# Low-Rank Compression in Sparse direct solvers

Grégoire Pichon

CNRS, INRIA, Université Lyon 1 & ENS Lyon

October, 10th 2023

# Context

## Sparse direct solvers

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

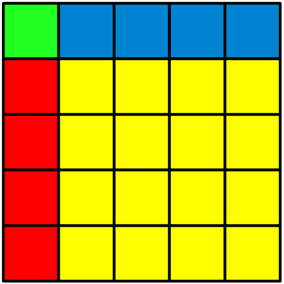
## Low-rank compression

- Represent data in low-rank form
- Reduce storage and operations cost
- Reduce the quality of the representation

# Block LU

**Algorithm 1** LU Factorization

**for** $k = 1$ to $n$ **do**
    Factorize $A_{kk} = L_{kk} U_{kk}$
    **for** $i = k + 1$ to $n$ **do**
        Solve $A_{ik} = L_{ik} * U_{kk}$
    **for** $j = k + 1$ to $n$ **do**
        Solve $A_{kj} = L_{kk} * U_{kj}$
    **for** $i = k + 1$ to $n$ **do**
        **for** $j = k + 1$ to $n$ **do**
            $A_{ij} = A_{ij} - L_{ik} * U_{kj}$



# Outline

1. Sparse direct solvers
   - Structure: fill-in, dependencies
   - Complexity
2. Low-rank compression
3. Low-rank into sparse direct solvers
   - General approach
   - PaStiX strategies

# Outline

1. Sparse direct solvers
   - Structure: fill-in, dependencies
   - Complexity
2. Low-rank compression
3. Low-rank into sparse direct solvers
   - General approach
   - PaStiX strategies

# Sparse matrices

## Where they come from

- Many applications
- Discretization of PDEs
- Finite Element Method on 2D/3D graphs

## Possible factorization

- $A = LU$ if $A$ is general
- $A = LDL^t$ if $A$ is symmetric
- $A = LL^t$ if $A$ is symmetric positive definite

In this class, we will only consider matrices that are at least symmetric in structure: $a_{i,j} \neq 0 \Rightarrow a_{j,i} \neq 0$.
If not, we will work on the structure of $A + A^t$.

# Fill-in problem with $A$ symmetric

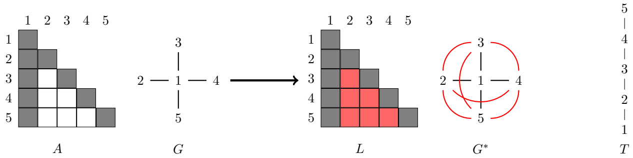



Figure: Natural ordering

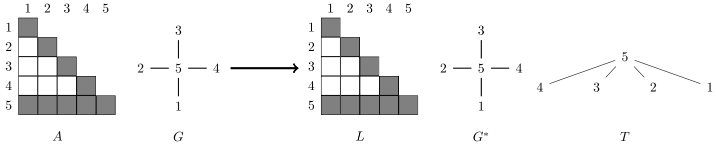
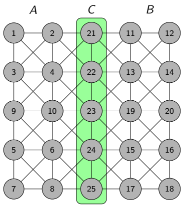



Figure: Optimal ordering

# Ordering with Nested Dissection

## Partition $V = A \cup B \cup C$

1. Order $C$ with larger indices: $V_A < V_C$ and $V_B < V_C$
2. Apply the process recursively on $A$ and $B$
3. Apply local heuristic such as AMF on small subgraphs



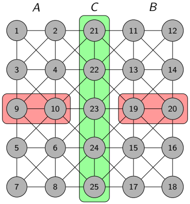

## Nested dissection performed by an external partitioner tool

- Find a separator $C$ as small as possible
- Balance subparts $A$ and $B$

# Ordering with Nested Dissection

## Partition $V = A \cup B \cup C$

1. Order $C$ with larger indices: $V_A < V_C$ and $V_B < V_C$
2. Apply the process recursively on $A$ and $B$
3. Apply local heuristic such as AMF on small subgraphs



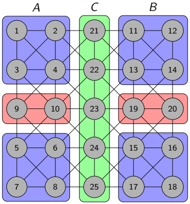

## Nested dissection performed by an external partitioner tool

- Find a separator $C$ as small as possible
- Balance subparts $A$ and $B$

# Ordering with Nested Dissection

## Partition $V = A \cup B \cup C$

1. Order $C$ with larger indices: $V_A < V_C$ and $V_B < V_C$
2. Apply the process recursively on $A$ and $B$
3. Apply local heuristic such as AMF on small subgraphs




## Nested dissection performed by an external partitioner tool

- Find a separator $C$ as small as possible
- Balance subparts $A$ and $B$

# Building an efficient solver

## Efficiency

- Sparse structure, small blocks
- The underlying hardware requires smart memory management
- We would like to know the final structure before starting the factorization to better balance memory and computations

## Parallelism

- Express high level of parallelism
- Avoid bottlenecks
- Still keeping a sufficient granularity

# Symbolic factorization – naive algorithm

**Theorem**

*(Characterization Theorem) Given an $n \times n$ sparse matrix $A$, and its adjacency graph $G = (V, E)$, any entry $a_{i,j} = 0$ from $A$ will become a non-null entry in the factorized matrix if and only if there is a path in $G$ from vertex $i$ to vertex $j$ that only goes through vertices with a lower index than $i$ and $j$.*

**Algorithm 2** Naive algorithm, as expensive as the numerical factorization

```
for k = 1 to n do
    Build Col(A_{*,k}) the set of vertices adjacent to V_k
    for i in Col(A_{*,k}) do
        for j in Col(A_{*,k}) do
            If a_{i,j} = 0 fill-in entry (when forming the clique)
```

# Symbolic factorization – linear algorithm

## Objects

- $Col(A_{*,k})$ is the set of vertices adjacent to $V_k$
- $SCol(A_{*,k})$ is the sorted version of $Col(A_{*,k})$

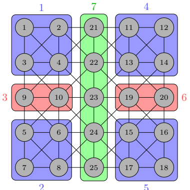
**Algorithm 3** Linear algorithm: optimal

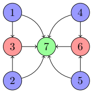
1: **for** $k = 1$ to $n$ **do**
2:     Build $SCol(A_{*,k})$ the sorted set of vertices adjacent to $V_k$
3: **for** $k = 1$ to $n$ **do**
4:     Select *first*, the first element of $SCol(A_{*,k})$
5:     $SCol(A_{*,first}) = SCol(A_{*,first}) \cup SCol(A_{*,k})$

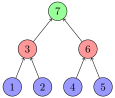
# Block Symbolic Factorization

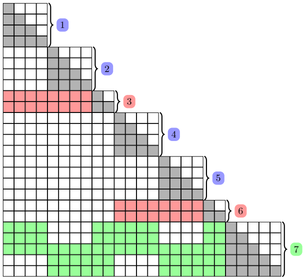
## General approach

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

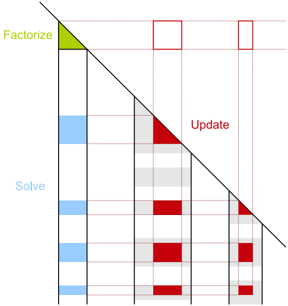

Elimination tree (T).


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)



# Outline

# Context of the complexity study

## Conditions

- For bounded-density graphs [Miller, Vavasis - 1991]
- $\bar{G} = (\bar{V}, \bar{E})$ with $|\bar{V}| = p$
- Partition into $\bar{V} = A \cup B \cup C$

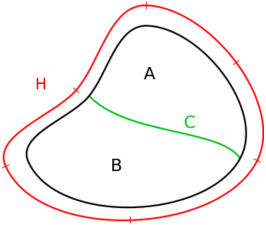


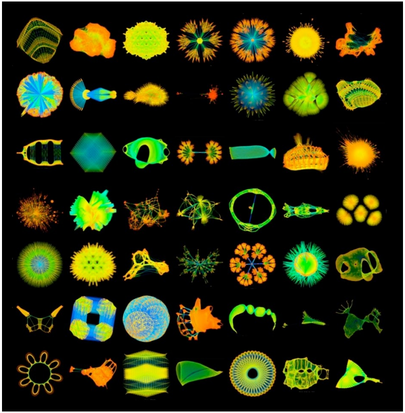

Figure: Partitioning a graph

## $p^\sigma$-Separation Theorem [Lipton, Tarjan - 1979] when using nested dissection

- $0 < \alpha < 1$, $\beta > 0$, $\frac{1}{2} \leq \sigma < 1$
- $|A| \leq \alpha p$, $|B| \leq \alpha p$
- $|C| \leq \beta p^\sigma$

# Context of the Study

The University of Florida Sparse Matrix Collection



# Computation

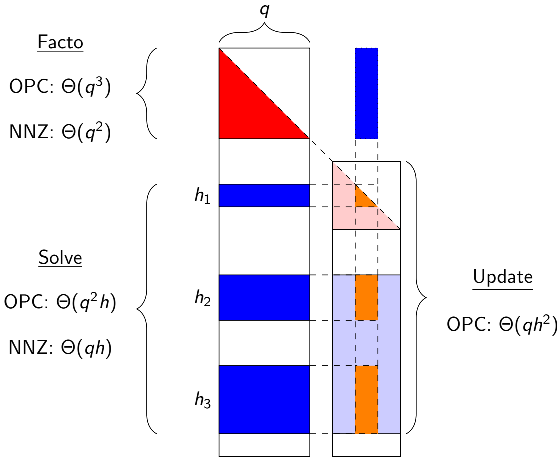
## Properties

- $|C| = q \leq \beta p^\sigma$
- $|H| = h = \Theta(q)$: halo size for graphs with a good aspect ratio

## Operations

- Storage: $\Theta(q^2)$ for a $q$-by-$q$ block
- Factorization (GETRF): $\Theta(q^3)$ for a $q$-by-$q$ block
- Solve (TRSM): $\Theta(q^2 h)$ when solving $h$ unknowns with the previous factorization
- Update (GEMM): $\Theta(q h_1 h_2)$ when updating (any block) with the product of a $h_1 \times q$ block with a $h_2 \times q$ block

# Complexity on a Block-Column




$$\sum_i h_i = h$$

Data exchanges: $\Theta(h^2)$

# Complexity on a Block-Column

## In practice

- For a $d$-dim bounded-density graph, $\sigma = \frac{d-1}{d}$

$$compl(\bar{G}, H) \leq contrib_C + compl(A, H_A) + compl(B, H_B)$$

## Overall complexity depending on $contrib_C = \Theta(p^y)$ [Lipton, Rose, Tarjan - 1977]

- $y < 1 \rightarrow \Theta(n)$
- $y = 1 \rightarrow \Theta(n\ ln(n))$
- $y > 1 \rightarrow \Theta(n^y)$

# Results

For the Factorization on general 2D/3D finite element meshes

## Previous results

- OPC: $\Theta(p^{3\sigma})$
- NNZ: $\Theta(p^{2\sigma})$

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

# Outline

# Objectives

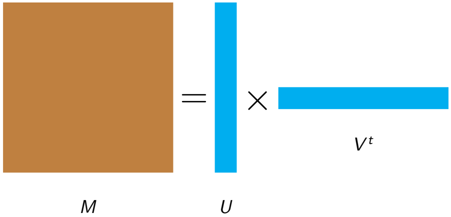
## Reduce the complexity

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

## Similar properties

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

# 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, $n = 500$

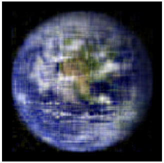


Figure: $r = 10$, 4% of original storage

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 $k_\epsilon$ of a matrix $A$ at accuracy $\epsilon$ is defined as the smallest integer such that there exists a matrix $A_\epsilon$ of rank $k_\epsilon$ 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 $||A - \hat{A}||_2 = \sigma_{k+1}$

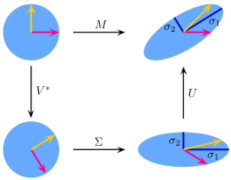
## Low-rank matrix

$A$ is said to be low-rank (for a given accuracy $\epsilon$) if its numerical rank $k_\epsilon$ is small enough such that its rank-$k_\epsilon$ 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 4** 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

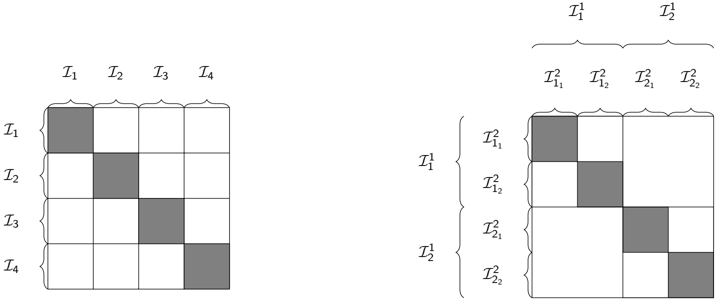



Figure: BLR clustering

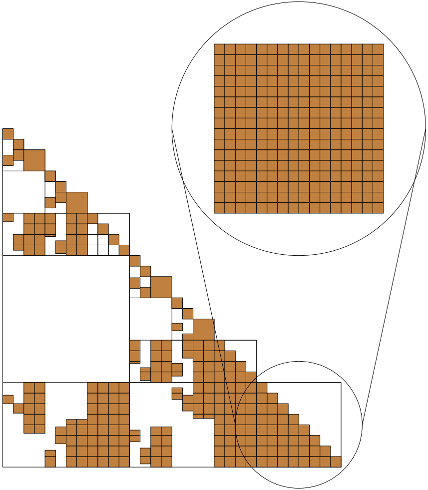
Figure: HODLR clustering

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

# Outline

# Outline

# BLR compression – Symbolic factorization



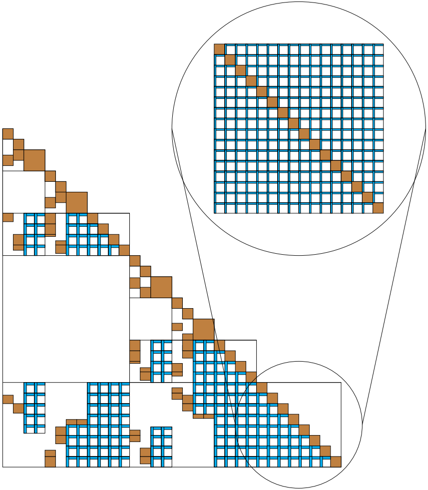
## Approach

- 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

# Outline

# 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

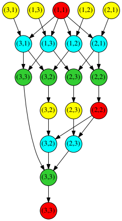
- During the allocation of the matrix
- When a block has received all its updates
- When a block was eliminated

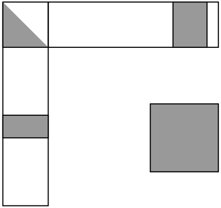
# Strategy *Just-In-Time*

## 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

| | |
|---|---|
| Yellow | *Compression* |
| Red | *GETRF* (Facto) |
| Cyan | *TRSM* (Solve) |
| Purple | *LR2LR* (Update) |
| Green | *LR2GE* (Update) |



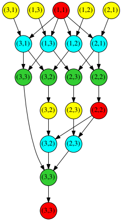


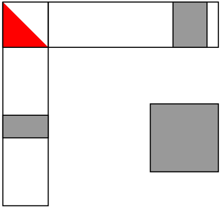
# Strategy *Just-In-Time*

## 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

| | |
|---|---|
| (yellow) | *Compression* |
| (red) | *GETRF* (Facto) |
| (cyan) | *TRSM* (Solve) |
| (purple) | *LR2LR* (Update) |
| (green) | *LR2GE* (Update) |



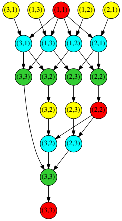


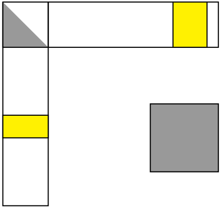
# Strategy *Just-In-Time*

## 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) |



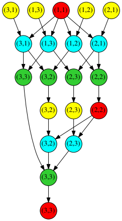



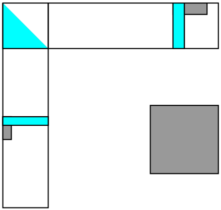
# Strategy *Just-In-Time*

## 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

| | |
|---|---|
| (yellow) | *Compression* |
| (red) | *GETRF* (Facto) |
| (cyan) | *TRSM* (Solve) |
| (purple) | *LR2LR* (Update) |
| (green) | *LR2GE* (Update) |



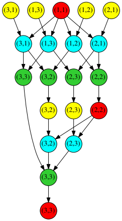


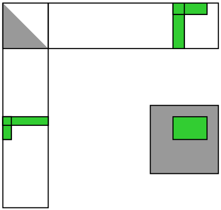
# Strategy *Just-In-Time*

## 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) |



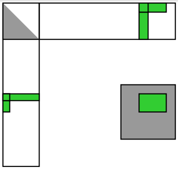
# Summary of the *Just-In-Time* strategy

## 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*

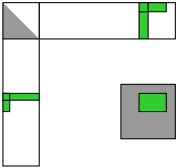
# Summary of the *Just-In-Time* strategy

## Advantages

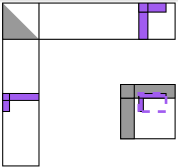
- The expensive update operation, is faster using *LR2GE* kernel
- The formation of the dense update and its application is not expensive
- The size of the factors is reduced, as well as the solve cost

## A limitation of this approach

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



*Just-In-Time*



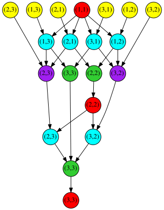
*Minimal Memory*

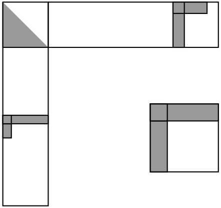
# Strategy *Minimal Memory*

## 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

| Colour | Operation |
|---|---|
| Yellow | *Compression* |
| Red | *GETRF* (Facto) |
| Cyan | *TRSM* (Solve) |
| Purple | *LR2LR* (Update) |
| Green | *LR2GE* (Update) |



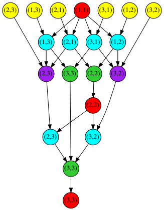


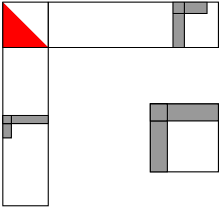
# Strategy *Minimal Memory*

## 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) |



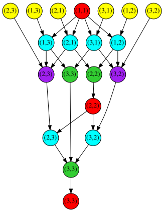


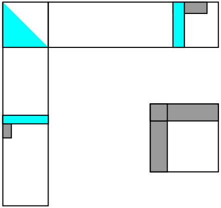
# Strategy *Minimal Memory*

## 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) |



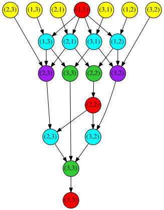


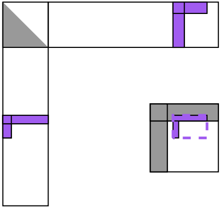
# Strategy *Minimal Memory*

## 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

| Colour | Operation |
|---|---|
| Yellow | *Compression* |
| Red | *GETRF* (Facto) |
| Cyan | *TRSM* (Solve) |
| Purple | *LR2LR* (Update) |
| Green | *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 $L U_x V_x^t = U_b V_b^t$
2. Let us take $V_x^t = V_b^t$
3. We need to solve $L U_x = U_b$

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. $\hat{A}\hat{B}^t = (u_A u_T)(v_T^t v_B^t)$

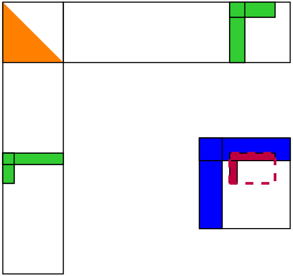
## Application to a dense matrix (*LR2GE*)

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

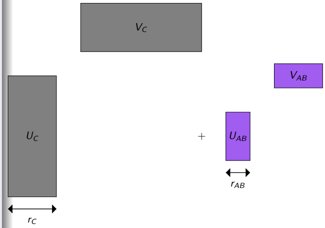
A low-rank structure $U_C V_C^t$ receives a low-rank contribution $U_{AB} V_{AB}^t$

## 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

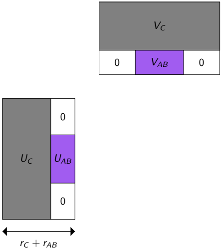
A low-rank structure $U_C V_C^t$ receives a low-rank contribution $U_{AB} V_{AB}^t$

## Recompression algorithm

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

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

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




The complexity of this operation depends on the dimensions of $C$

# *LR2LR* kernel using SVD

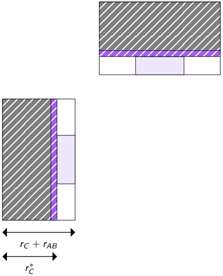
A low-rank structure $U_C V_C^t$ receives a low-rank contribution $U_{AB} V_{AB}^t$

## Recompression algorithm

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

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

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




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

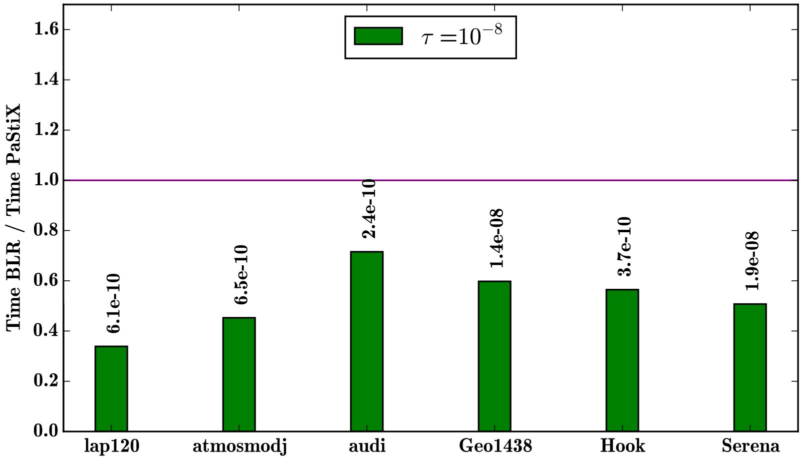
## Machine: 2 Intel Xeon $E5-2680v3$ at 2.50 GHz

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

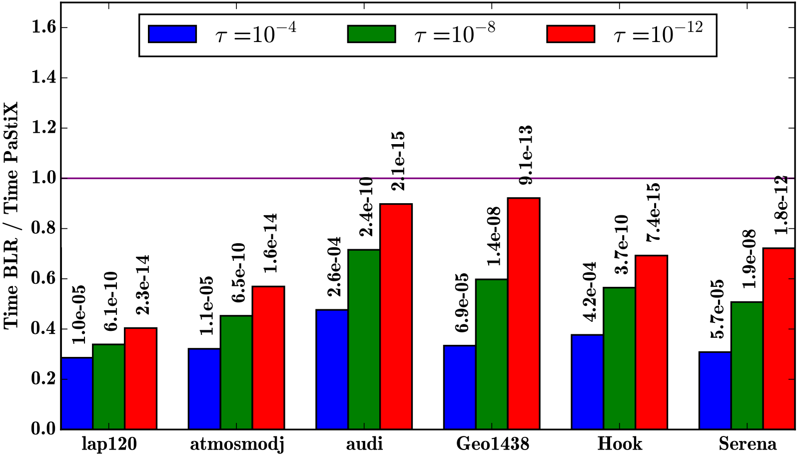
## Entry parameters

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

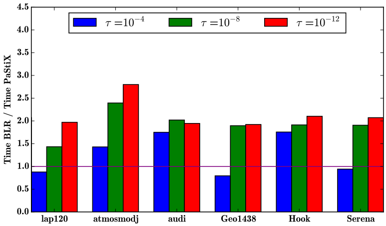
# Performance of RRQR/*Just-In-Time* wrt full-rank version



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

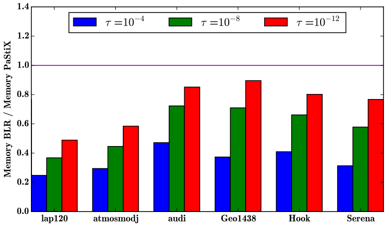


# Behavior of RRQR/*Minimal Memory* wrt full-rank version




## 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