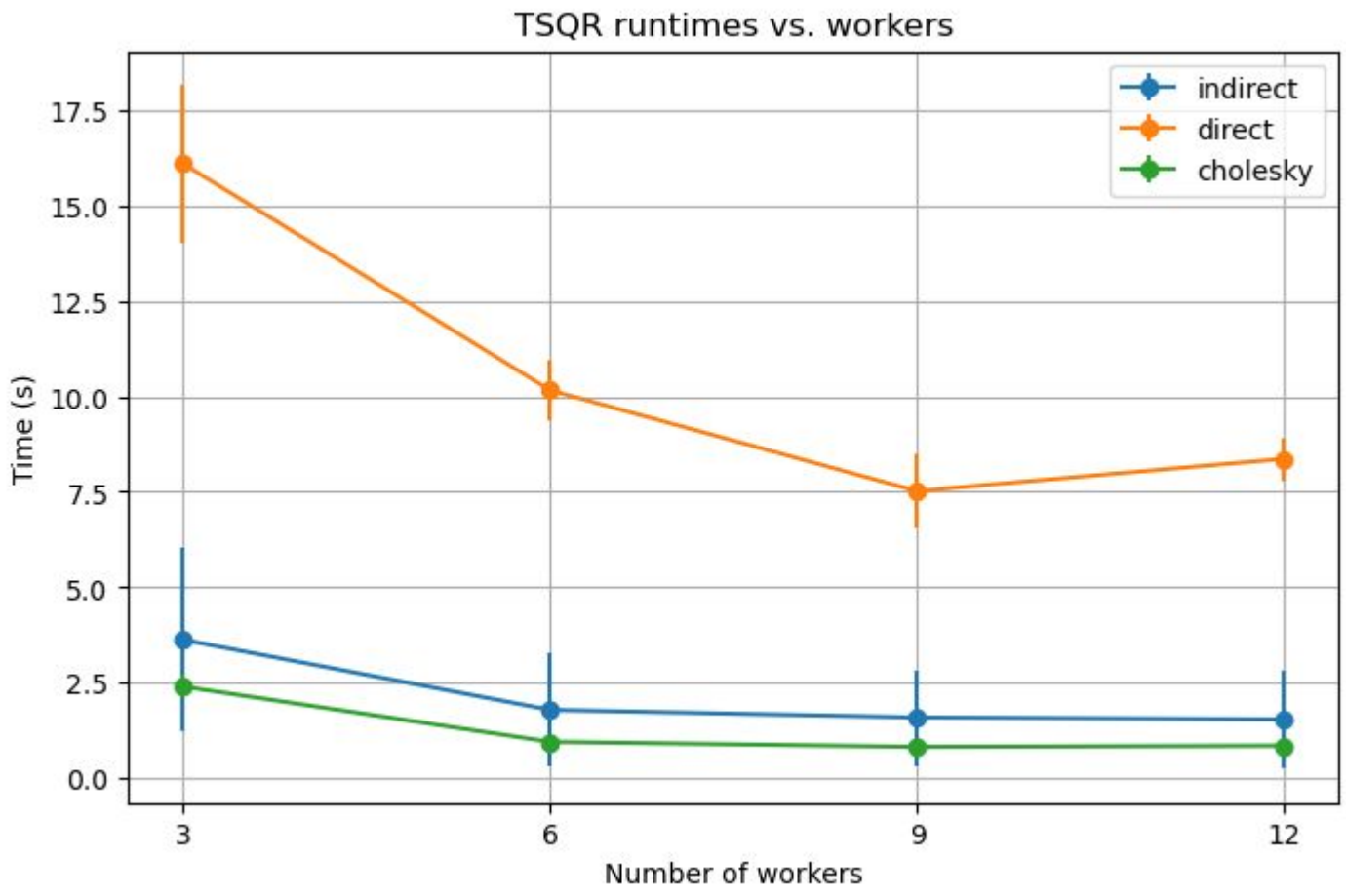
Observations:

- For a **small dataset** (e.g., < 500 Mb) The workload for a single worker isn't excessive, and the results are comparable between the cases.

- The **standard deviation** of the runtimes can be relatively high and oscillate significantly across runs. Due to light computational load and external factors.

California Housing dataset

Benchmark 2: Workers number




```
cluster = SSHCluster(...)
```

Observations:

- For a **bigger dataset** (e.g., ~ 3 Gb) The results depends heavily on the number of workers. The optimal number of workers is between 9-12.

- In this study we also demonstrated a property of the **indirect and direct TSQR** methods. Where their performances are swapped depending on the dimension of the dataset. This result will be examined later.

HIGGS dataset

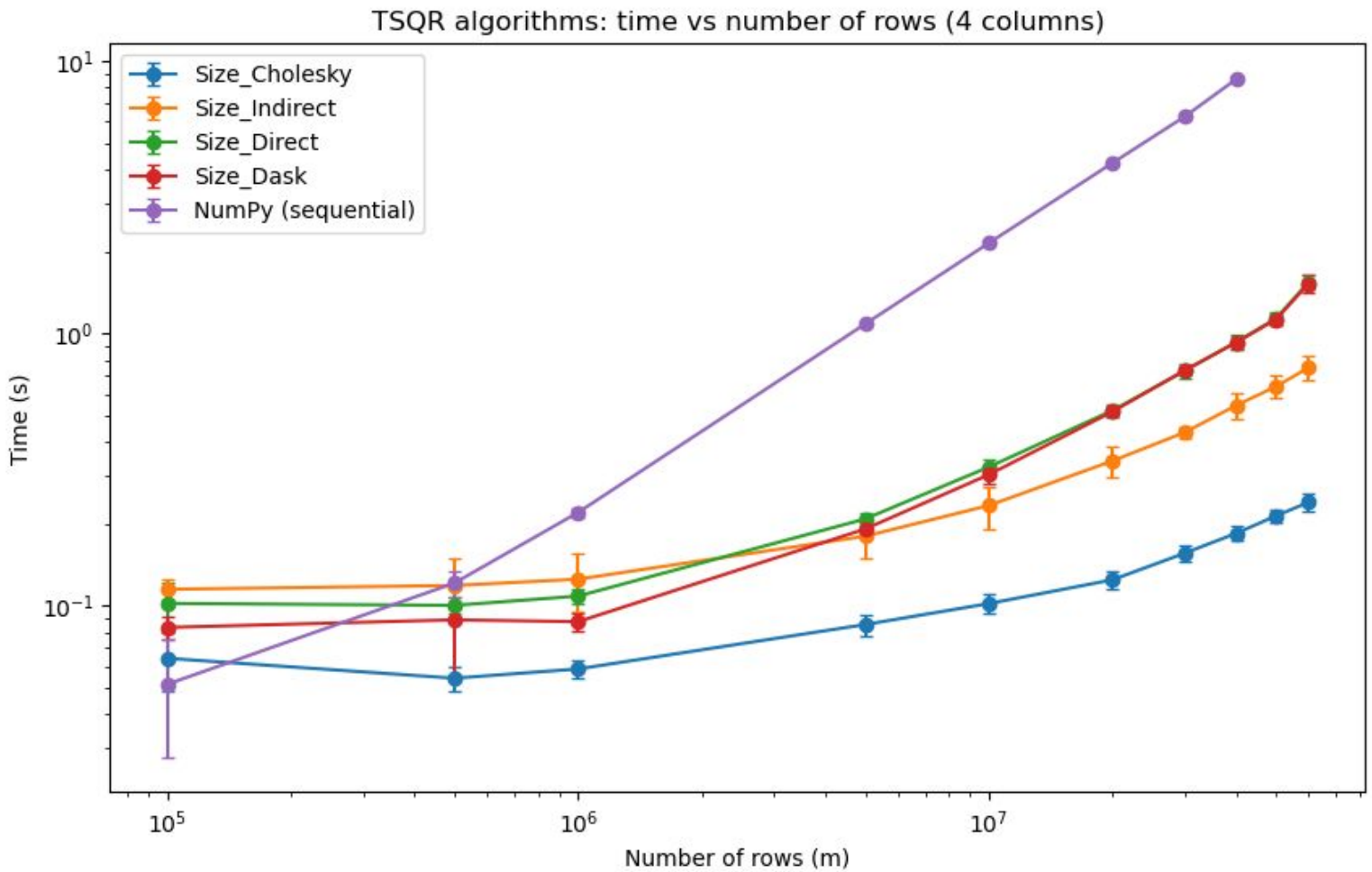
BENCHMARK 3:

Matrix size

*This benchmark measures how the **execution time** of the TSQR algorithms scales with **increasing row counts** for tall-and-skinny matrices, comparing parallel Dask-based methods to sequential NumPy QR.*

`./Benchmark_size.ipynb`

Benchmark 3: Matrix size



The asymptotic execution time of all TSQR algorithms increases roughly linearly with the number of rows, reflecting the time complexity of their steps:

Matrix-matrix multiplication	$\mathcal{O}(mn^2)$
Matrix inversion	$\mathcal{O}(n^3)$
QR decomposition	$\mathcal{O}(mn^\omega)$

Synthetic dataset
12 workers

Benchmark 3: Matrix size

Our in-memory benchmark was designed to show the performance differences between TSQR methods and the state-of-the-art sequential QR algorithm.

While sequential calculation limits the matrix size, bigger matrices only amplify the performance gap with parallel implementations

The benchmarks by Benson et al. (*2013*) use much larger matrices, where disk read/write times become the main bottleneck, allowing them to create a performance model based on distributed computing platform specifications

Rows	Cols.	HDFS Size (GB)	Cholesky	Indirect TSQR	Direct TSQR
			job time (secs.)		
4,000,000,000	4	134.6	2931	4076	6128
2,500,000,000	10	193.1	2508	2509	4035
600,000,000	25	112.0	1098	1104	1910
500,000,000	50	183.6	1563	1618	3090
150,000,000	100	109.6	921	954	2154

Table VI, Benson et al. (*2013*)

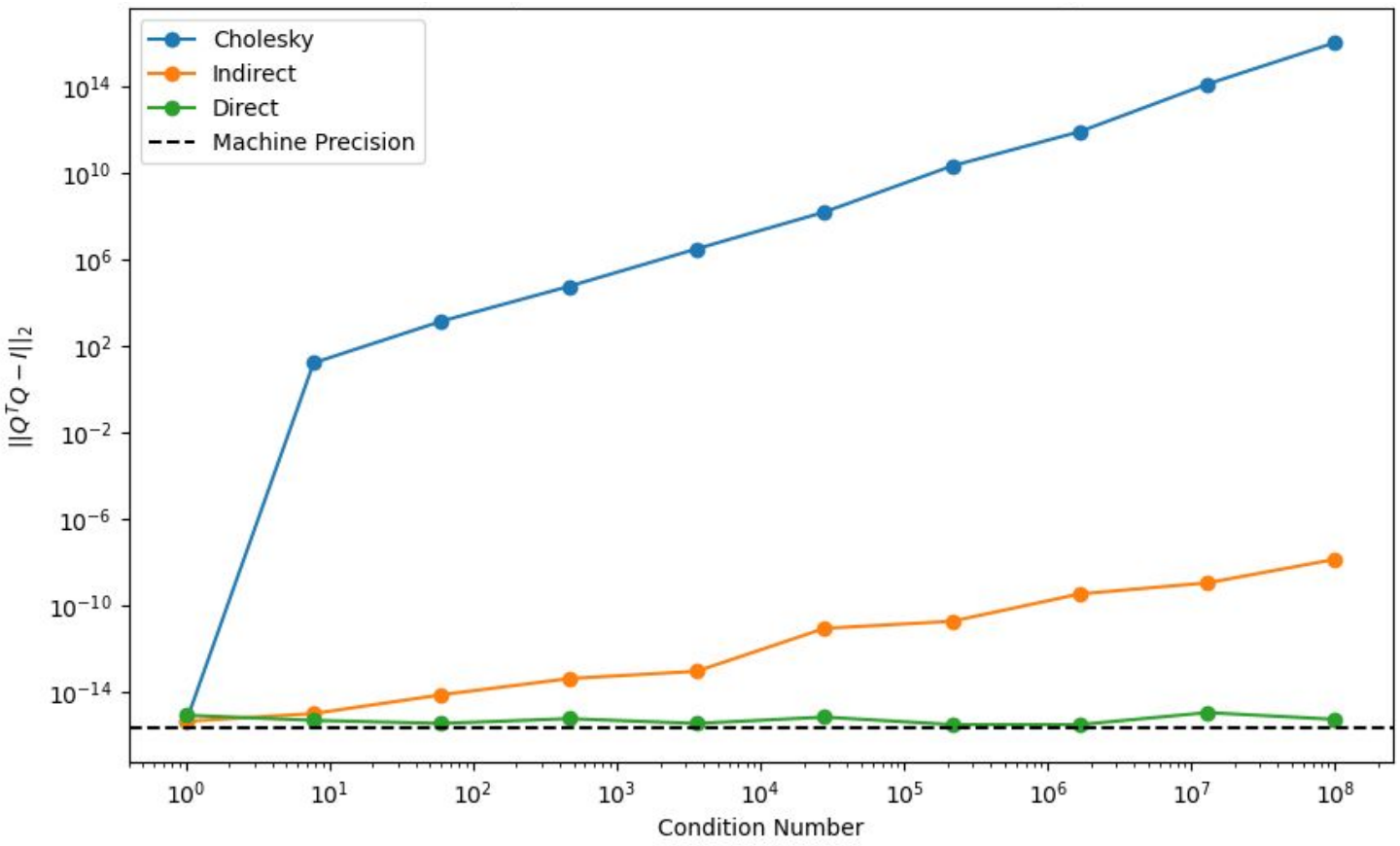
BENCHMARK 4:

Stability

This benchmark evaluates how the ***orthogonality error*** *of different TSQR algorithms depends on the condition number of the input matrix, and tests* ***iterative correction methods*** *for improving stability on ill-conditioned matrices.*

`./Benchmark_cond.ipynb`

Benchmark 3: Stability



The condition number of a matrix measures how sensitive its output is to errors in the input

$$\kappa(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$$

When the matrix A is ill-conditioned, the matrix inversion in the Cholesky and Indirect methods can produce a non-orthogonal matrix Q

Synthetic dataset
1000×4
3 workers (local)

Benchmark 3: Stability

Iterative refinement is a simple and fast post-processing technique that can improves the orthogonality of Q

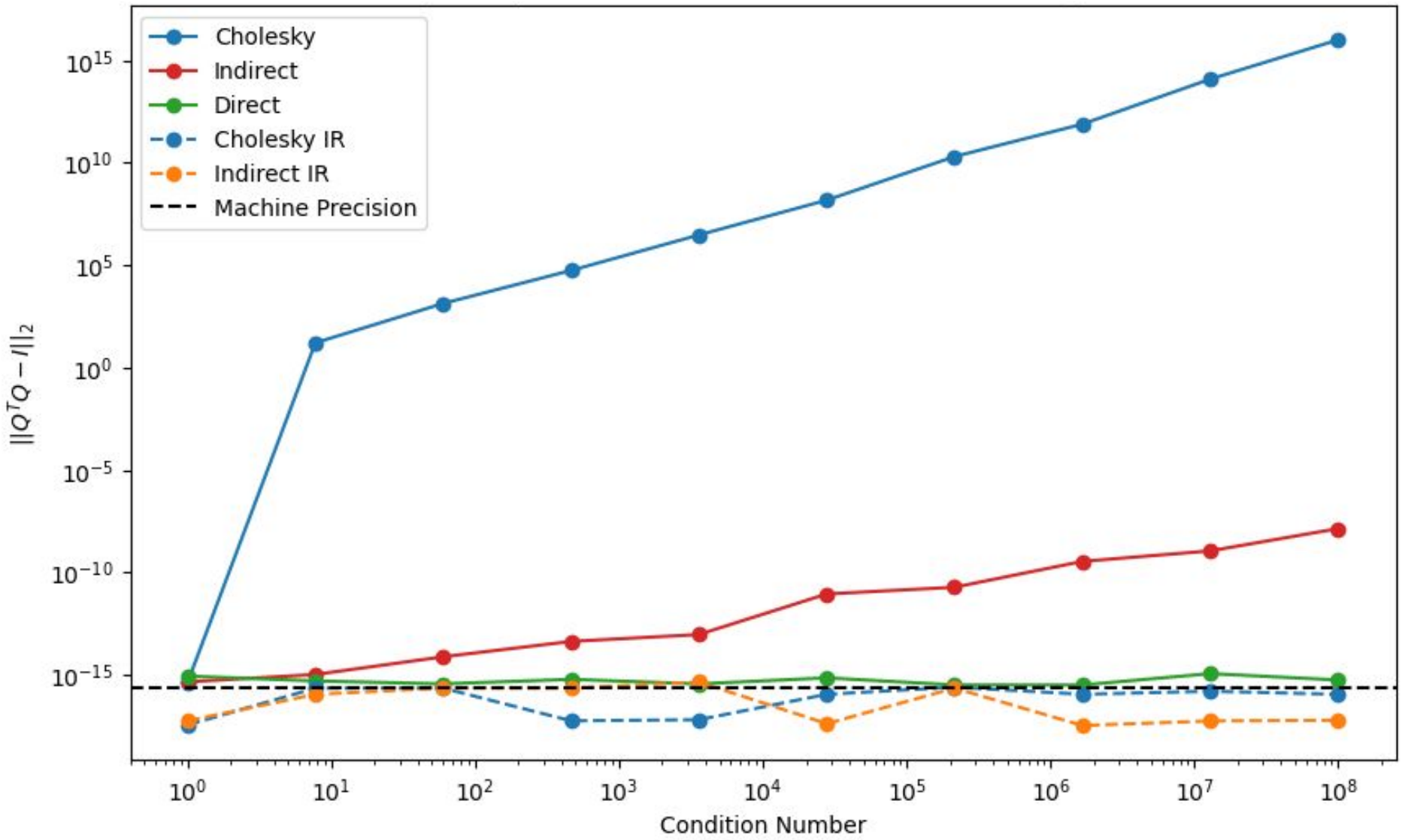
$$\left.\begin{array}{rl} \text{near-orthogonal} & Q = Q_0 + \Delta Q \\ \text{orthogonality error} & E = \mathbb{I} - Q_0^T Q_0 \end{array}\right\} \quad Q \leftarrow Q + \alpha Q E$$

./Benchmark_cond.ipynb

```
def iterative_refinement(Q : np.ndarray, n_iter : int):
    for _ in range(n_iter):
        E = np.eye(n) - Q.T @ Q
        error_norm = np.linalg.norm(E)
                alpha = min(0.5, 1.0 / (1.0 + error_norm))
        Q = Q + alpha * Q @ E
    return Q
```

Iterative refinement with adaptive step α

Benchmark 3: Stability



Synthetic dataset
1000×4
3 workers (local)