Low-Rank Compression in Sparse direct solvers

Grégoire Pichon, Bora Uçar & Frédéric Vivien

CNRS, INRIA, Université Lyon 1 & ENS Lyon

CR15: December 2022

gpichon.gitlabpages.inria.fr/m2if-numerical_algorithms/

Outline

Context

Sparse direct solvers

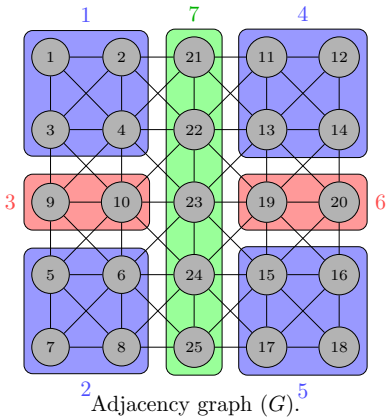
- Very robust wrt other approaches
- High time and memory complexities
- Using efficient BLAS Level 3 kernels

2D		3D	
$\sigma = \frac{1}{2}$		$\sigma = \frac{2}{3}$	
OPC	NNZ	OPC	NNZ
$\Theta(n^{\frac{3}{2}})$	$\Theta(n\ ln(n))$	$\Theta(n^2)$	$\Theta(n^{\frac{4}{3}})$

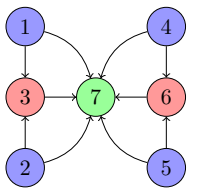
Block Symbolic Factorization

General approach

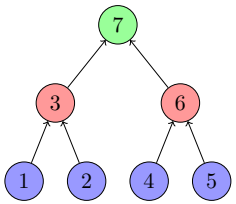
1. Build a partition with the nested dissection process
2. Compress information on data blocks
3. Compute the block elimination tree using the block quotient graph



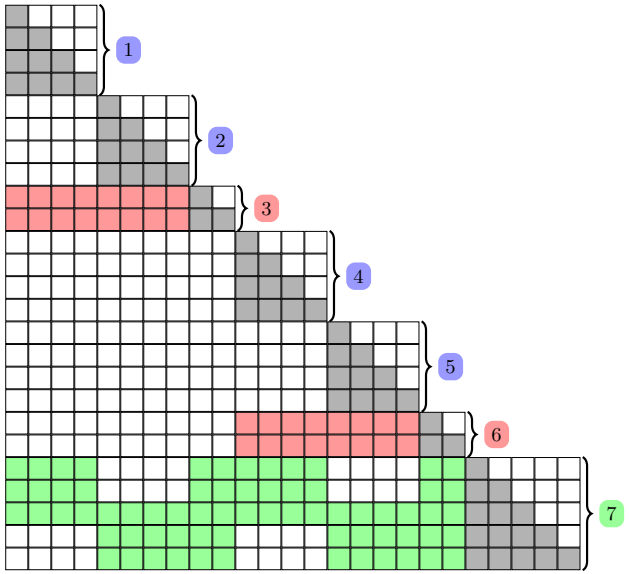
Adjacency graph (G).



Quotient graph (G^*/P) = $(G/P)^*$



Elimination tree (T).

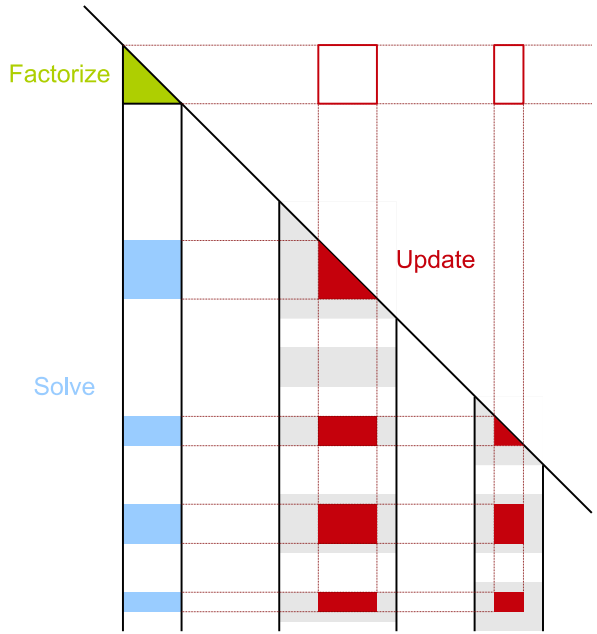


Factorized matrix (L).

Block Numerical Factorization

Algorithm to eliminate the k^{th} supernode

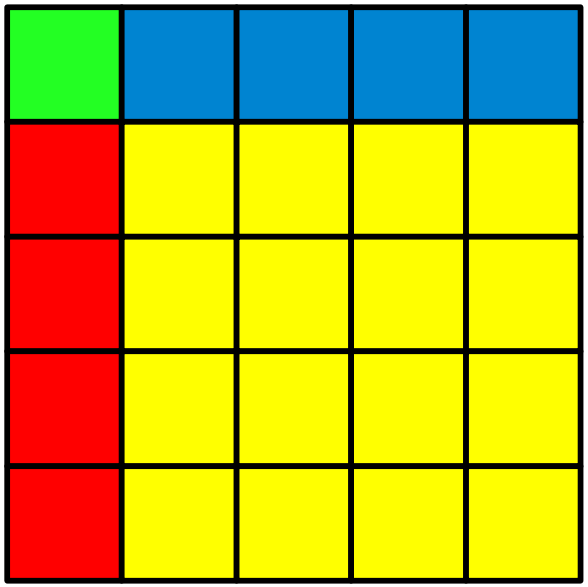
1. Factorize the diagonal block (POTRF/GETRF)
2. Solve off-diagonal blocks in the current supernode (TRSM)
3. Update the trailing matrix with the supernode contribution (GEMM)



Block LU Factorization (dense)

Algorithm 1 LU Factorization

1: **for** $k = 1$ to n **do**
2: Factorize $A_{kk} = L_{kk} U_{kk}$
3: **for** $i = k + 1$ to n **do**
4: Solve $A_{ik} = L_{ik} * U_{kk}$
5: **for** $j = k + 1$ to n **do**
6: Solve $A_{kj} = L_{kk} * U_{kj}$
7: **for** $i = k + 1$ to n **do**
8: **for** $j = k + 1$ to n **do**
9: $A_{ij} = A_{ij} - L_{ik} * U_{kj}$



Low-rank compression

Figure: Original picture of size 500×500

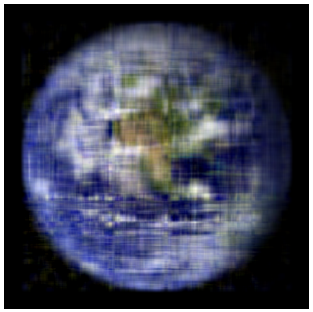


Figure: 4% of original storage cost

Figure: 20% of original storage cost

Objectives

Reduce the complexity

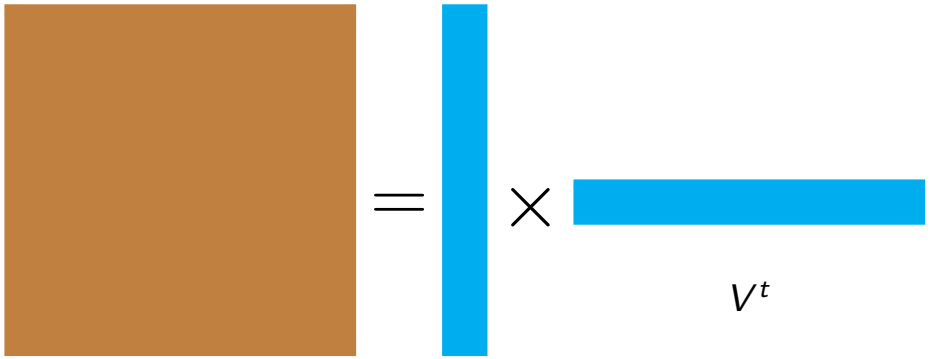
- Replace dense blocks by low-rank blocks
- Adapt underlying kernels

Similar properties

- Keep the same level of parallelism
- Use efficient underlying kernels

Outline

Low-rank compression



$U, V \in \mathbb{R}^{n \times r}$

$M \in \mathbb{R}^{n \times n}$

Storage in $2nr$ instead of n^2

Figure: Original picture, $n = 500$

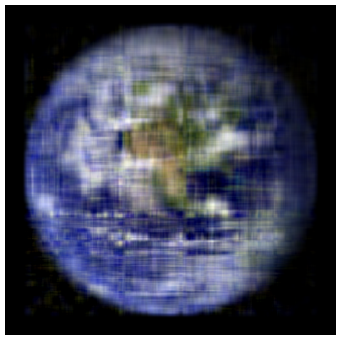


Figure: $r = 10$, 4% of original storage

Figure: $r = 50$, 20% of original storage

Rank definitions (1/2)

Rank

The rank k of a matrix A is defined as the smallest integer such that there exist matrices U and V of size $n \times k$ with $A = UV^t$

Numerical rank

The numerical rank k_ϵ of a matrix A at accuracy ϵ is defined as the smallest integer such that there exists a matrix A_ϵ of rank k_ϵ with $||A - A_\epsilon|| \leq \epsilon$

Rank definitions (2/2)

Eckart-Young theorem

Let $U\Sigma V^t$ be the SVD decomposition of A and $\sigma_i = \Sigma_{i,i}$ be its singular values. Then, $\hat{A} = U_{1:n,1:k}\Sigma_{1:k}V^t_{1:n,1:k}$ is the optimal rank-k approximation of A and $||A - \hat{A}||_2 = \sigma_{k+1}$

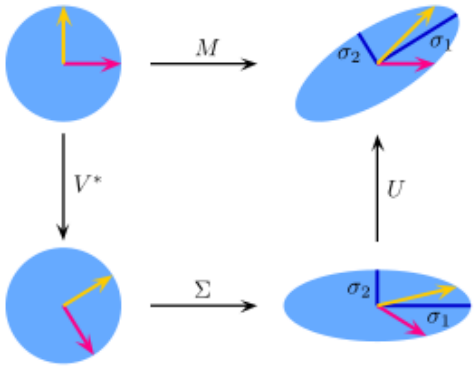
Low-rank matrix

A is said to be low-rank (for a given accuracy ϵ) if its numerical rank k_ϵ is small enough such that its rank-k_ϵ approximation requires less storage than the full-rank matrix A, *i.e.*, if $k_\epsilon(m+n) \leq mn$

Singular Value Decomposition (Figure from Wikipedia)

Idea

- Image of the unit sphere
- The singular values can be seen as the magnitude of the semiaxis of an n-dimensional ellipsoid
- Unique decomposition
- The smallest singular values represent less important data



$$M = U \cdot \Sigma \cdot V^*$$

QR Factorization (1/2)

Idea

- For rectangular matrices
- $A = QR$, A of size $m \times n$, Q of size $m \times m$, R of size $m \times n$
- Q is orthogonal, R is upper triangular

Reduced QR

- If the matrix is not full-rank, some columns of R will be made of zeroes
- Can be used to compress a matrix

QR Factorization (2/2)

How to build it ?

- Gram-Schmidt Orthogonalization
- Using Givens rotations
- Using reflections with Householder matrices

Ideas behind Householder matrices

- Cancel elements below the diagonal in R
- First step where x is the first column of A
  1. $e_1 = (1, 0, \ldots, 0)^t$
  2. $u = x - ||x||e_1$ (or $+||x||$ if $x_1 < 0$)
  3. $v = \frac{u}{||u||}$
  4. $Q_1 = I - 2vv^t$
  5. In $Q_1 A$, only the first element of the first column is non-zero

Rank-Revealing QR Factorization

Algorithm 2 QR with Column Pivoting: $[Q, R, P] = \text{QRCP}(A)$

for $j = 1, 2, ..., min(m, n)$ **do**
- $p_j = \max_{l=j-1,...,n}(||A^{(j-1)}_{:,l}||_2)$ ▷ Find the pivot
- $A^{(j-1)} = A^{(j-1)}p_j$ ▷ Apply the pivot
- $H^{(j)} = I - y_j \tau_j y_j^T$ ▷ Compute the Householder reflection
- $A^{(j)} = H^{(j)} A^{(j-1)}$ ▷ Update the trailing matrix

In practice, stop when the norm of the trailing submatrix is small enough

Compression kernels

Kernel	Complexity
Singular Value Decomposition (SVD)	$\Theta(mn^2)$
Rank-Revealing QR (RRQR)	$\Theta(mnr)$
RRQR with randomization	$\Theta(mnr)$
ACA, BDLR, CUR	$\Theta((m+n)r)$

Properties

- SVD provides the best ranks at a given accuracy with $||.||_2$
- RRQR keeps a control of accuracy, but efficiency is poor due to pivoting
- Randomization techniques are suitable to perform a rank-r approximation but may be costly for computing an accurate representation
- The accuracy of ACA/BDLR/CUR is problem dependent

Compression formats for dense matrices

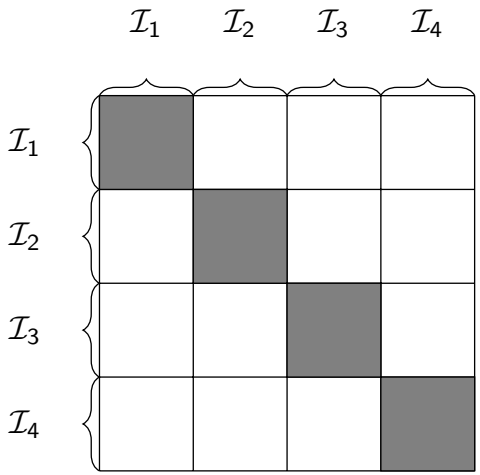


Figure: BLR clustering

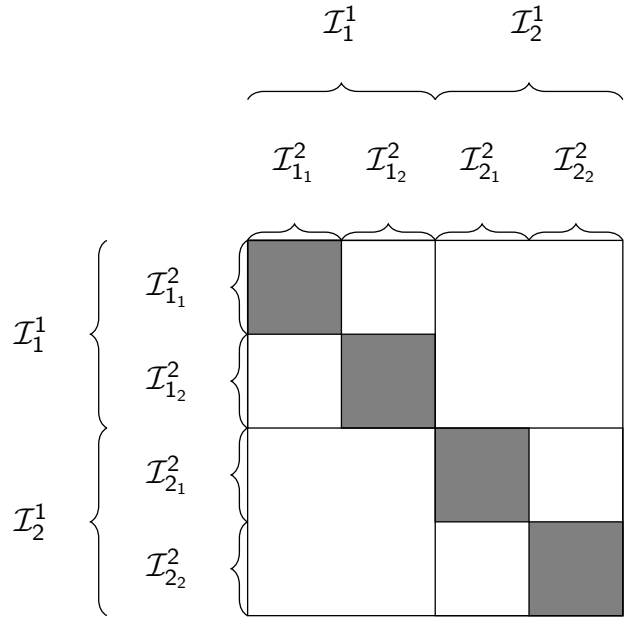
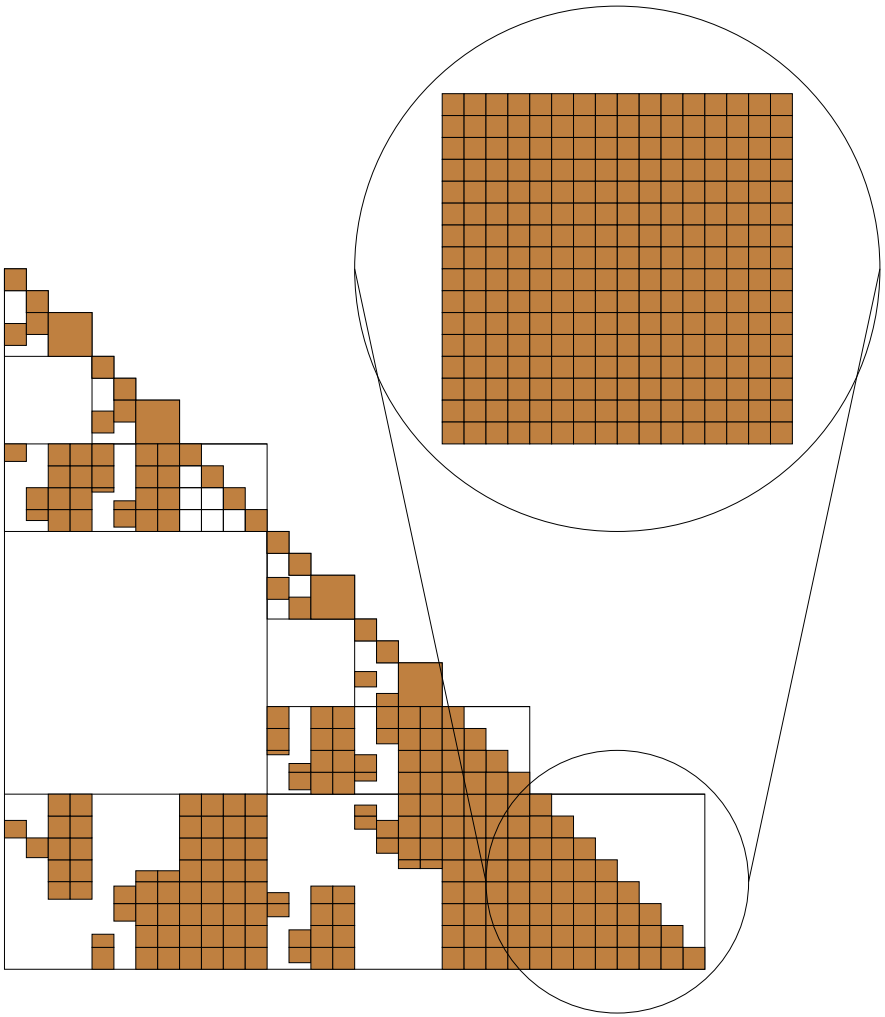


Figure: HODLR clustering

Block-admissibility	Partitioning		
	Flat	Hierarchical	
		Without nested bases	With nested bases
Weak	BLR	HODLR	HSS
Strong		$\mathcal{H}$	$\mathcal{H}^2$

Outline

BLR compression – Symbolic factorization



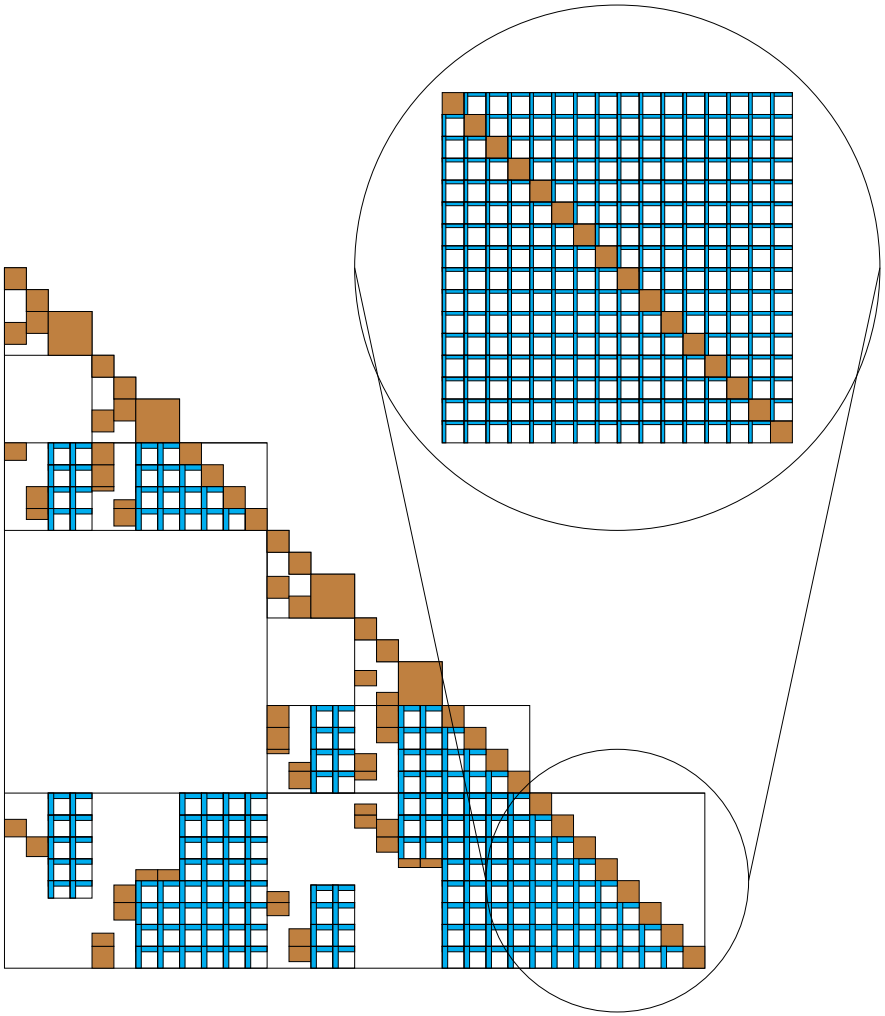
Approach

- Large supernodes are split
- It increases the level of parallelism

Operations

- Dense diagonal blocks
- TRSM are performed on dense off-diagonal blocks
- GEMM are performed between dense off-diagonal blocks

BLR compression – Symbolic factorization



Approach

- Large supernodes are split
- Large off-diagonal blocks are **low-rank**

Operations

- Dense diagonal blocks
- TRSM are performed on **low-rank** off-diagonal blocks
- GEMM are performed between **low-rank** off-diagonal blocks

When to compress ?

What do we have for now?

- Methods to compress dense blocks into low-rank form
- We potentially need to perform operations differently on low-rank blocks

Several strategies to choose **when** to compress

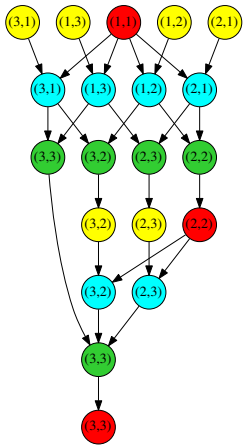
- During the allocation of the matrix
- When a block has received all its updates
- When a block was eliminated

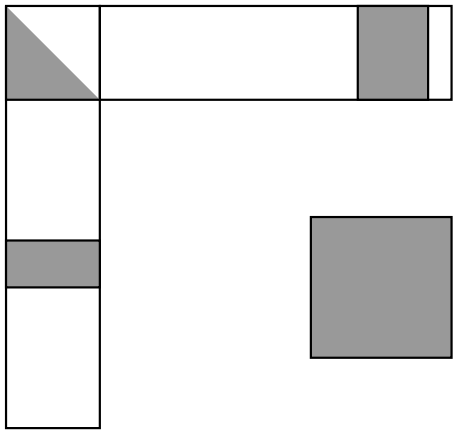
Strategy *Just-In-Time*

Compress *L*

1. Eliminate each column block
   1. Factorize the dense diagonal block
      Compress off-diagonal blocks belonging to the supernode
   2. Apply a TRSM on LR blocks (cheaper)
   3. LR update on dense matrices (*LR2GE* extend-add)
2. Solve triangular systems with low-rank blocks

	Compression
	GETRF (Facto)
	TRSM (Solve)
	LR2LR (Update)
	LR2GE (Update)



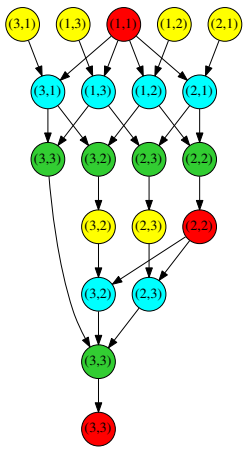


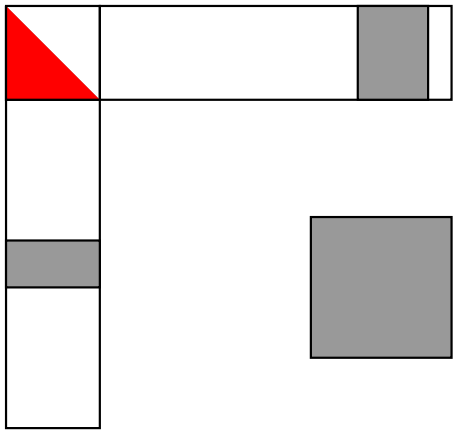
Strategy *Just-In-Time*

Compress *L*

1. Eliminate each column block
   1. Factorize the dense diagonal block
      Compress off-diagonal blocks belonging to the supernode
   2. Apply a TRSM on LR blocks (cheaper)
   3. LR update on dense matrices (*LR2GE* extend-add)
2. Solve triangular systems with low-rank blocks

Compression
GETRF (Facto)
TRSM (Solve)
LR2LR (Update)
LR2GE (Update)



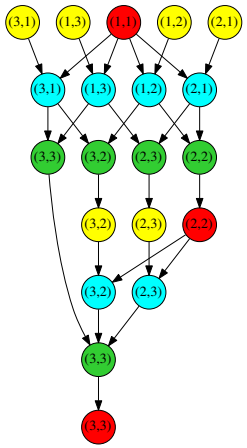


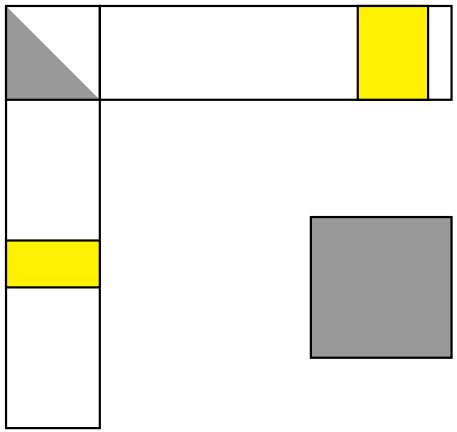
Strategy *Just-In-Time*

Compress *L*

1. Eliminate each column block
   1. Factorize the dense diagonal block
      Compress off-diagonal blocks belonging to the supernode
   2. Apply a TRSM on LR blocks (cheaper)
   3. LR update on dense matrices (*LR2GE* extend-add)
2. Solve triangular systems with low-rank blocks

	Compression
	GETRF (Facto)
	TRSM (Solve)
	LR2LR (Update)
	LR2GE (Update)



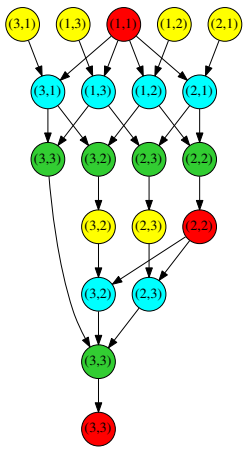


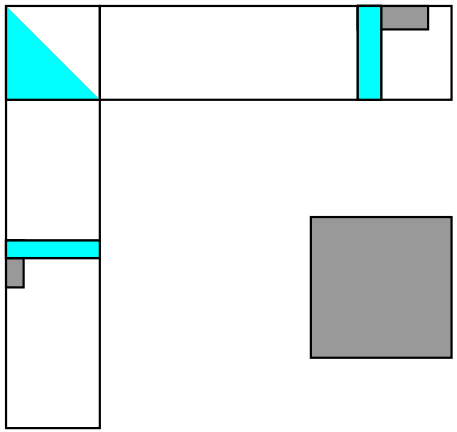
Strategy *Just-In-Time*

Compress *L*

1. Eliminate each column block
   1. Factorize the dense diagonal block
      Compress off-diagonal blocks belonging to the supernode
   2. Apply a TRSM on LR blocks (cheaper)
   3. LR update on dense matrices (*LR2GE* extend-add)
2. Solve triangular systems with low-rank blocks

Compression
GETRF (Facto)
TRSM (Solve)
LR2LR (Update)
LR2GE (Update)



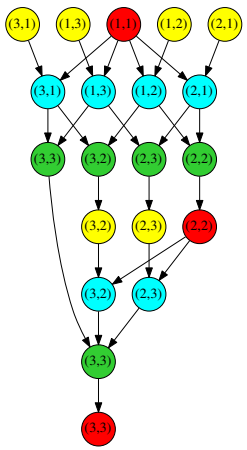


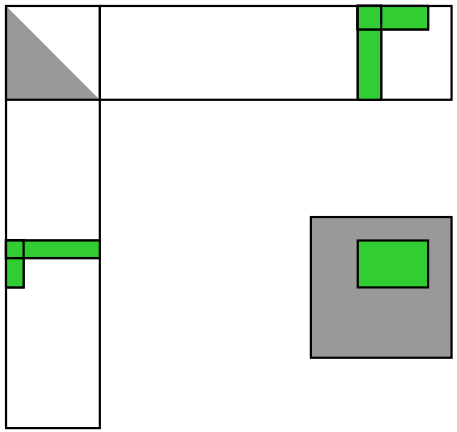
Strategy *Just-In-Time*

Compress L

1. Eliminate each column block
   1. Factorize the dense diagonal block
      Compress off-diagonal blocks belonging to the supernode
   2. Apply a TRSM on LR blocks (cheaper)
   3. LR update on dense matrices (*LR2GE* extend-add)
2. Solve triangular systems with low-rank blocks

Compression
GETRF (Facto)
TRSM (Solve)
LR2LR (Update)
LR2GE (Update)





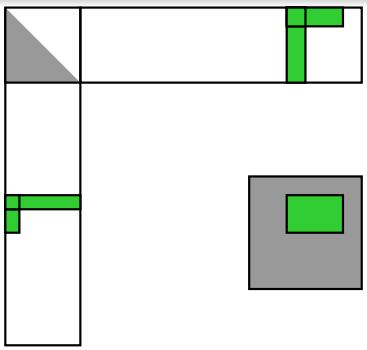
Summary of the *Just-In-Time* strategy

Advantages

- The expensive update operation, is faster using *LR2GE* kernel
- The formation of the dense update and its application is not expensive
- The size of the factors is reduced, as well as the solve cost

A limitation of this approach

- All blocks are allocated in full-rank before being compressed
- Limiting this constraint may reduce the level of parallelism



Just-In-Time

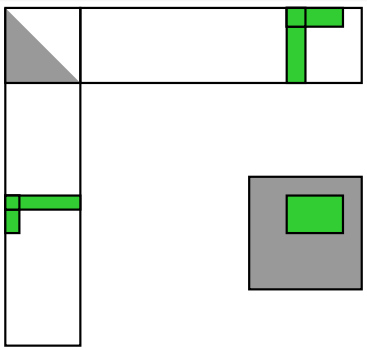
Summary of the *Just-In-Time* strategy

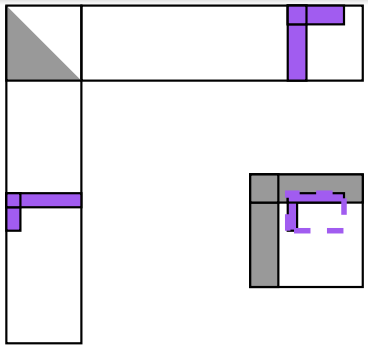
Advantages

- The expensive update operation, is faster using *LR2GE* kernel
- The formation of the dense update and its application is not expensive
- The size of the factors is reduced, as well as the solve cost

A limitation of this approach

- All blocks are allocated in full-rank before being compressed
- Limiting this constraint may reduce the level of parallelism



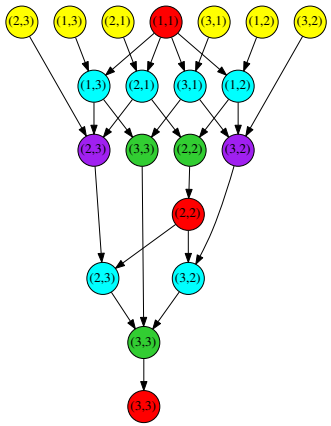


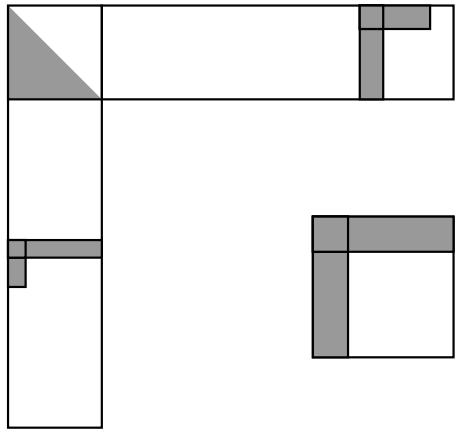
Strategy *Minimal Memory*

Compress A

1. Compress large off-diagonal blocks in A (exploiting sparsity)
2. Eliminate each column block
   1. Factorize the dense diagonal block
   2. Apply a TRSM on LR blocks (cheaper)
   3. LR update on LR matrices (*LR2LR* extend-add)
3. Solve triangular systems with LR blocks

Compression
GETRF (Facto)
TRSM (Solve)
LR2LR (Update)
LR2GE (Update)



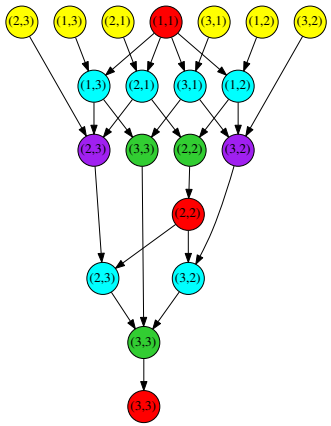


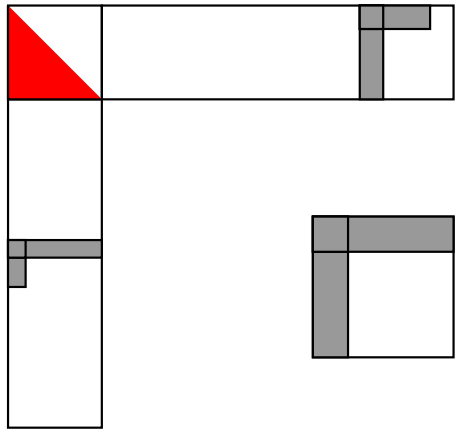
Strategy *Minimal Memory*

Compress A

1. Compress large off-diagonal blocks in A (exploiting sparsity)
2. Eliminate each column block
   1. Factorize the dense diagonal block
   2. Apply a TRSM on LR blocks (cheaper)
   3. LR update on LR matrices (*LR2LR* extend-add)
3. Solve triangular systems with LR blocks

Compression
GETRF (Facto)
TRSM (Solve)
LR2LR (Update)
LR2GE (Update)



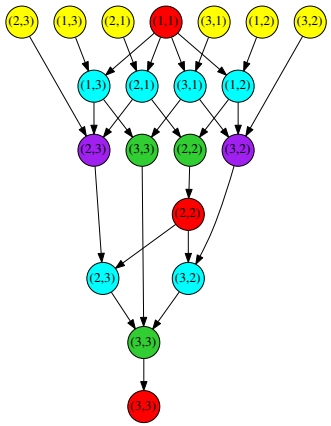


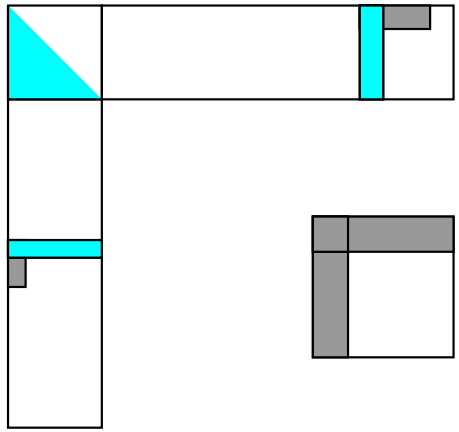
Strategy *Minimal Memory*

Compress A

1. Compress large off-diagonal blocks in A (exploiting sparsity)
2. Eliminate each column block
   1. Factorize the dense diagonal block
   2. Apply a TRSM on LR blocks (cheaper)
   3. LR update on LR matrices (*LR2LR* extend-add)
3. Solve triangular systems with LR blocks

Compression
GETRF (Facto)
TRSM (Solve)
LR2LR (Update)
LR2GE (Update)



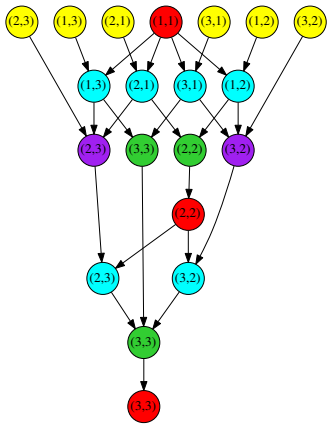


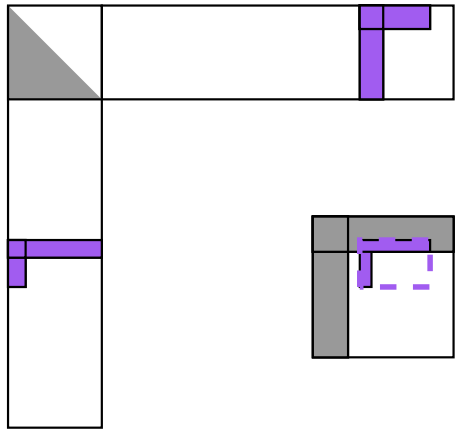
Strategy *Minimal Memory*

Compress A

1. Compress large off-diagonal blocks in A (exploiting sparsity)
2. Eliminate each column block
   1. Factorize the dense diagonal block
   2. Apply a TRSM on LR blocks (cheaper)
   3. LR update on LR matrices (*LR2LR* extend-add)
3. Solve triangular systems with LR blocks

	Compression
	GETRF (Facto)
	TRSM (Solve)
	LR2LR (Update)
	LR2GE (Update)





Solve operation

The solve operation for a generic lower triangular matrix L is applied to blocks in low-rank forms in our two scenarios.

1: Solve $A_{ik} = L_{ik} * U_{kk}$
2: Solve $A_{kj} = L_{kk} * U_{kj}$

Steps for (2) – similar for (1)

1. $L\hat{x} = \hat{b}$ becomes $LU_x V_x^t = U_b V_b^t$
2. Let us take $V_x^t = V_b^t$
3. We need to solve $LU_x = U_b$

The operation is then equivalent to applying a dense solve only to U_b, and the complexity is only $\Theta(m_L^2 r_x)$, instead of $\Theta(m_L^2 n_L)$ for the full-rank (dense) representation.

Extend-add process: $C = C - AB^t$

Product of two low-rank blocks with recompression

- $\hat{A}\hat{B}^t = (u_A(v_A^t v_B))u_B^t = u_A((v_A^t v_B)u_B^t)$
- Recompression
  1. $T = (v_A^t v_B)$
  2. $\hat{T} = \widehat{v_A^t v_B} = u_T v_T^t$
  3. $\hat{A}\hat{B}^t = (u_A u_T)(v_T^t v_B^t)$

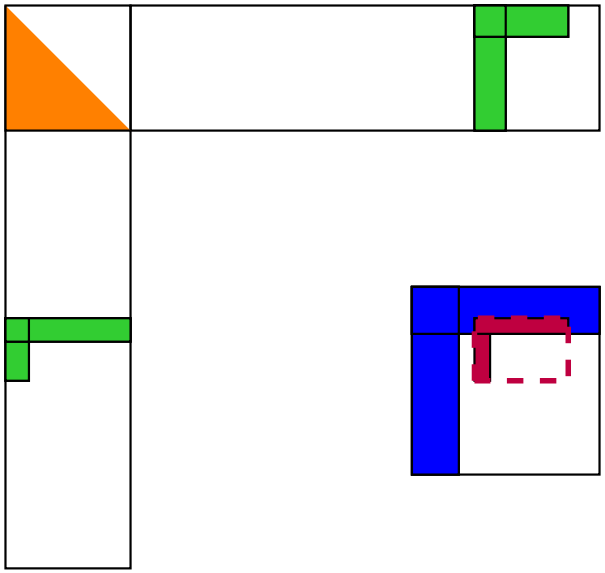
Application to a dense matrix (*LR2GE*)

Form explicitly the product

Application to a low-rank matrix (*LR2LR*)

- $u_{C'} v_{C'}^t = [u_C, u_{AB}]([v_C, -v_{AB}])^t$ (recompression ?)

Focus on the *LR2LR* kernel



LR2LR kernel using SVD

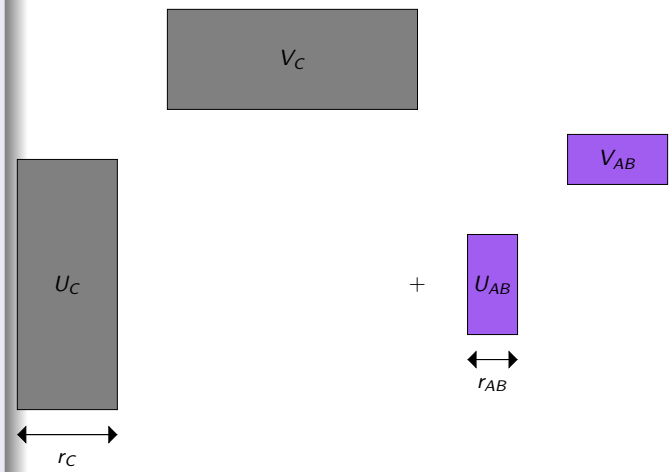
A low-rank structure $U_C V_C^t$ receives a low-rank contribution $U_{AB} V_{AB}^t$

Recompression algorithm

$$U_C V_C^t + U_{AB} V_{AB}^t = \left([U_C, U_{AB}]\right) \times \left([V_C, V_{AB}]\right)^t$$

- QR: $[U_C, U_{AB}] = Q_1 R_1$
- QR: $[V_C, V_{AB}] = Q_2 R_2$
- SVD: $R_1 R_2^t = u \sigma v^t$

$$A = \left(Q_1 u \sigma\right) \times \left(Q_2 v\right)^t$$



The complexity of this operation depends on the dimensions of C

LR2LR kernel using SVD

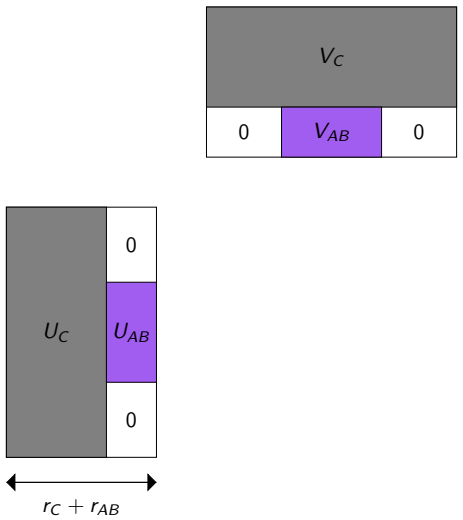
A low-rank structure $U_C V_C^t$ receives a low-rank contribution $U_{AB} V_{AB}^t$

Recompression algorithm

$$U_C V_C^t + U_{AB} V_{AB}^t = \left([U_C, U_{AB}]\right) \times \left([V_C, V_{AB}]\right)^t$$

- QR: $[U_C, U_{AB}] = Q_1 R_1$
- QR: $[V_C, V_{AB}] = Q_2 R_2$
- SVD: $R_1 R_2^t = u \sigma v^t$

$$A = \left(Q_1 u \sigma\right) \times \left(Q_2 v\right)^t$$



The complexity of this operation depends on the dimensions of C

LR2LR kernel using SVD

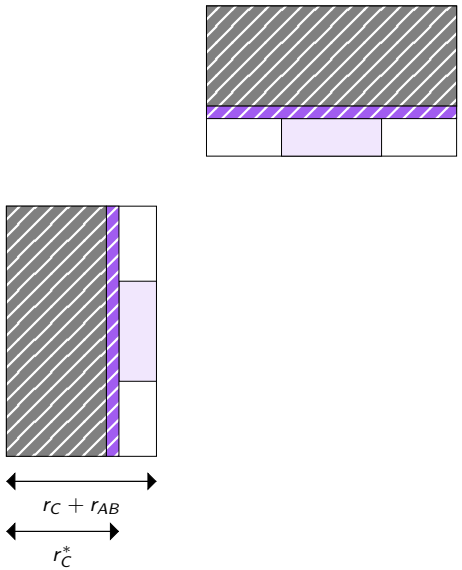
A low-rank structure $U_C V_C^t$ receives a low-rank contribution $U_{AB} V_{AB}^t$

Recompression algorithm

$$U_C V_C^t + U_{AB} V_{AB}^t = \left([U_C, U_{AB}]\right) \times \left([V_C, V_{AB}]\right)^t$$

- QR: $[U_C, U_{AB}] = Q_1 R_1$
- QR: $[V_C, V_{AB}] = Q_2 R_2$
- SVD: $R_1 R_2^t = u \sigma v^t$

$$A = \left(Q_1 u \sigma\right) \times \left(Q_2 v\right)^t$$



The complexity of this operation depends on the dimensions of C

LR2LR kernel using RRQR

Taking advantage of orthogonality

- If we handle low-rank matrices of the form uv^t, we can ensure that u matrices are always orthogonal
- This is true after the first compression (for SVD, apply singular values on the right)
- This is conserved by the **Solve** and the **Update** operations
- Warning: we have to store U^t in the LU factorization to ensure orthogonality

Maintaining orthogonality by enlarging an existing basis

- QR or partialQR
- Modified Gram-Schmidt

Extend-add: RRQR Recompression

A low-rank structure $u_1 v_1^t$ receives a low-rank contribution $u_2 v_2^t$.
u_1 and u_2 are orthogonal matrices

Algorithm

$$A = u_1 v_1^t + u_2 v_2^t = \left([u_1, u_2]\right) \times \left([v_1, v_2]\right)^t$$

Orthogonalize u_2 with respect to u_1 :

$$u_2^* = u_2 - u_1(u_1^t u_2)$$

Form new orthogonal basis, and normalize each column :

$$[u_1, u_2] = [u_1, u_2^*] \times \begin{pmatrix} I & u_1^t u_2 \\ 0 & I \end{pmatrix}$$

Apply a RRQR on :

$$\begin{pmatrix} I & u_1^t u_2 \\ 0 & I \end{pmatrix} \times \left([v_1, v_2]\right)^t$$

Experimental setup

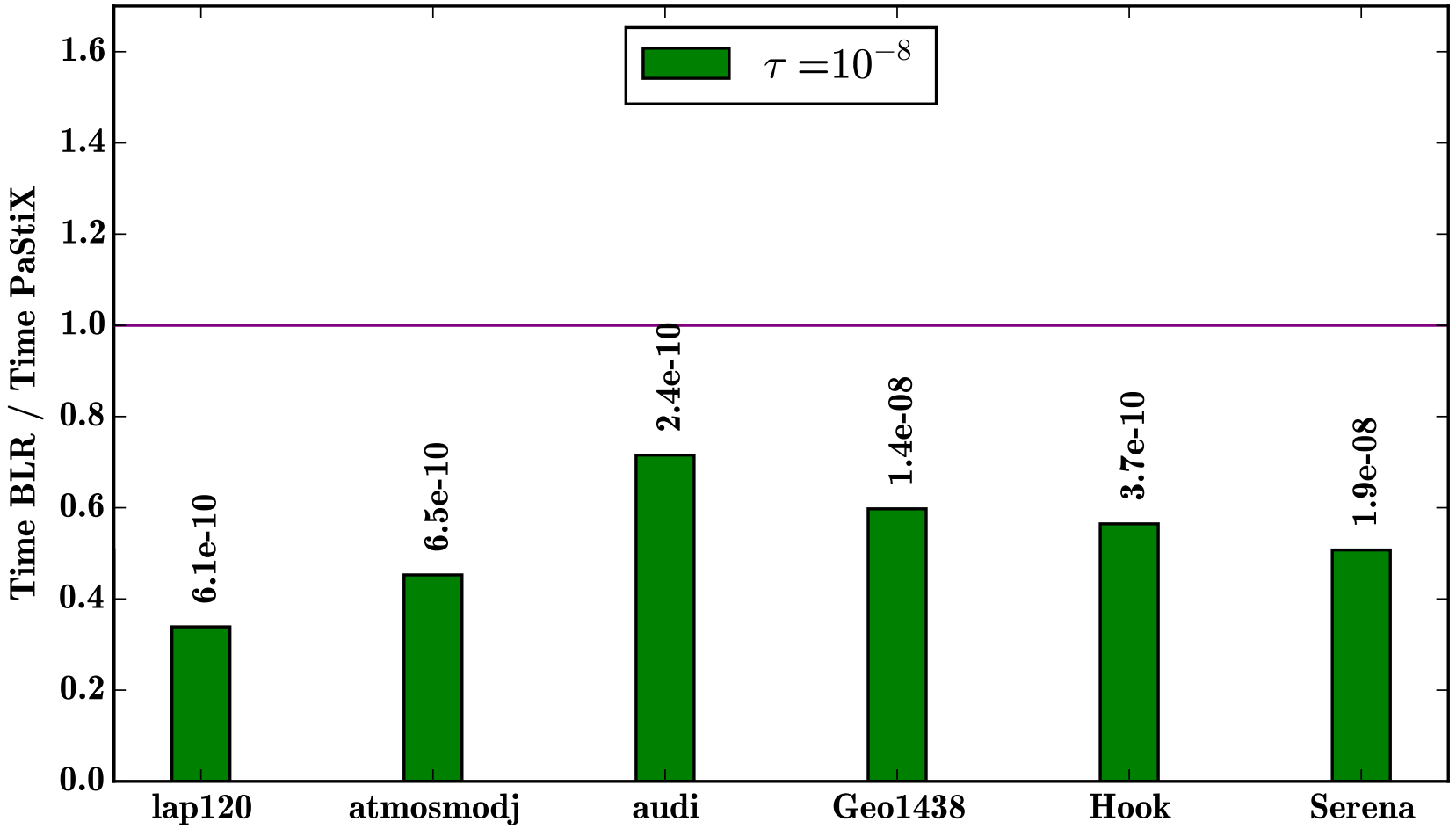
Machine: 2 Intel Xeon $E5-2680v3$ at 2.50 GHz

- 128 GB
- 24 threads
- Parallelism is obtained following PaStiX static scheduling for multi-threaded architectures

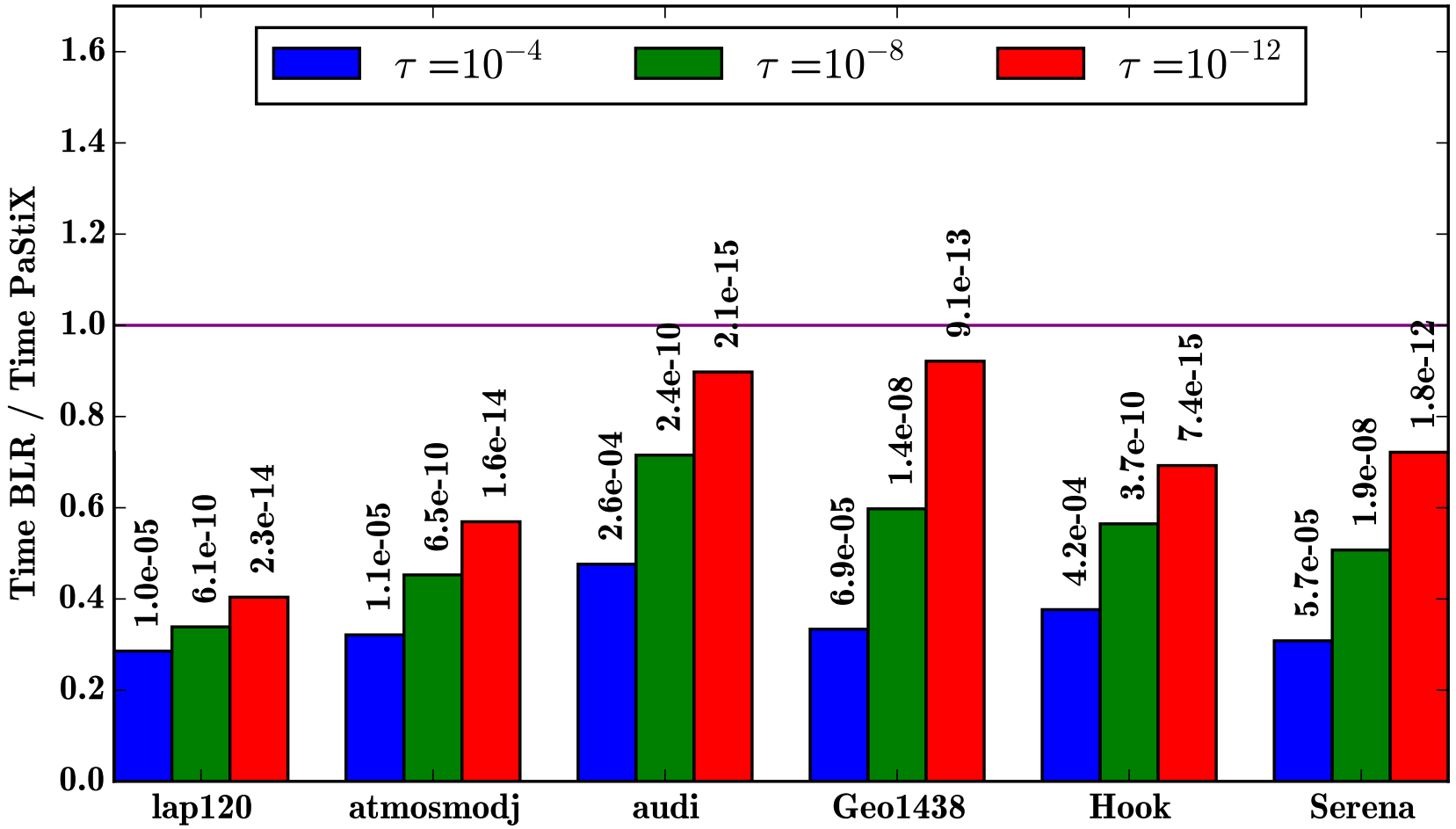
Entry parameters

- Tolerance τ: absolute parameter (normalized for each block)
- Compression method is RRQR
- Blocking sizes: between 128 and 256 in following experiments

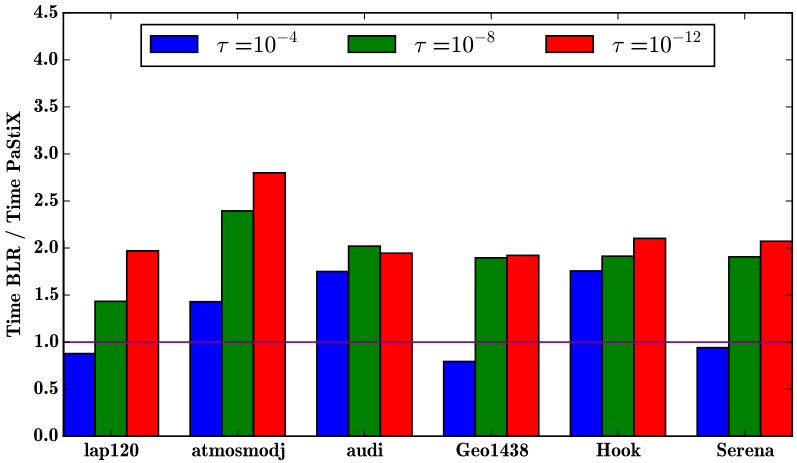
Performance of RRQR/*Just-In-Time* wrt full-rank version



Performance of RRQR/*Just-In-Time* wrt full-rank version

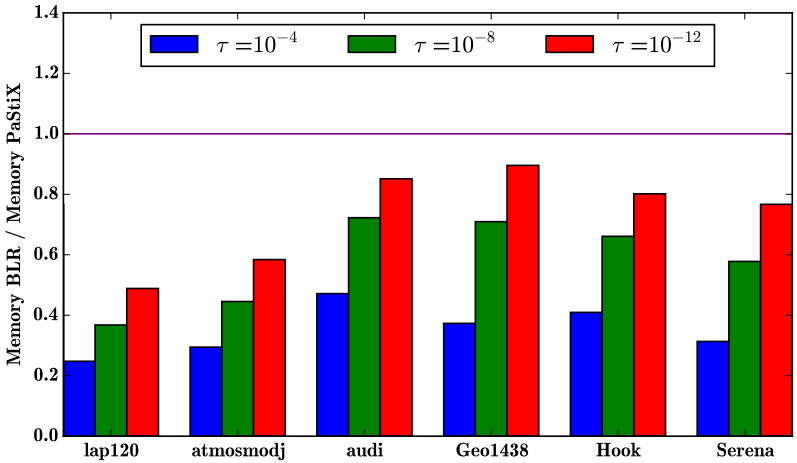


Behavior of RRQR/*Minimal Memory* wrt full-rank version



Performance

- Increase by a factor of 1.9 for $\tau = 10^{-8}$
- Better for a lower accuracy



Memory peak

- Reduction by a factor of 1.7 for $\tau = 10^{-8}$
- Close to the results obtained using SVD

Summary

A $330^3 = 36M$ unknowns Laplacian has been solved with $\tau = 10^{-4}$ while it was restricted to $220^3 = 8M$ using the full-rank version

Memory consumption

- *Minimal Memory* strategy really saves memory
- *Just-In-Time* strategy reduces the size of *L*' factors, but supernodes are allocated dense at the beginning: no gain in pure *right-looking*

Factorization time

- *Minimal Memory* strategy requires expensive extend-add algorithms to update (recompress) low-rank structures with the *LR2LR* kernel
- *Just-In-Time* strategy continues to apply dense update at a smaller cost through the *LR2GE* kernel

Outline

What is asked ?

Compute the dense Block Low-Rank complexity (memory and time) of an $n \times n$ matrix when using updates on dense matrices (Just-In-Time).

Assumptions

- The rank is in $\Theta(n^{\alpha})$
- The block size is n^{x}, there are n^{1-x} blocks
- Diagonal blocks are dense
- There are $\Theta(1)$ full-rank blocks in each column

Objectives: 1) compute the complexity (memory and time) depending on the rank and the block size and 2) compute the optimal block size. Apply with $r = \Theta(1)$ and $r = \Theta(n^{\frac{1}{2}})$.