2024 SIAM Conference on Parallel Processing and Scientific Computing, March 5-8, 2024, Baltimore, USA

Improving stability and performance: Integration of novel CholeskyQR2 into the ChASE library

Nenad Mijić[*], **Davor Davidović**[*], Xinzhe Wu[+], Edoardo di Napoli[+]

[*]Ruđer Bošković Institute, Croatia
[+]Jülich Supercomputing Centre, Germany

CROATIAN COMPETENCE CENTRE FOR HPC

Content

1. ChASE eigenvalue solver

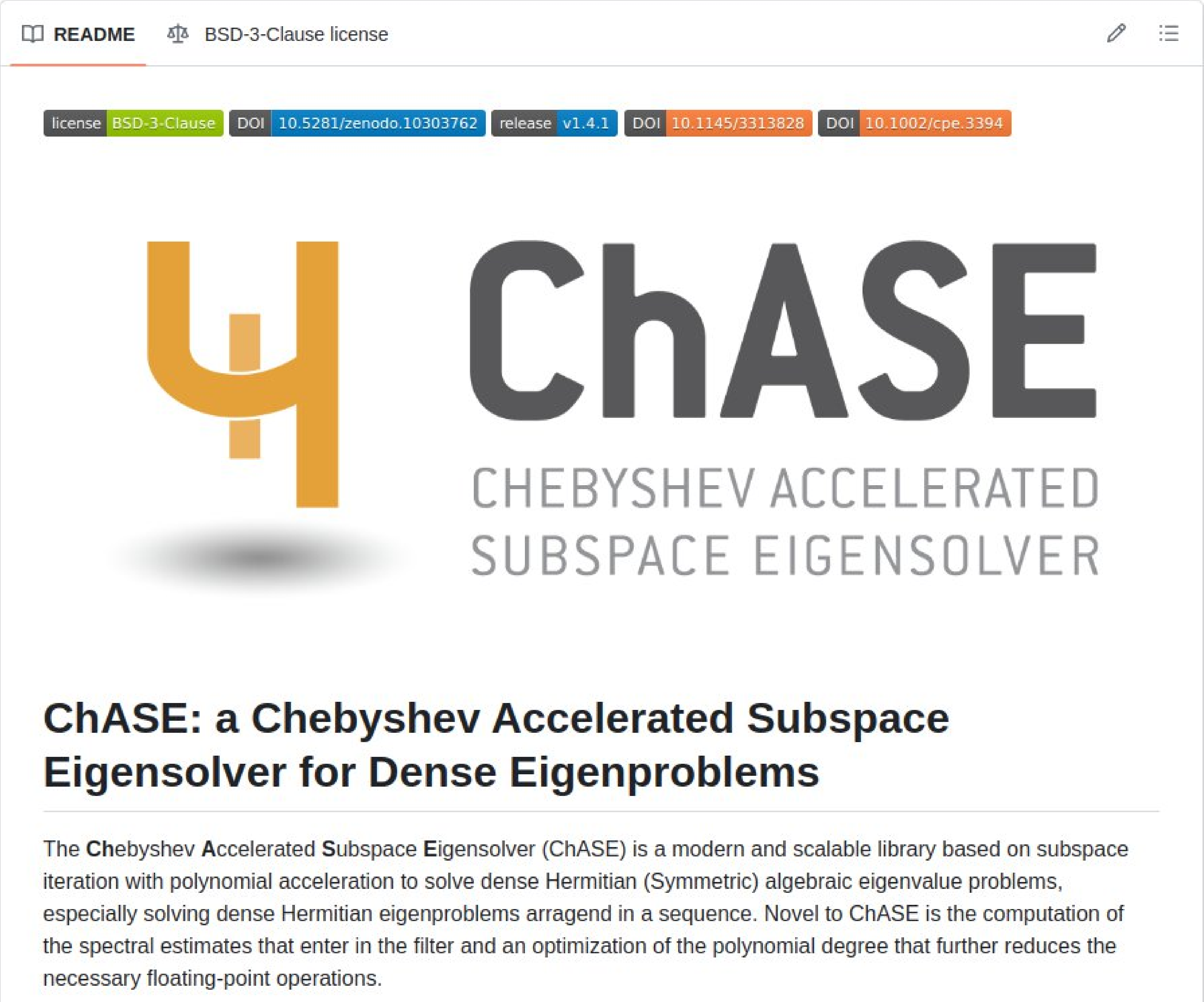
2. Parallelisation model and scalability issues

3. QR factorization → CholeskyQR

4. Improving the numerical stability of CholeskyQR

5. Preliminary results

ChASE library


README BSD-3-Clause license

license BSD-3-Clause | DOI 10.5281/zenodo.10303762 | release v1.4.1 | DOI 10.1145/3313828 | DOI 10.1002/cpe.3394

ChASE: a Chebyshev Accelerated Subspace Eigensolver for Dense Eigenproblems

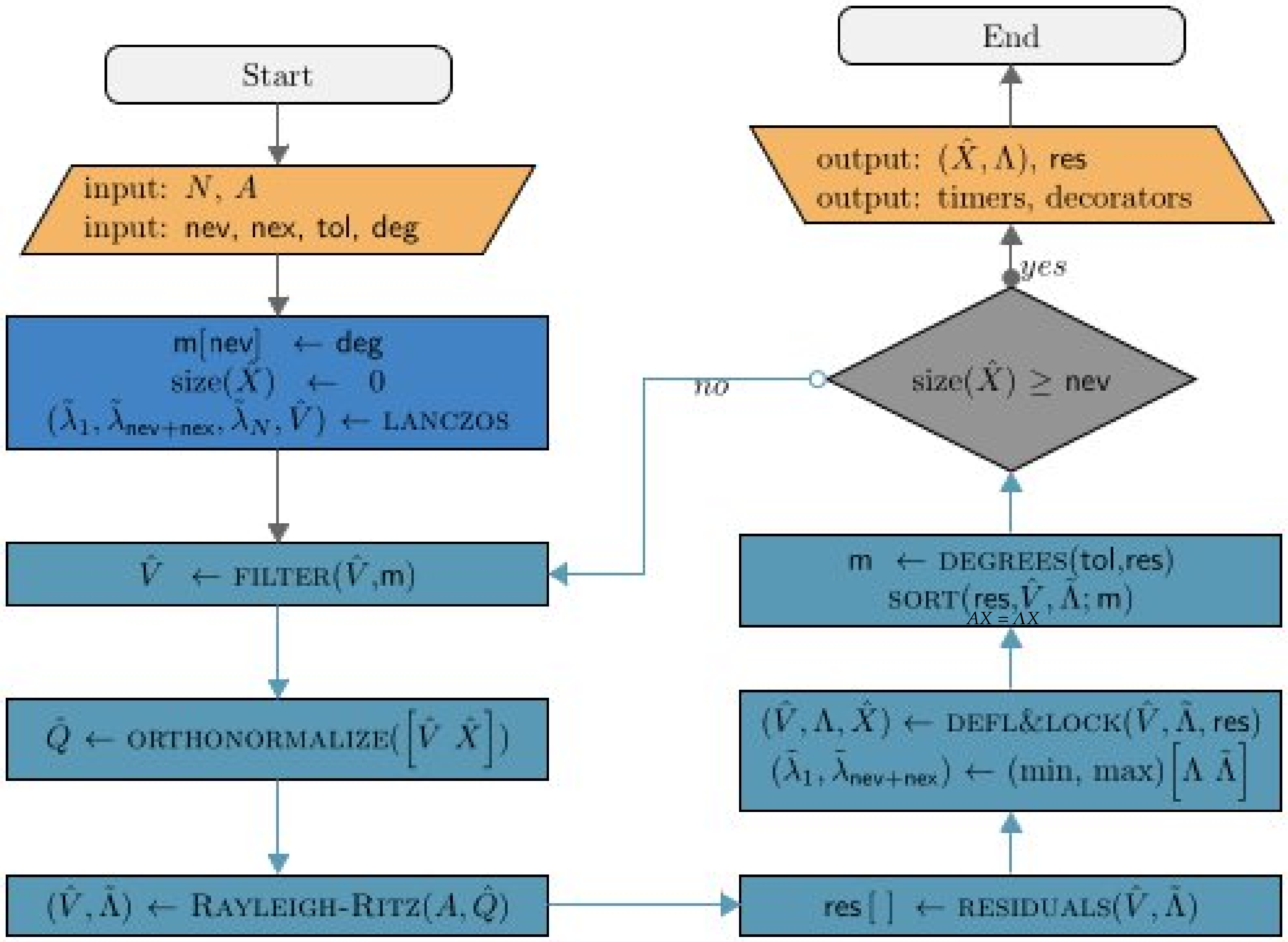
The **Ch**ebyshev **A**ccelerated **S**ubspace **E**igensolver (ChASE) is a modern and scalable library based on subspace iteration with polynomial acceleration to solve dense Hermitian (Symmetric) algebraic eigenvalue problems, especially solving dense Hermitian eigenproblems arragend in a sequence. Novel to ChASE is the computation of the spectral estimates that enter in the filter and an optimization of the polynomial degree that further reduces the necessary floating-point operations.

[1] https://dl.acm.org/doi/10.1145/3313828

- Chebyshev polynomial with **degree optimization** to accelerate convergence[1]
- Accurately approximates the **extremal eigenvalues** of dense **Hermitian** eigenproblems
- Particularly effective on solving **a sequence** of correlated **eigenproblems**
- Support for homogeneous and heterogeneous architectures with **shared and distributed** memory
- Modern C++ interface: easy-to-integrate in application codes
- https://github.com/ChASE-library/ChASE

The algorithm

- Iterative solver for standard symmetric/Hermitian eigenvalue problem:
- **$A\ X = \Lambda\ X$**
- where only a portion on eigenvalues are required
- Mostly cast in terms of BLAS-3 operations



The algorithm

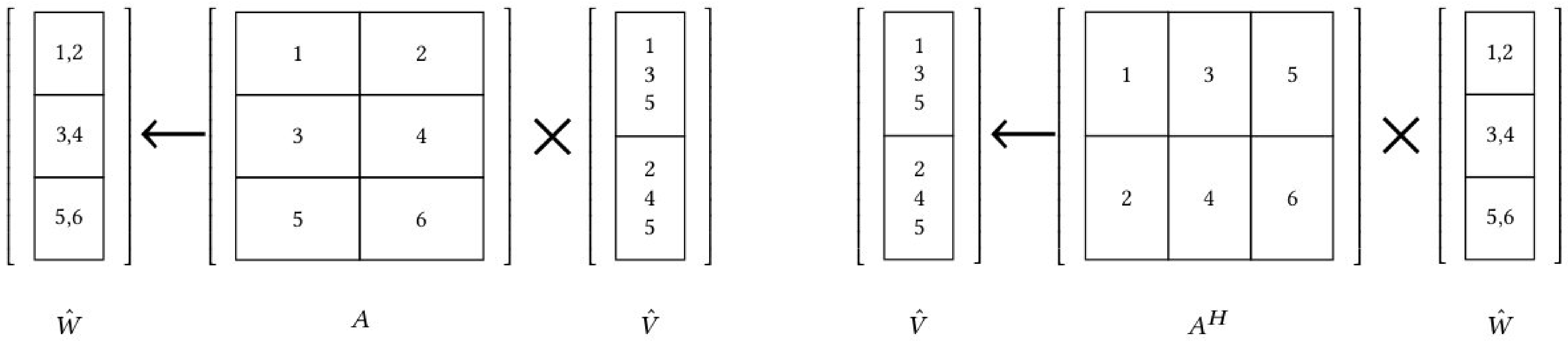
Algorithm step	Description	Math operation
FILTER	$V_{i+1} = A * V_i + V_{i-1}$	GEMM
ORTHONORMALIZE	$[Y\ V] \rightarrow Q\ R$	QR
RAYLEIGH-RITZ	TSOLVE (Q^T A Q) $\rightarrow V, \Lambda$	GEMM + EIG Solver
RESIDUAL	Compute residual of (V, Λ)	AXPY
DEFL & LOCK	Lock converged $\rightarrow Y$	Copy/index
SORT & DEGREE	Sort(Y), update degree	log/min/max

[1]Wu, X., Davidović, D., Achilles, S. & Di Napoli, E. (2022) ChASE: a distributed hybrid CPU-GPU eigensolver for large-scale hermitian eigenvalue problems. PASC'22: Proceedings of the Platform for Advanced Scientific Computing Conference. New York, NY, USA, ACM, 9, 12 doi:10.1145/3539781.3539792.

Parallelisation model

- Input matrix A divided into 2D block layout.
- 2D MPI process grid (fixed 1 block per rank)
  - Large and contiguous matrix multiplication per MPI rank
- Hybrid CPU-GPU and CPU-only
- Column-matrices (V, W) are divided into 1D row block layout and distributed among MPI ranks (one block replicted on multiple ranks)
- Parallelism of the level of fine-tuned libraries (Lapack, ScaLapack, MKL, CUDA)

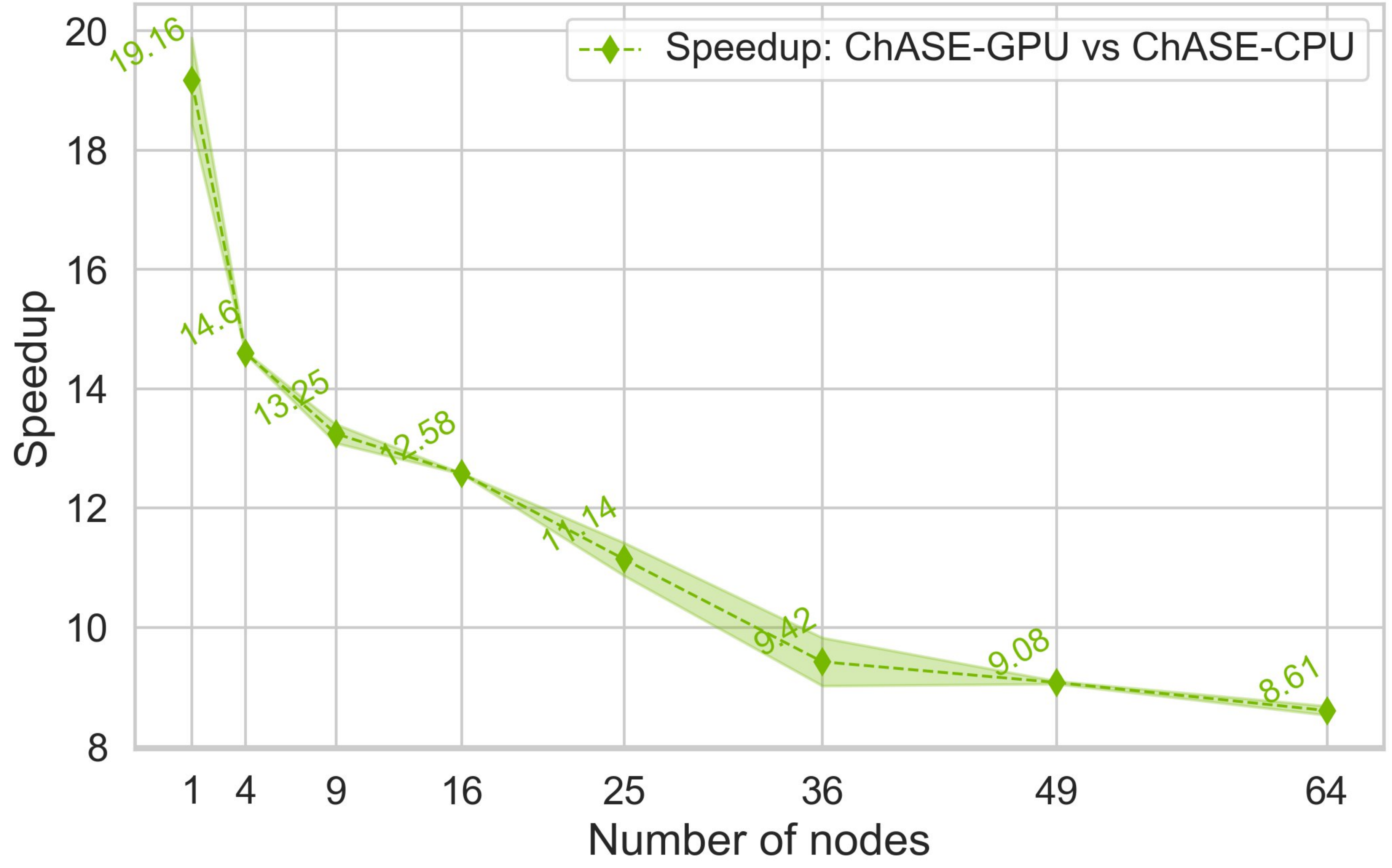
$$A_{dist} = \left(\begin{array}{c|c} A_{0,0} & A_{0,1} \\ \hline A_{1,0} & A_{1,1} \\ \hline A_{2,0} & A_{2,1} \end{array}\right), \hat{V}_{dist} = \left(\begin{array}{c|c} \hat{V}_0 & \hat{V}_1 \\ \hline \hat{V}_0 & \hat{V}_1 \\ \hline \hat{V}_0 & \hat{V}_1 \end{array}\right)$$



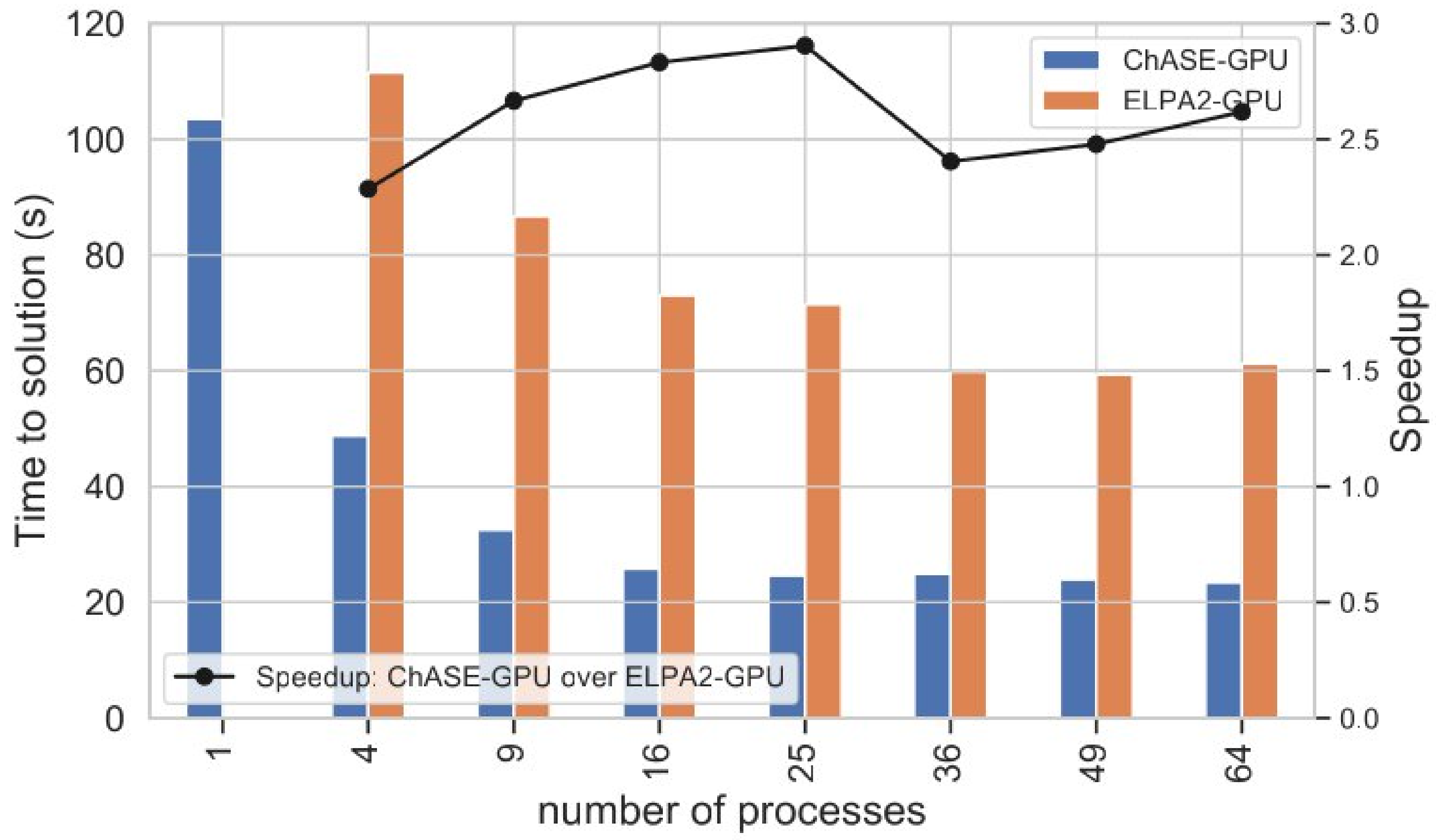
(a) Multiplication of A with $\hat{V}$ into $\hat{W}$

(b) Multiplication of A^H with $\hat{W}$ into $\hat{V}$

ChASE v1.2 speedup and total execution time

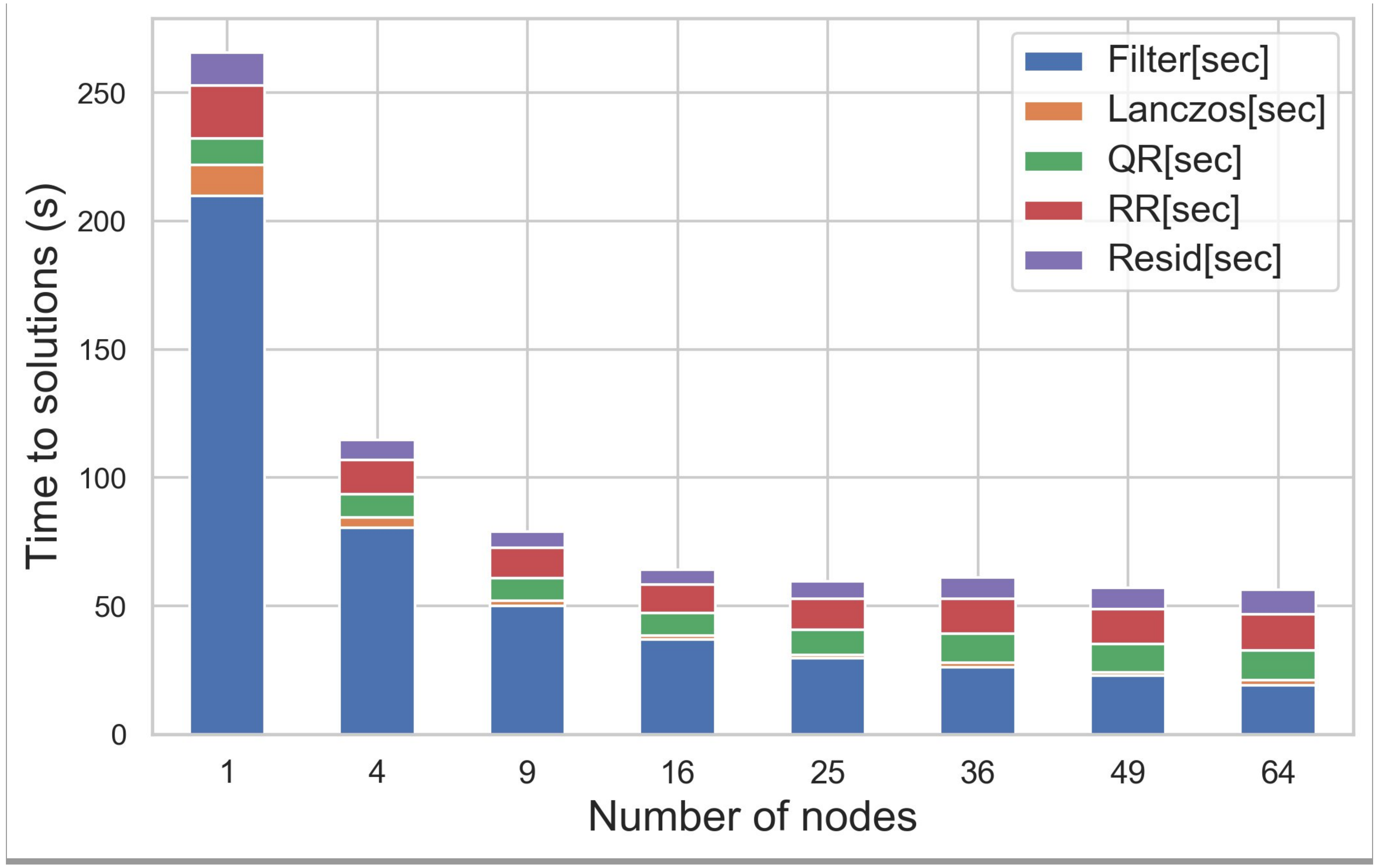


Speedup GPU vs CPU. N = 130k, NEV = 1k

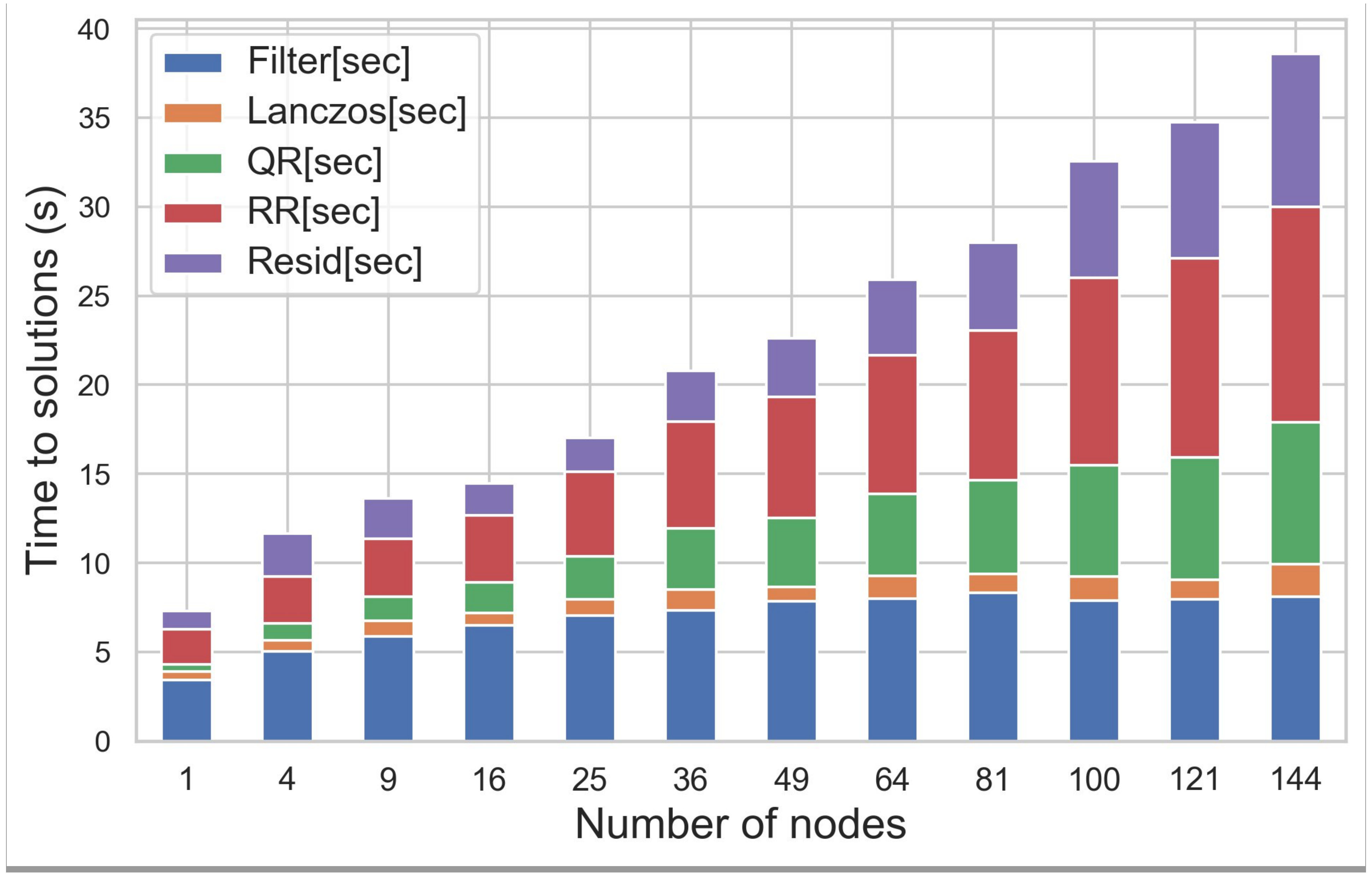


In_2O_3 N = 76k, NEV = 800

Scalability performance



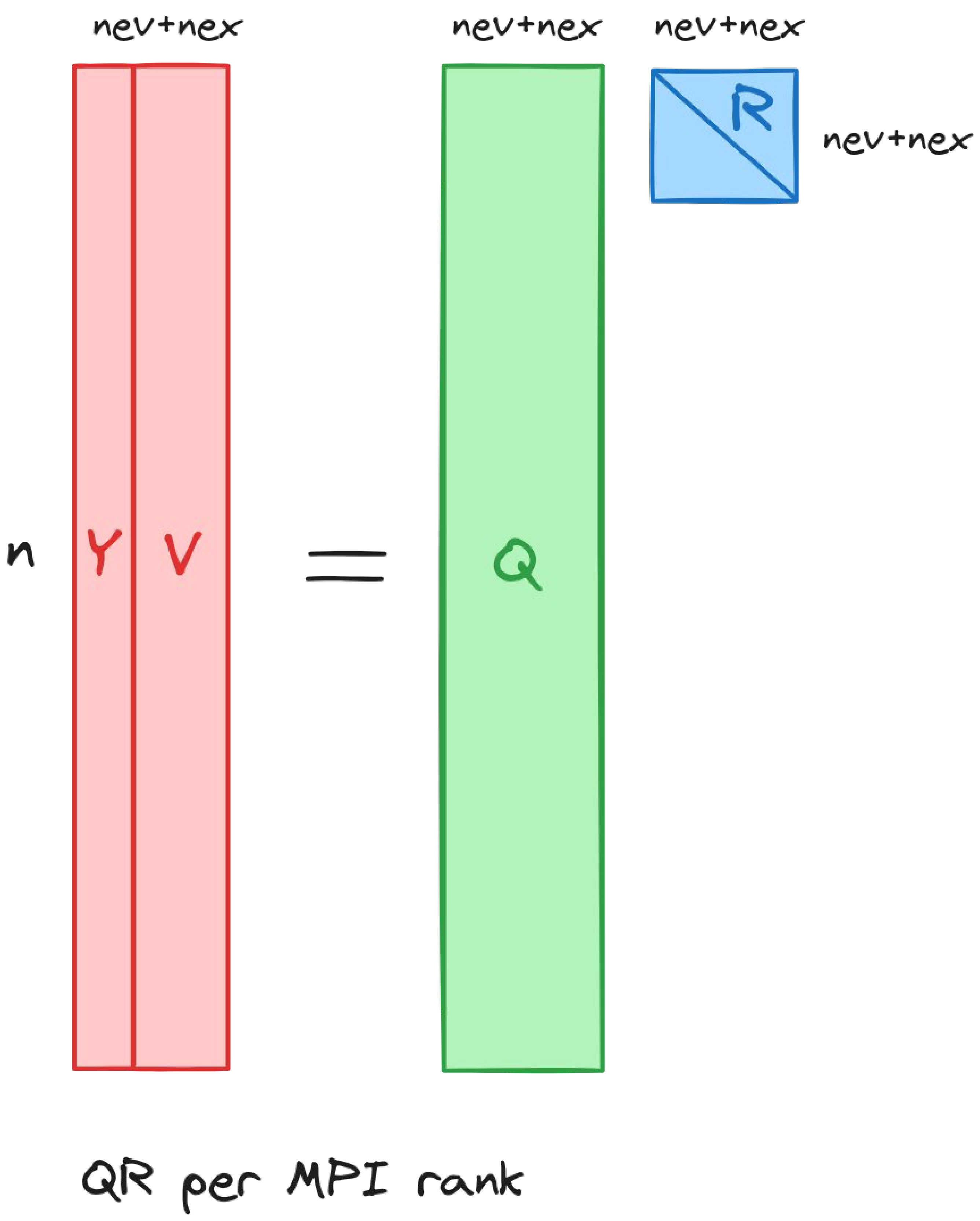
Strong scalability
Matrix size 130k, #eigenvalues = 1k



Weak scalability
Matrix size = 30k – 360k (30k per node),
#eigenvalues = 2250

Scalability issues

- Tall-and-skinny matrix
- QR factorization main issue:
  - full row-rank is participating in the calculating QR
  - does not scale with the number of nodes/GPUs
  - As N grows, the computational load per MPI increases
  - Increased memory footprint
- QR redundantly computed on each MPI rank
  - For small cases QR was small enough to be efficiently computed locally, on each MPI rank
- Original version was Householder QR factorization from the ScaLAPACK and/or cuSolver libraries
- **Block MS46**, tomorrow, 3:35 – 5:45, Xinzhe Wu: Advancing Chase Library Towards Exascale Applications on Distributed Multi-GPUs and ARM-based Systems



QR factorization

- Replace QR with a distributed implementation
- Improve the scalability with the number of nodes
- ScaLAPACK → no GPU support
- TSQR → QR factorization for tall-and-skinny matrices
  - Parallel algorithm but expensive
  - Expensive - > especially if the orthogonal matrix Q is required

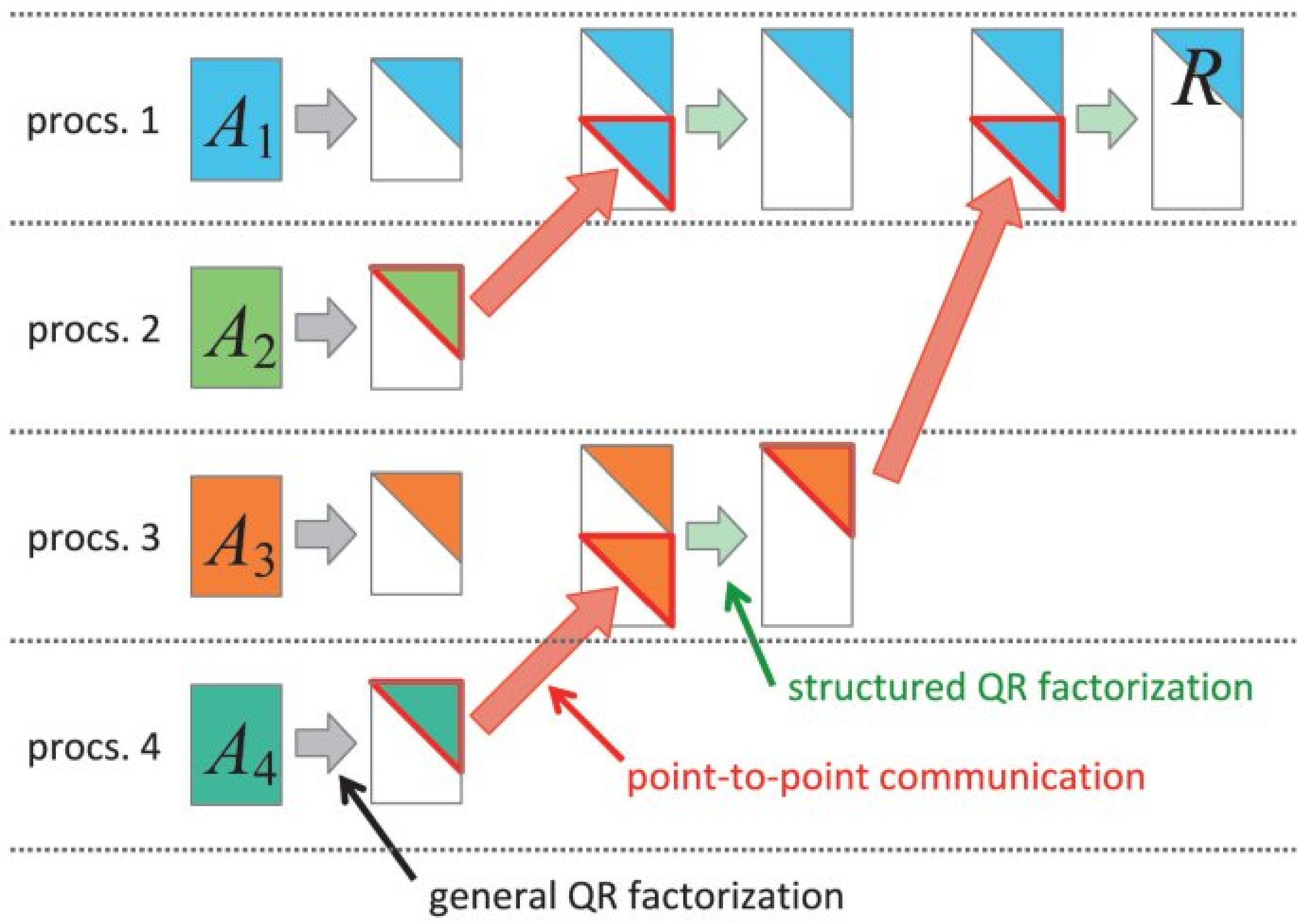


Figure taken from https://link.springer.com/chapter/10.1007/978-3-031-29927-8_22

CholeskyQR

- Tall-and-skinny QR factorization
- Simple algorithm that can be fully cast in terms of **BLAS-3** operations
- Easy parallelisation → high performance
- Drawbacks:
  - Can not produce a fully orthogonal matrix Q
  - Numerically unstable for ill conditioned matrices (cond(A) > 10^8)
- **Solution**: repeat the process twice (or multiple times) → CholeskyQR2

CholeskyQR (CQR)

$G = A^T A$ ➡ $\mathrm{Chol}(G) = R^T R$ ➡ $Q = AR^{-1}$

Algorithm CholeskyQR2 (CQR2)

Input: $A \in \mathbb{R}^{m \times n}$
Output: $Q \in \mathbb{R}^{m \times n}$ orthogonal and $R \in \mathbb{R}^{n \times n}$ upper triangular matrix
1: $[Q_1, R_1] := CQR(A)$
2: $[Q, R_2] := CQR(Q_1)$
3: $R := R_2 R_1$

Integration into the ChASE library

- 1D row-block and MPI grid
- Mix of Householder and CholeskyQR2
- If the condition number > 10^8 fallback to ScaLapack (Householder QR)

CholeskyQR with Gram-Schmidt

Input: $A \in \mathbb{R}^{m \times n}$, panel width b and number of panels $k = \frac{n}{b}$

Output: $Q \in \mathbb{R}^{m \times n}$ orthogonal and $R \in \mathbb{R}^{n \times n}$ upper triangular matrix

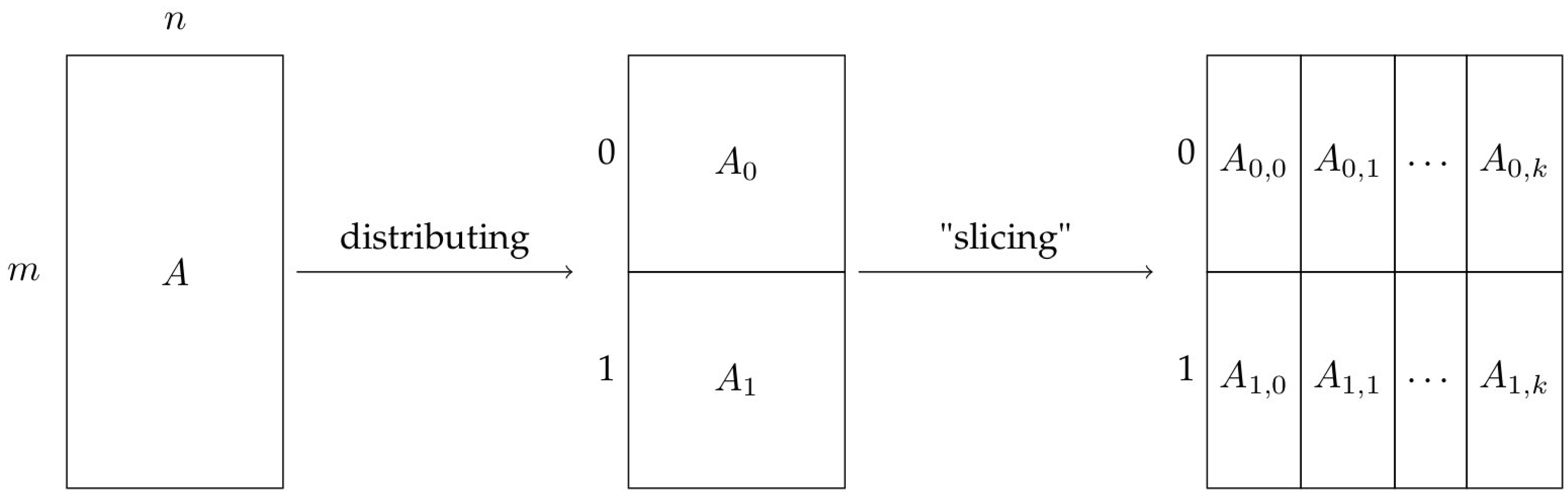
for $j = 1 \dots k$ **do**
- $W_j := A_j^T A_j$ ▷ Construct Gram matrix
- $W_j = U^T U$ ▷ Cholesky factorization
- $Q_j = A_j U^{-1}$
- $R_{j,j} = U$
- $Y := Q_j^T A_{j+1:k}$
- $A_{j+1:k} := A_{j+1:k} - Q_j Y$ ▷ Update panels
- $R_{j,j+1:k} := Y$

end for

- Modified Gram-Schmidt (MGS)
- Processed by panels of width **b** on **P** processors
- Computational cost:
- **2/3 b^2*n + n^3/3 + 4 m n^2 / P**
- Communication cost:
- **n(n+b) logP**

Parallelisation of CholeskyQR2 with Gram-Schmidt

- Fine-grain parallelism on per-MPI rank level
- Dividing row-blocks into panels



Pseudo-code – parallel version

Input: Number of processors P, $A \in \mathbb{R}^{m \times n}$ partitioned into block rows and distributed among processors, panel width b

Output: $Q \in \mathbb{R}^{m \times n}$ orthogonal and $R \in \mathbb{R}^{n \times n}$ upper triangular matrix

for j=1,2, . . . , k **do**

  $W_{p,j} := A_{p,j}^T A_{p,j}$

  $W_j := \text{MPI\_Allreduce}(W_{p,j})$ ▷ Communication

  $W_j = U^T U$

  $Q_{p,j} := A_{p,j} U^{-1}$

  $R_{j,j} := U$

  $Y_p := Q_{p,j}^T [A_{p,j+1}, A_{p,j+2}, \dots, A_{p,k}]$

  $Y := \text{MPI\_Allreduce}(Y_p)$ ▷ Communication

  $[A_{p,j+1}, \dots, A_{p,k}] := [A_{p,j+1}, \dots, A_{p,k}] - Q_{p,j} Y$

  $[R_{j,j+1}, R_{j,j+2}, \dots, R_{j,k}] := Y$

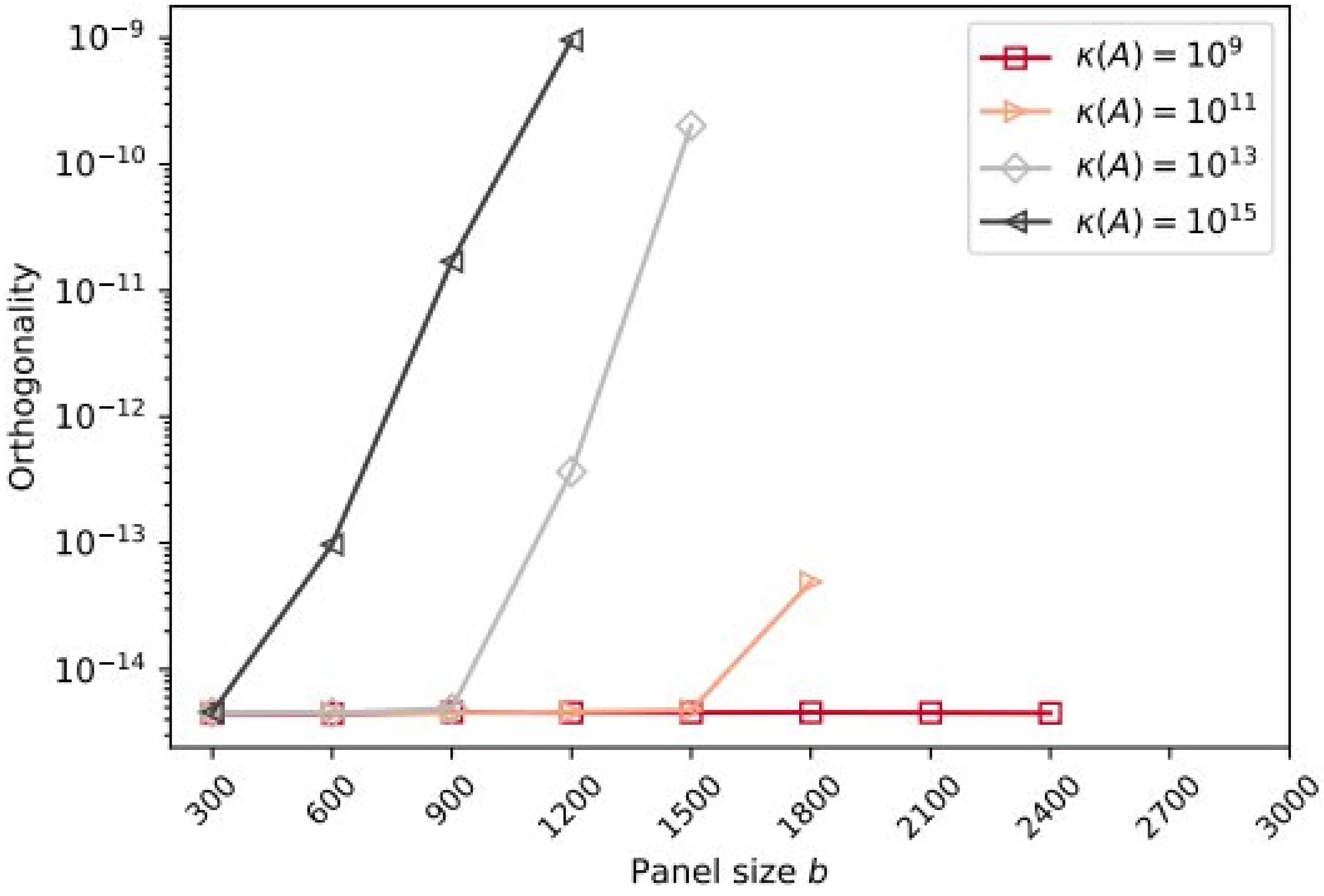
end for

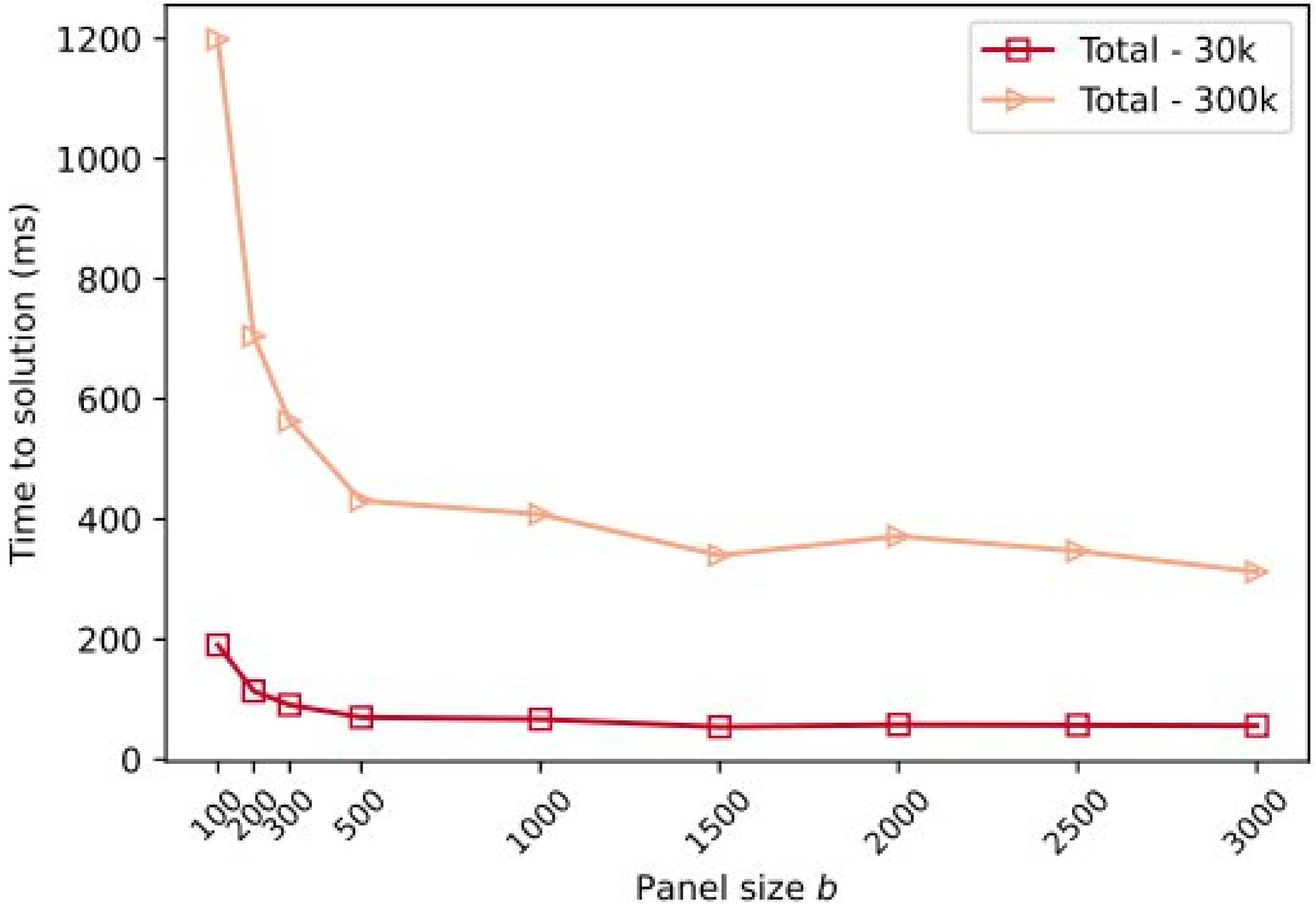
- Two collective communication calls per iteration (panel)
- Panel width **b** is the main performance and stability factor

$\text{cond}(A_i) = 10^{10} \rightarrow \text{cond}(A_i^T A_i) = 10^{20}$

- Smaller b decreases computational cost, but increases the communication (#words)
- Tradeoff between the communication and computation

Performance w.r.t. the panel width

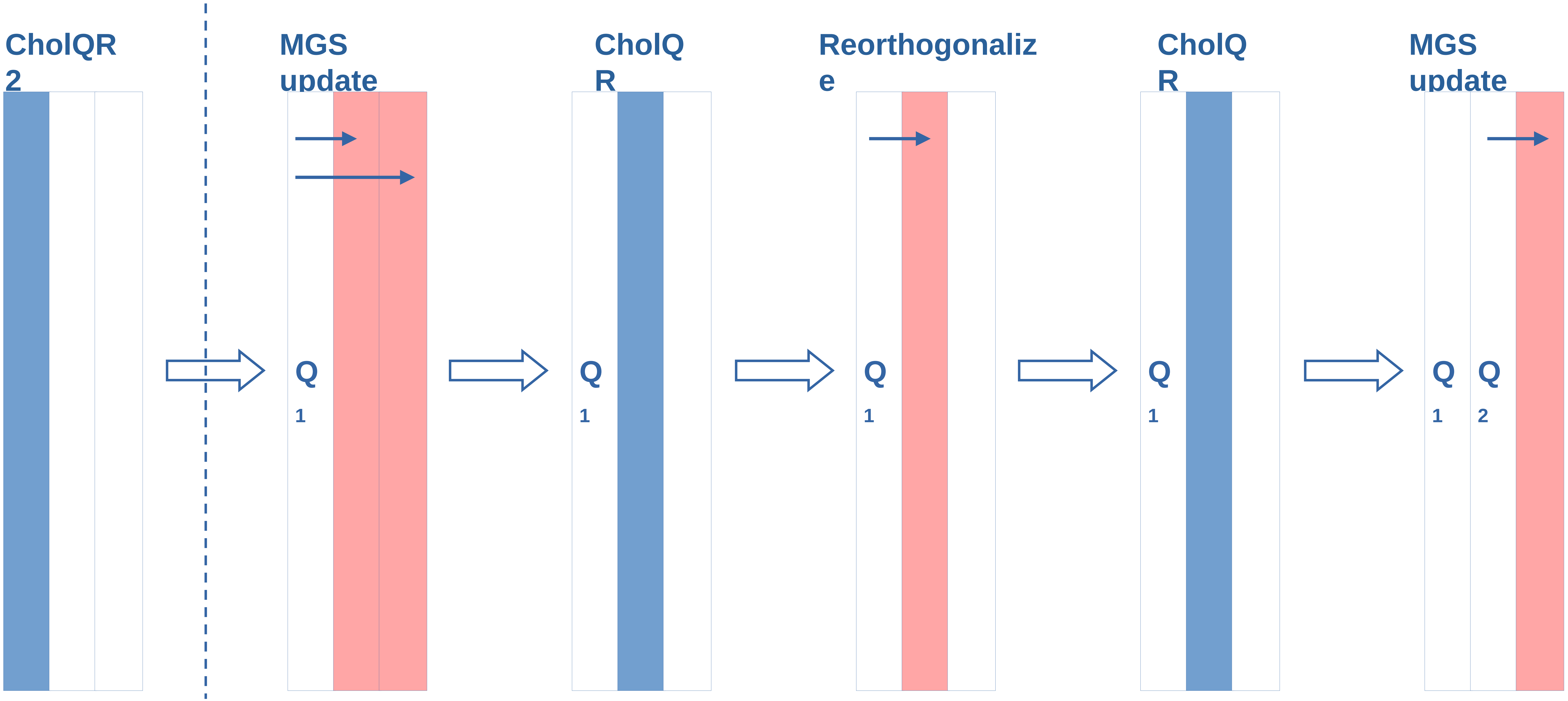




Distributed CholeskyQR2 with Gram-Schmidt

- Smaller panel width **decrease computational cost** in constructing the Gram matrix, but **increases the communication** in Gram-Schmidt re-orthogonalization part
- Stability of the algorithm depends on the panel width b → constructing the Gram matrix squares the condition number!
- Tradeoff between the computation and communication → **panel width b!**

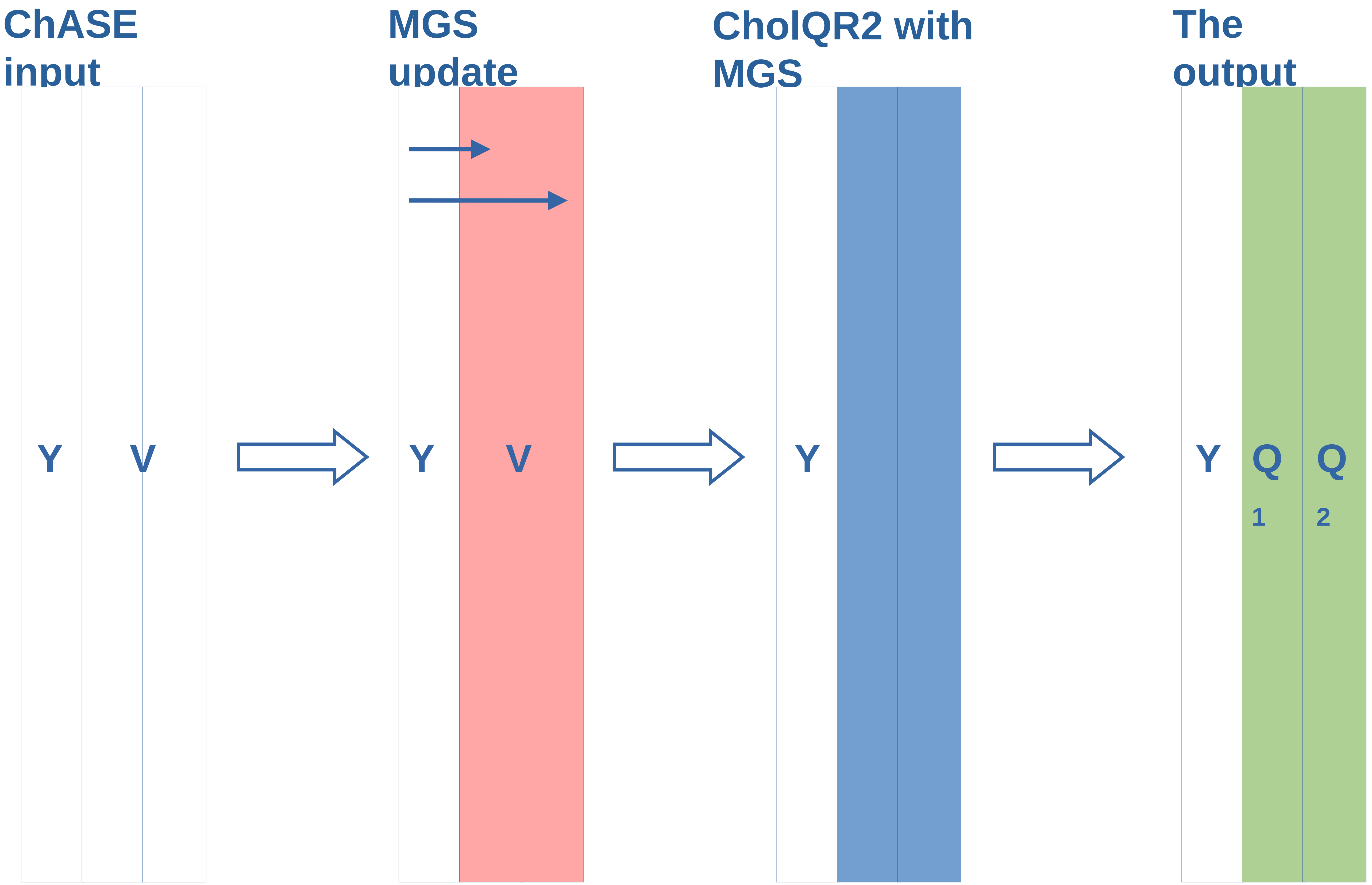
Modified CholeskyQR2 with MGS



Integration with ChASE

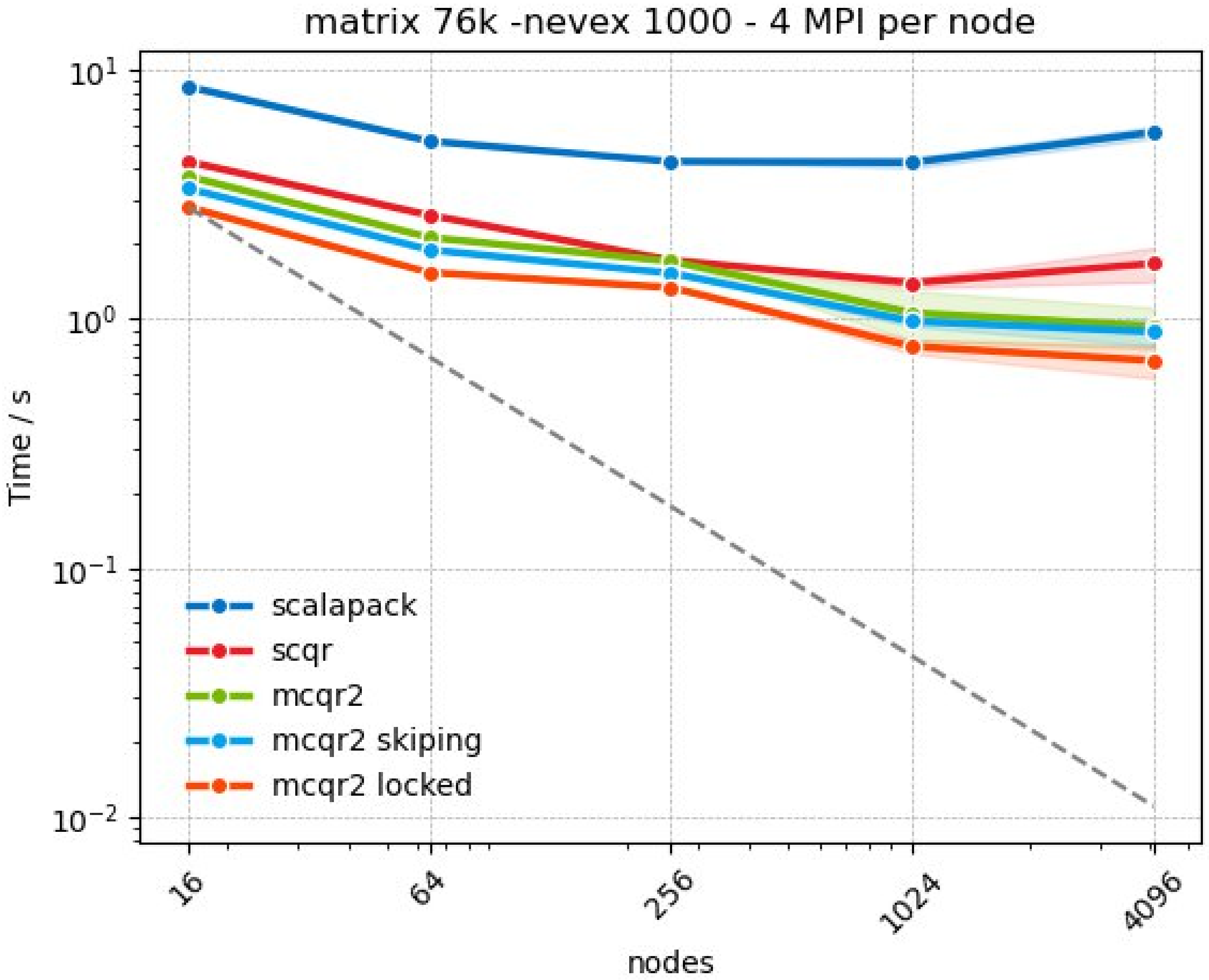
- Integrate in the existing ChASE 2D MPI grid using only the column communicators
- The CholeskyQR2 with MGS naturally brings the possibility to avoid re-orthogonalization of the already converged vectors in Y:
- [Y V] = Q R
- The first step is to apply the already computed Q (Y) to the vectors in V panel (Gram-Schmidt re-orthogonalization) and then processed with modified CholeskyQR2 with GS by panels

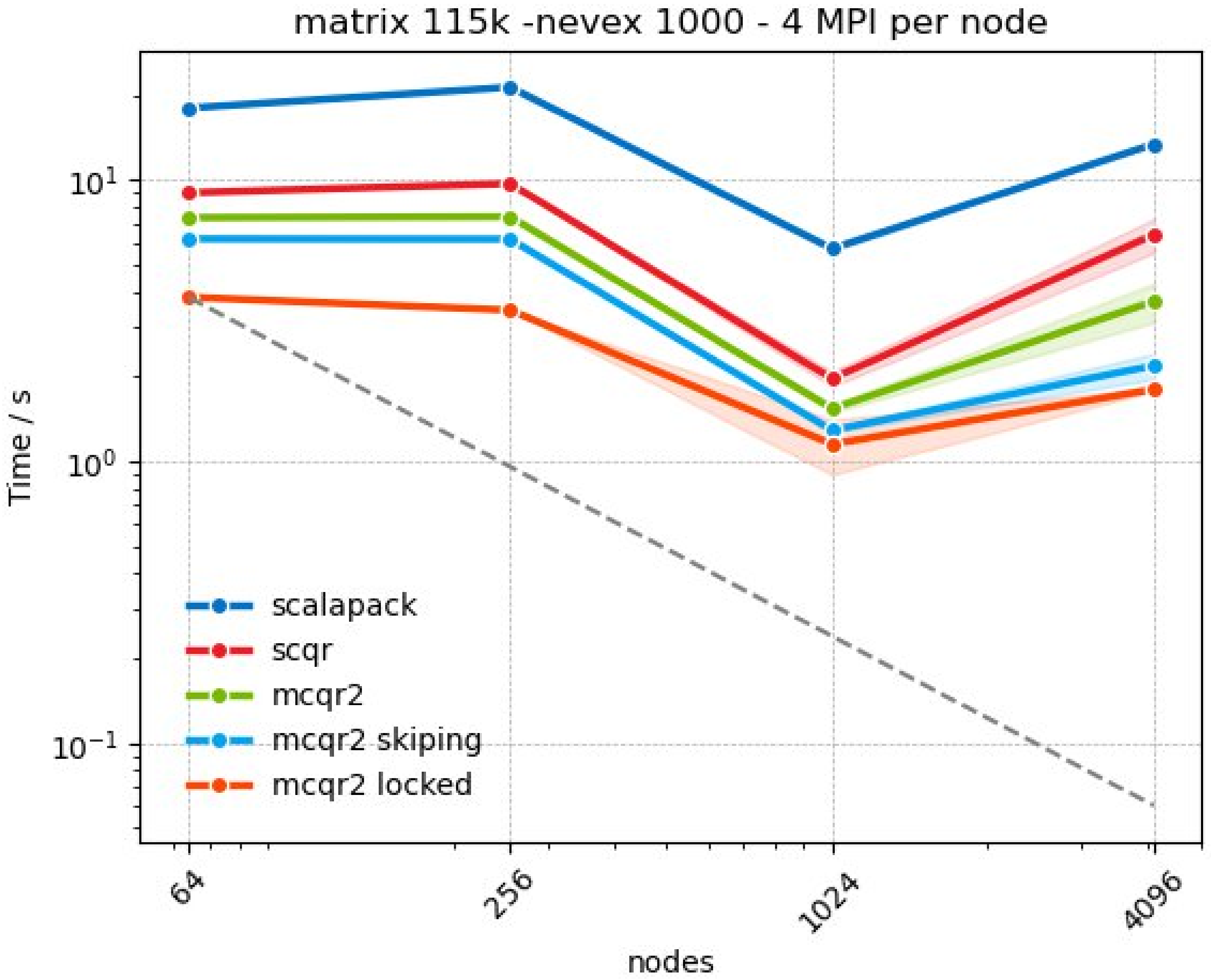
Integrate into the ChASE algorithm



- Effectively implements the updated QR factorization
- The first step → orthogonalize the filter vectors V against already converged eigevectors in Y
- MPI parallelisation via 1D column communicator

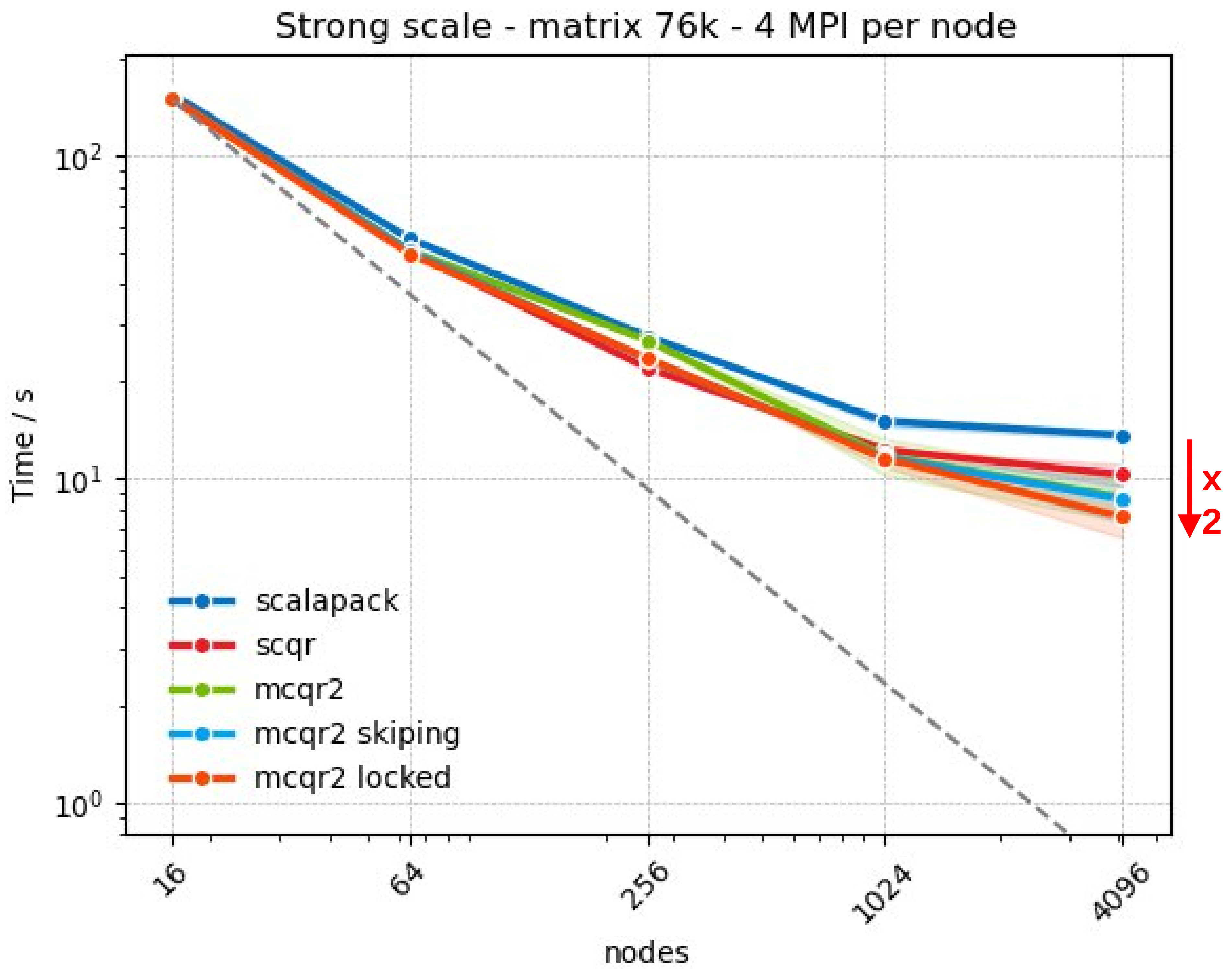
ChASE - CholeskyQR2 with MGS only

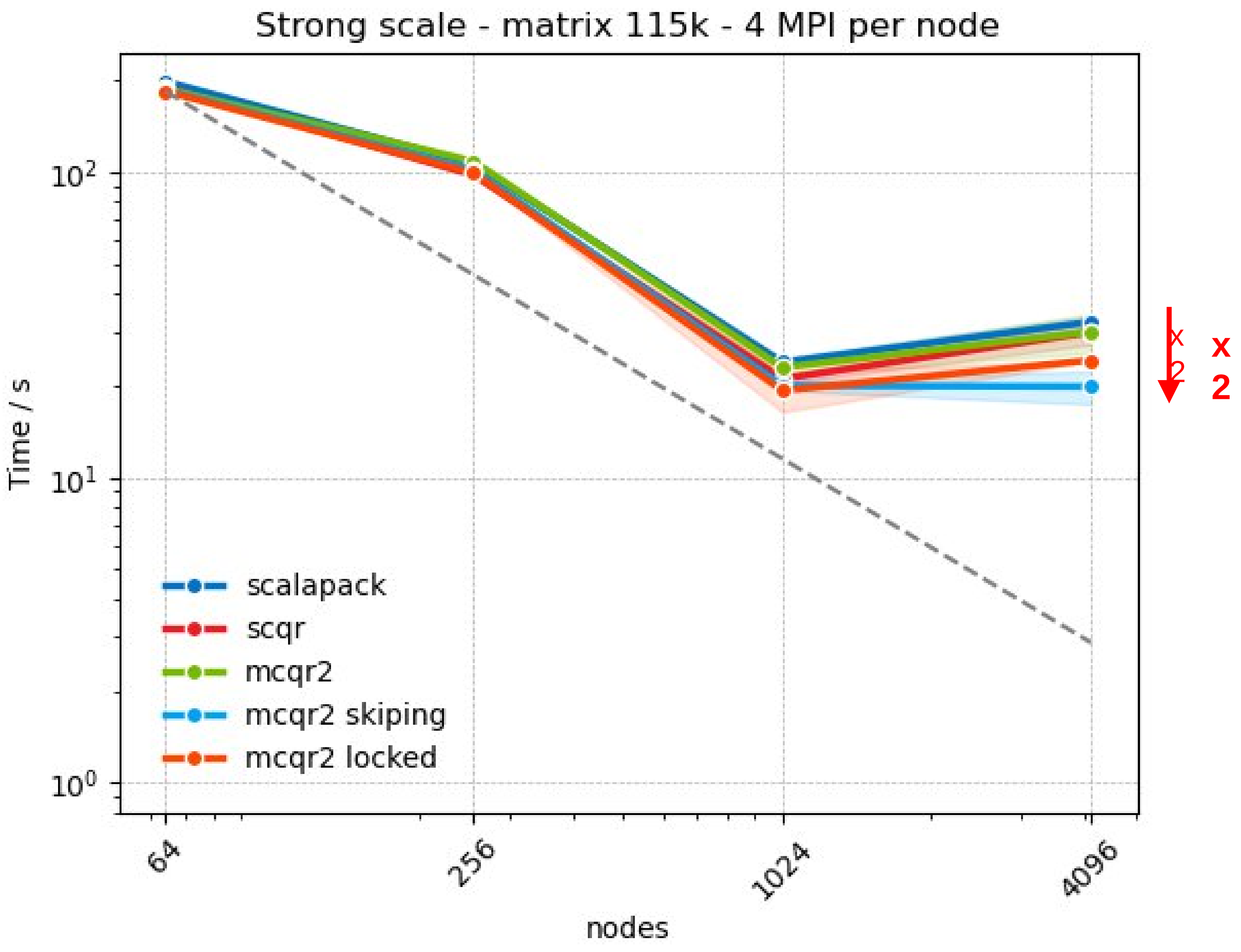




The Ni_2O_3 use-cases with sizes 115k and 76k on Fugaku in complex double precision.

ChASE All - CholeskyQR2 with MGS





Test done on Fugaku, complex double, 4096 nodes, no GPU

Conclusion

- Modified CholeskyQR2 with MGS numerically stable for extremely large condition numbers ($k(10^{15})$)
- Added support for processing QR factorization in a distributed GPU environment
- A simpler and more efficient implementation of the ChASE on distributed memory systems
- Increased scalability of the ChASE → no need to fallback to ScaLapack
- Drawbacks
  - The CholeskyQR2 with MGS won't work if singular values are highly clustered
- Future work
  - Improve stability using shifting for highly clustered singular values
  - Explore the possibility using 2D grid for processing CholeskyQR2 with Gram-Schmidt

Research group

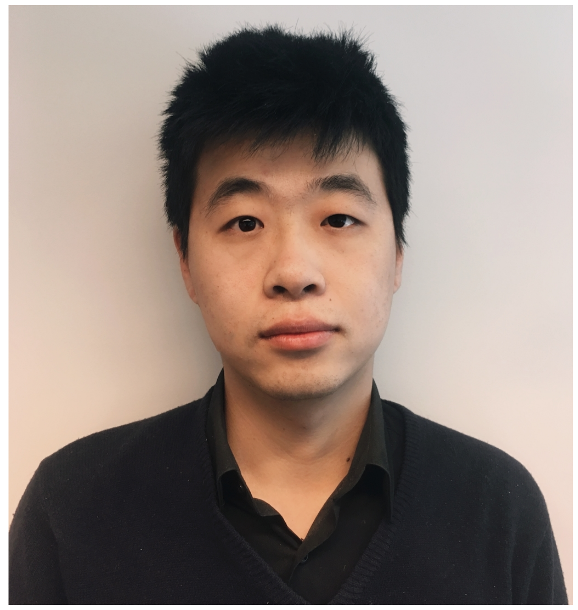
JÜLICH
Forschungszentrum

Nenad Mijić,
PhD

Abhiram
Kaushik
postdoc

Davor Davidović,
group leader

Edoardo di Napoli,
PI, group leader



Xinzhe Wu,
postdoc

Thank you!

Questions?