



# Sparse Supernodal Solver exploiting Low-Rankness Property

September 6th, 2017 - Sparse Days

Grégoire Pichon<sup>a</sup>, Eric Darve<sup>b</sup>, Mathieu Faverge<sup>a</sup>, Pierre Ramet<sup>a</sup>, Jean Roman<sup>a</sup>

---

<sup>a</sup>Inria, Bordeaux INP, CNRS, University of Bordeaux

<sup>b</sup>Stanford University

## Introduction

### Current sparse direct solver for 3D problems

- $\Theta(n^2)$  time complexity
- $\Theta(n^{\frac{4}{3}})$  memory complexity
- BLAS 3 operations

### Block Low-Rank solver

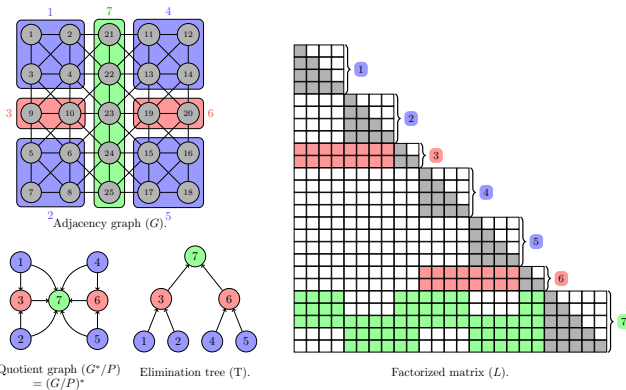
- Large blocks are compressed into a low-rank form
- Operations keep untouched the global behaviour of PASTIX
- *Minimal Memory* strategy allows to save memory
- *Just-In-Time* strategy allows to reduce time-to-solution

Objective: build an algebraic low-rank solver following the supernodal approach of PASTIX, with block-data structures

# Symbolic Factorization

## General approach

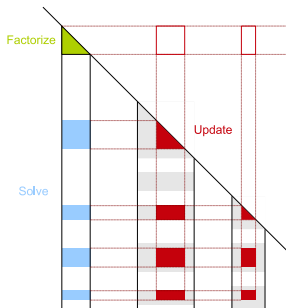
1. Build a partition with the nested dissection process
2. Compress information on data blocks
3. Compute the block elimination tree using the block quotient graph



# Numerical Factorization

Algorithm to eliminate the column block  $k$

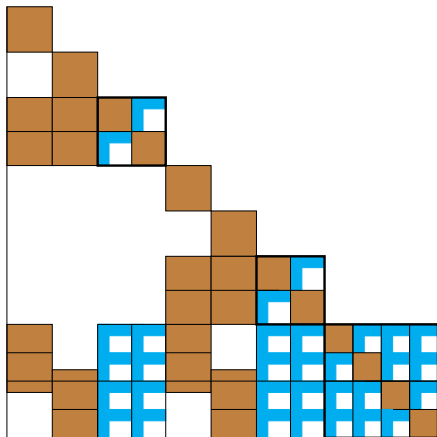
1. **Factorize** the diagonal block (POTRF/GETRF)
2. **Solve** off-diagonal blocks in the current column (TRSM)
3. **Update** the trailing matrix with the column's contribution (GEMM)



## Parallelism

- Right-Looking
- Left-Looking

## Block-Low-Rank Compression – Symbolic Factorization



Large off-diagonal are low-rank, in the form  $uv^t$

## Some Related Works

### Block Low-Rank solvers

- BLR-MUMPS introduced an approach focused on reducing the time to solution which is close to our *Just-In-Time* strategy
- LSTC developed compression/recompression techniques in a dense context

### Other approaches for sparse

- Strumpack HSS by Ghysels et al. utilizes randomized sampling to perform an efficient extend-add
- HODLR by Darve et al. uses pre-selection of rows and columns (BDLR)
- $\mathcal{H}$ -LU by Hackbusch et al. does not fully exploit the symbolic factorization

# Block-Low-Rank Algorithm

## Approach

- Large supernodes are partitioned into a set of smaller supernodes
- Large off-diagonal blocks are represented as low-rank blocks

## Operations

- Diagonal blocks are dense
- TRSM are performed on low-rank off-diagonal blocks
- GEMM are performed between low-rank off-diagonal blocks. It creates contributions to dense or low-rank blocks: this is the extend-add problem

## Compression techniques

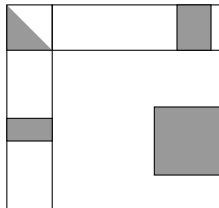
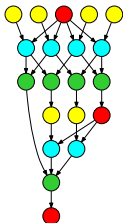
- SVD, RRQR for now
- Possible extension to any algebraic method: ACA, randomized techniques...

## Strategy *Just-In-Time*

### Compress $L$

1. Eliminate each column block
  - 1.1 Factorize the dense diagonal block  
Compress off-diagonal blocks belonging to the supernode
  - 1.2 Apply a TRSM on LR blocks (cheaper)
  - 1.3 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)








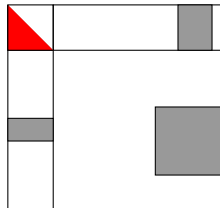
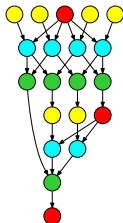


## Strategy *Just-In-Time*

### Compress $L$

1. Eliminate each column block
  - 1.1 Factorize the dense diagonal block  
Compress off-diagonal blocks belonging to the supernode
  - 1.2 Apply a TRSM on LR blocks (cheaper)
  - 1.3 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)

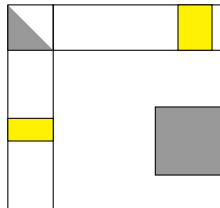
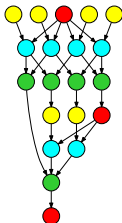


## Strategy *Just-In-Time*

### Compress $L$

1. Eliminate each column block
  - 1.1 Factorize the dense diagonal block  
Compress off-diagonal blocks belonging to the supernode
  - 1.2 Apply a TRSM on LR blocks (cheaper)
  - 1.3 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)

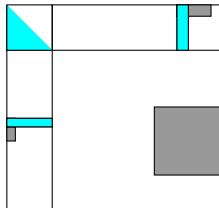
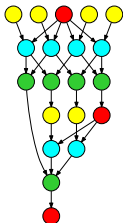


## Strategy *Just-In-Time*

### Compress $L$

1. Eliminate each column block
  - 1.1 Factorize the dense diagonal block  
Compress off-diagonal blocks belonging to the supernode
  - 1.2 Apply a TRSM on LR blocks (cheaper)
  - 1.3 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)

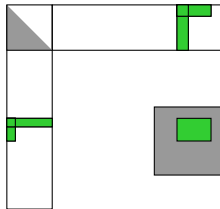
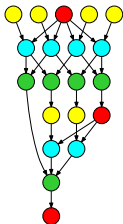


## Strategy *Just-In-Time*

### Compress $L$

1. Eliminate each column block
  - 1.1 Factorize the dense diagonal block  
Compress off-diagonal blocks belonging to the supernode
  - 1.2 Apply a TRSM on LR blocks (cheaper)
  - 1.3 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)

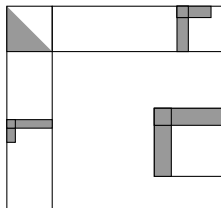
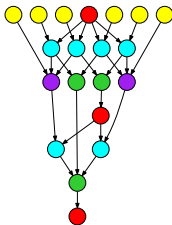


## Strategy *Minimal Memory*

### Compress $A$

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

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

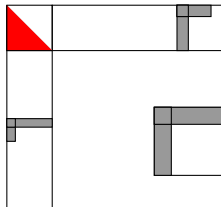
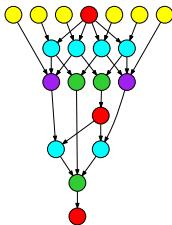


## Strategy *Minimal Memory*

### Compress $A$

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

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

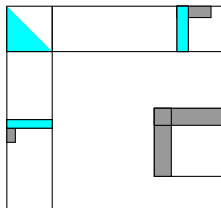
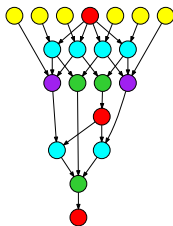


## Strategy *Minimal Memory*

### Compress $A$

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

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

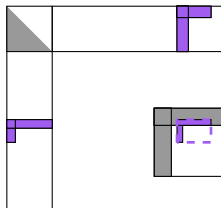
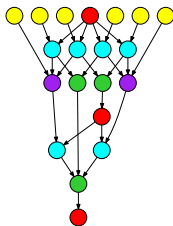


## Strategy *Minimal Memory*

### Compress $A$

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

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





## Comparison of both strategies

### Memory consumption

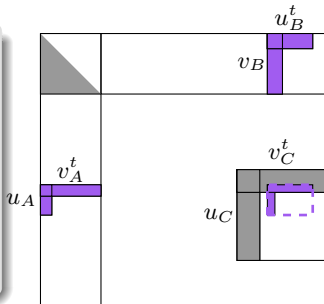
- *Minimal Memory* strategy really saves memory
- *Just-In-Time* strategy reduces the size of  $L'$  factors, but supernodes are allocated dense at the beginning: no gain in pure *right-looking*

### Low-Rank extend-add

- *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

## Focus on the *LR2LR* kernel

- Update of  $C$  with contribution from blocs  $A$  and  $B$
- The low-rank matrix  $u_{AB}v_{AB}^t$  is added to  $u_Cv_C^t$
- $u_{AB}v_{AB}^t = (u_A(v_A^t v_B))u_B^t$  or  $u_{AB}v_{AB}^t = u_A((v_A^t v_B)u_B^t)$
- Eventually, recompression of  $(v_A^t v_B)$



## LR2LR kernel using SVD

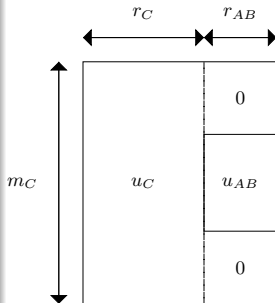
A low-rank structure  $u_C v_C^t$  receives a low-rank contribution  $u_{AB} v_{AB}^t$ .

### Algorithm

$$A = u_C v_C^t + u_{AB} v_{AB}^t = ([u_C, u_{AB}]) \times ([v_C, v_{AB}])^t$$

- QR:  $[u_C, u_{AB}] = Q_1 R_1 \quad \Theta(m(r_C + r_{AB})^2)$
- QR:  $[v_C, v_{AB}] = Q_2 R_2 \quad \Theta(n(r_C + r_{AB})^2)$
- SVD:  $R_1 R_2^t = u \sigma v^T \quad \Theta((r_C + r_{AB})^3)$

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



Optimizations using RRQR to reduce complexity to  $\Theta(n(r_C + r_{AB})r_C^*)$

## Experimental setup

Machine: 2 INTEL Xeon E5 – 2680v3 at 2.50 GHz

- 128 GB
- 24 threads

### 3D Matrices from The SuiteSparse Matrix Collection

- *Audi*: structural problem (943 695 dofs)
- *Atmosmodj*: atmospheric model (1 270 432 dofs)
- *Geo1438*: geomechanical model of earth (1 437 960 dofs)
- *Hook*: model of a steel hook (1 498 023 dofs)
- *Serena*: gas reservoir simulation (1 391 349 dofs)
- + Laplacian: Poisson problem (7-points stencil)

Parallelism is obtained following PASTIX static scheduling for multi-threaded architectures

## Parameters

### Entry parameters

- Tolerance  $\tau$ : absolute parameter (normalized for each block)
- Compression method: SVD or RRQR
- Compression strategy: *Minimal Memory* or *Just-In-Time*
- Blocking sizes: between 128 and 256 in following experiments

### Strategy *Minimal Memory*

- Blocks are compressed at the beginning
- Each contribution implies a recompression

### Strategy *Just-In-Time*

- Blocks are compressed just before a supernode is eliminated
- Those blocks are never uncompressed

## Costs distribution on the Atmosmodj matrix with $\tau = 10^{-8}$

	Dense	<i>Just-In-Time</i>		<i>Minimal Memory</i>	
		RRQR	SVD	RRQR	SVD
Factorization time (s)					
Compression	-	49.53	418.5	15.20	180.9
Block factorization (GETRF)	0.9635	1.000	1.003	1.074	1.104
Panel solve (TRSM)	15.80	6.970	6.526	11.16	6.946
Update					
LR product	-	64.10	91.15	193.1	94.36
LR addition	-	-	-	774.6	6523
Dense update (GEMM)	418.7	47.94	47.03	-	-
<i>Total</i>	<i>436</i>	<i>169</i>	<i>564</i>	<i>995</i>	<i>6806</i>
Solve time (s)	2.43	1.54	1.8	2.22	1.29
Factors final size (GB)	15.9	7.4	6.86	11.4	6.76
Memory peak (GB)	15.9	15.9	15.9	11.4	6.76

## Costs distribution on the Atmosmodj matrix with $\tau = 10^{-8}$

	Dense	<i>Just-In-Time</i>		<i>Minimal Memory</i>	
		RRQR	SVD	RRQR	SVD
Factorization time (s)					
<b>Compression</b>	-	49.53	418.5	15.20	180.9
Block factorization (GETRF)	0.9635	1.000	1.003	1.074	1.104
Panel solve (TRSM)	15.80	6.970	6.526	11.16	6.946
Update					
LR product	-	64.10	91.15	193.1	94.36
LR addition	-	-	-	774.6	6523
Dense update (GEMM)	418.7	47.94	47.03	-	-
<i>Total</i>	<i>436</i>	<i>169</i>	<i>564</i>	<i>995</i>	<i>6806</i>
Solve time (s)	2.43	1.54	1.8	2.22	1.29
Factors final size (GB)	15.9	7.4	6.86	11.4	6.76
Memory peak (GB)	15.9	15.9	15.9	11.4	6.76

## Costs distribution on the Atmosmodj matrix with $\tau = 10^{-8}$

	Dense	<i>Just-In-Time</i>		<i>Minimal Memory</i>	
		RRQR	SVD	RRQR	SVD
Factorization time (s)					
Compression	-	49.53	418.5	15.20	180.9
Block factorization (GETRF)	0.9635	1.000	1.003	1.074	1.104
Panel solve (TRSM)	15.80	6.970	6.526	11.16	6.946
Update					
LR product	-	64.10	91.15	193.1	94.36
LR addition	-	-	-	774.6	6523
Dense update (GEMM)	418.7	47.94	47.03	-	-
<i>Total</i>	436	169	564	995	6806
Solve time (s)	2.43	1.54	1.8	2.22	1.29
Factors final size (GB)	15.9	7.4	6.86	11.4	6.76
Memory peak (GB)	15.9	15.9	15.9	11.4	6.76



## Costs distribution on the Atmosmodj matrix with $\tau = 10^{-8}$

	Dense	<i>Just-In-Time</i>		<i>Minimal Memory</i>	
		RRQR	SVD	RRQR	SVD
Factorization time (s)					
Compression	-	49.53	418.5	15.20	180.9
Block factorization (GETRF)	0.9635	1.000	1.003	1.074	1.104
Panel solve (TRSM)	15.80	6.970	6.526	11.16	6.946
Update					
LR product	-	64.10	91.15	193.1	94.36
LR addition	-	-	-	774.6	6523
Dense update (GEMM)	418.7	47.94	47.03	-	-
<i>Total</i>	<i>436</i>	<i>169</i>	<i>564</i>	<i>995</i>	<i>6806</i>
Solve time (s)	2.43	1.54	1.8	2.22	1.29
Factors final size (GB)	15.9	7.4	6.86	11.4	6.76
Memory peak (GB)	15.9	15.9	15.9	11.4	6.76

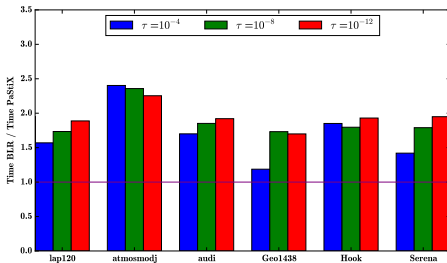
## Costs distribution on the Atmosmodj matrix with $\tau = 10^{-8}$

	Dense	<i>Just-In-Time</i>		<i>Minimal Memory</i>	
		RRQR	SVD	RRQR	SVD
Factorization time (s)					
Compression	-	49.53	418.5	15.20	180.9
Block factorization (GETRF)	0.9635	1.000	1.003	1.074	1.104
Panel solve (TRSM)	15.80	6.970	6.526	11.16	6.946
Update					
LR product	-	64.10	91.15	193.1	94.36
LR addition	-	-	-	774.6	6523
Dense update (GEMM)	418.7	47.94	47.03	-	-
<i>Total</i>	<i>436</i>	<i>169</i>	<i>564</i>	<i>995</i>	<i>6806</i>
Solve time (s)	2.43	1.54	1.8	2.22	1.29
Factors final size (GB)	15.9	7.4	6.86	11.4	6.76
Memory peak (GB)	15.9	15.9	15.9	11.4	6.76

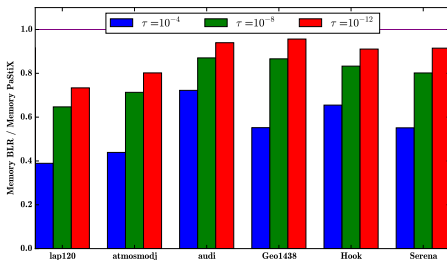
## Costs distribution on the Atmosmodj matrix with $\tau = 10^{-8}$

	Dense	<i>Just-In-Time</i>		<i>Minimal Memory</i>	
		RRQR	SVD	RRQR	SVD
Factorization time (s)					
Compression	-	49.53	418.5	15.20	180.9
Block factorization (GETRF)	0.9635	1.000	1.003	1.074	1.104
Panel solve (TRSM)	15.80	6.970	6.526	11.16	6.946
Update					
LR product	-	64.10	91.15	193.1	94.36
LR addition	-	-	-	774.6	6523
Dense update (GEMM)	418.7	47.94	47.03	-	-
<i>Total</i>	<i>436</i>	<i>169</i>	<i>564</i>	<i>995</i>	<i>6806</i>
Solve time (s)	2.43	1.54	1.8	2.22	1.29
Factors final size (GB)	15.9	7.4	6.86	11.4	6.76
Memory peak (GB)	15.9	15.9	15.9	11.4	6.76

## Behaviour of RRQR/Minimal Memory

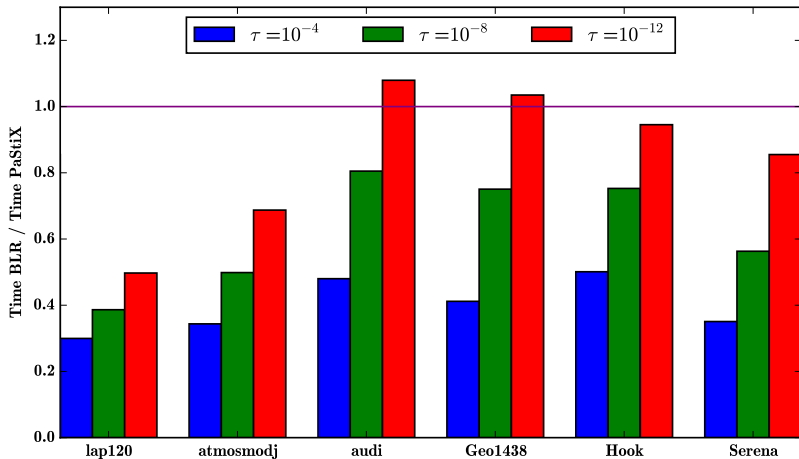


Performance

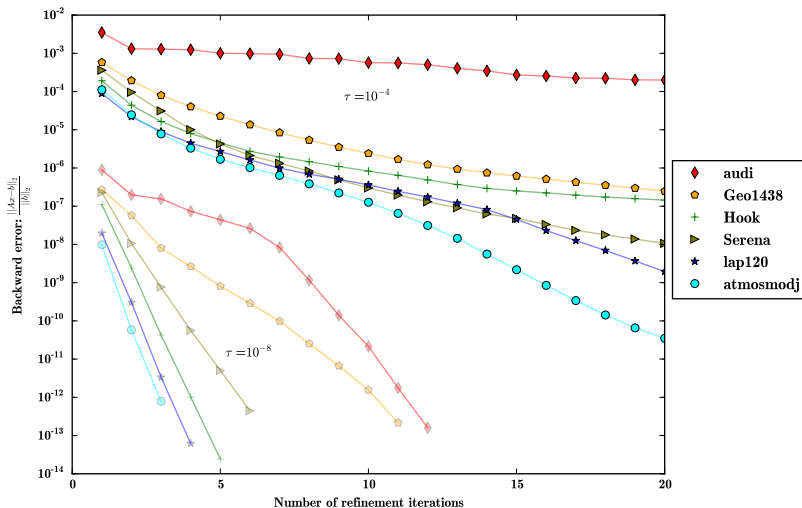


Memory footprint

## Performance of RRQR/*Just-In-Time*

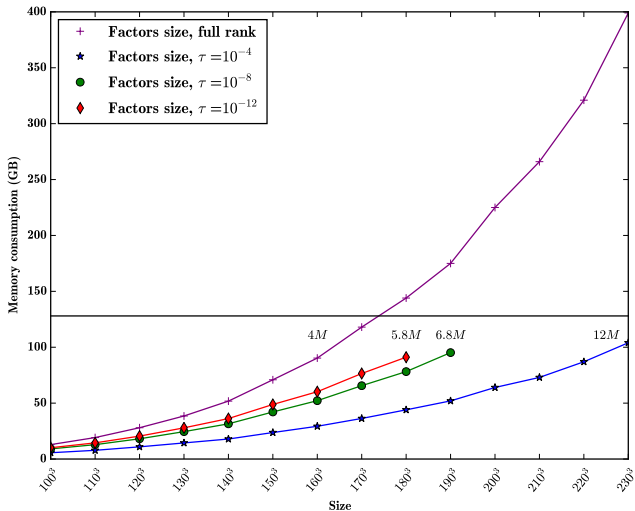


# Convergence of RRQR/Minimal Memory

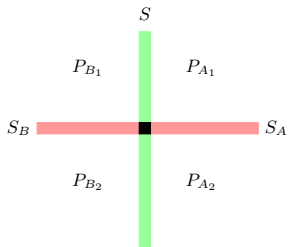


# Scaling on Laplacians: Memory Consumption

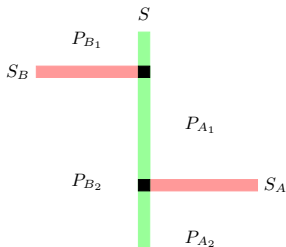
## RRQR/Minimal Memory



## Ongoing work with E. Esnard and M. Predari



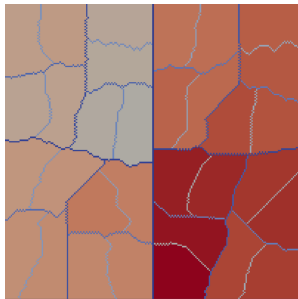
- Single interaction between  $S$  and  $S_A \cup S_B$  in the original graph
- Two large interaction blocks  $P_{A_1} \cup P_{B_1}$  and  $P_{A_2} \cup P_{B_2}$  during the factorization



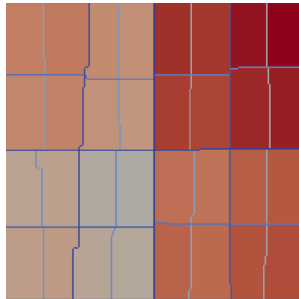
- Double interaction: between  $S$  and  $S_A$  and between  $S$  and  $S_B$  in the original graph
- Three “large” interaction blocks during the factorization



## Example on a $200^2$ Laplacian



Scotch ordering



Aligned ordering

# Conclusion

## Block Low-Rank solver

- Allows to see how to use the symbolic structure and the extend-add issues in a supernodal context
- The version presented follows the parallelism of PASTIX
- On-going work to implement efficiently this approach over the PARSEC runtime system

## Ordering strategies

- Study of a nested dissection designed to increase compressibility
- Add pre-selection to isolate data which is not compressible, and thus reduce the overhead of trying to compress uncompressible blocks

## PASTIX 6.0.0alpha is now available!

<http://gitlab.inria.fr/solverstack/pastix>

- Support shared memory with different schedulers:
  - ▶ sequential
  - ▶ static scheduler
  - ▶ PaRSEC runtime system
- Low-rank support
- Cholesky and LU factorizations

Thank you.