



université
de BORDEAUX



On the use of low-rank arithmetic to reduce the complexity of parallel sparse linear solvers based on direct factorization techniques

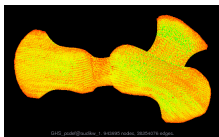
November 29th, 2018 - PhD defense

Grégoire Pichon^a, supervised by E. Darve^b, M. Faverge^a, P. Ramet^a and J. Roman^a

^aInria, Bordeaux INP, CNRS, Université de Bordeaux

^bStanford University

Context

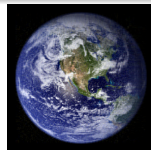


How to quickly solve $Ax = b$?

- Main kernel in the domain of computational science and engineering
- Often issued from the discretization of Partial Differential Equations
- Remote interactions are neglected: the number of non-zeroes is $\Theta(n)$



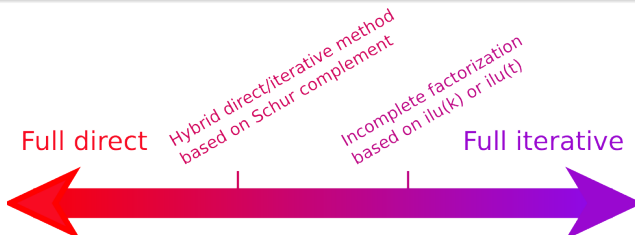
Original picture



Compressed picture with only 20% of original storage cost

Sparse linear solvers

- Iterative methods:
 - ▶ Improvement of an initial guess by an iterative process
 - ▶ Fast and low memory consumption
 - ▶ Depend on the problem and the initial guess
- Direct methods:
 - ▶ Factorization of the matrix of the sparse linear system into two triangular matrices for a simpler system solve
 - ▶ Accurate, robust and problem independent
 - ▶ High memory and time consumption
- Hybrid methods:
 - ▶ Try to gather the advantages of both categories



Overview of the thesis

Current sparse direct solver for 3D problems of size n

- $\Theta(n^2)$ time complexity
- $\Theta(n^{\frac{4}{3}})$ memory complexity
- BLAS Level 3 operations with computations on blocks

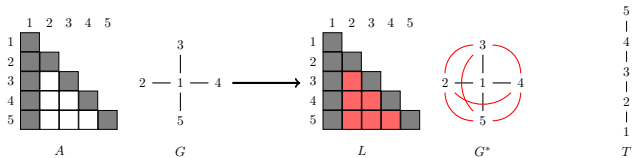
Contributions

- Introduction of the Block Low-Rank compression into the supernodal solver PASTIX
- Reordering strategy to reduce the number of elemental low-rank updates
- Clustering heuristic to increase the overall compressibility
- Partitioning strategy to obtain more regular patterns
- Study of BLR compression when using runtime systems
- Impact on a real-life application that uses domain decomposition

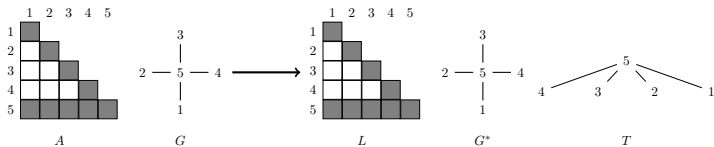
1

Background

Fill-in problem with A symmetric



Natural ordering

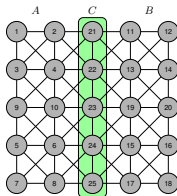


Optimal ordering

Ordering with Nested Dissection

Algorithm to compute the permutation of V vertices

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



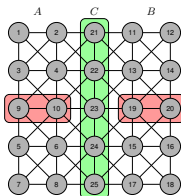
Nested dissection performed by an external partitioner tool

- Find a separator C as small as possible
- Exhibit balanced subparts A and B

Ordering with Nested Dissection

Algorithm to compute the permutation of V vertices

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



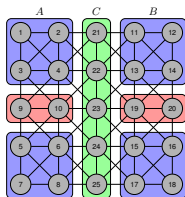
Nested dissection performed by an external partitioner tool

- Find a separator C as small as possible
- Exhibit balanced subparts A and B

Ordering with Nested Dissection

Algorithm to compute the permutation of V vertices

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



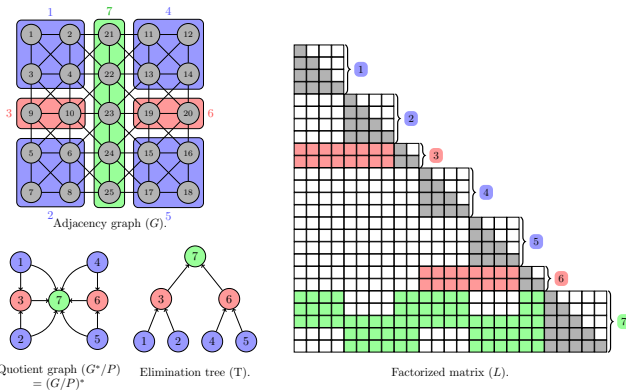
Nested dissection performed by an external partitioner tool

- Find a separator C as small as possible
- Exhibit balanced subparts A and B

Symbolic factorization

General approach

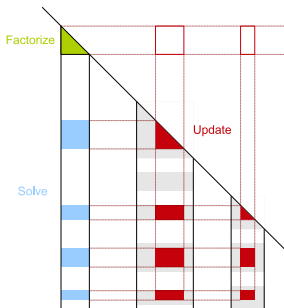
1. Build a partition with the nested dissection process
2. Compress information on data blocks
3. Compute the block elimination tree using the block quotient graph



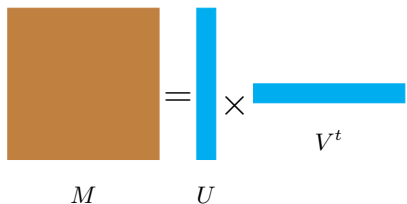
Numerical factorization

Algorithm to eliminate the 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)



Low-rank compression

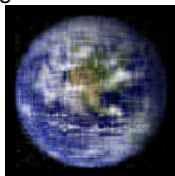

$$M = U \times V^t$$

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

Storage in $2nr$ instead of n^2



Original picture



$r = 10$, 4% of original storage



$r = 50$, 20% of original storage

Compression of an image with $n = 500$

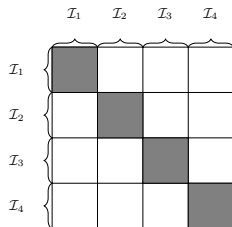
Compression kernels

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

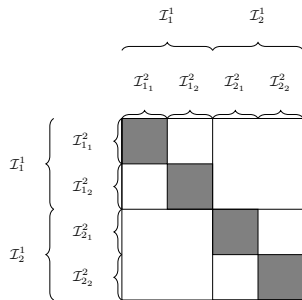
Properties

- SVD provides the best ranks at a given accuracy with $\|\cdot\|_2$
- 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



BLR clustering



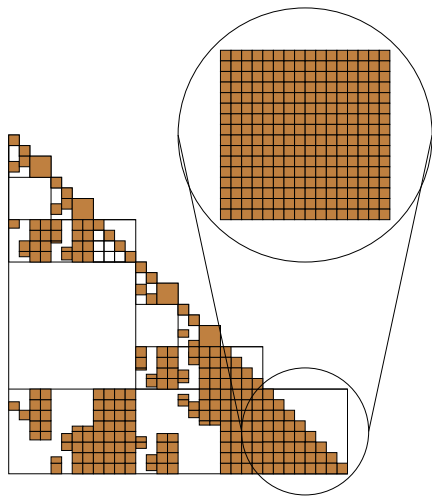
HODLR clustering

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

2

Sparse supernodal solver with BLR compression

BLR compression – Symbolic factorization



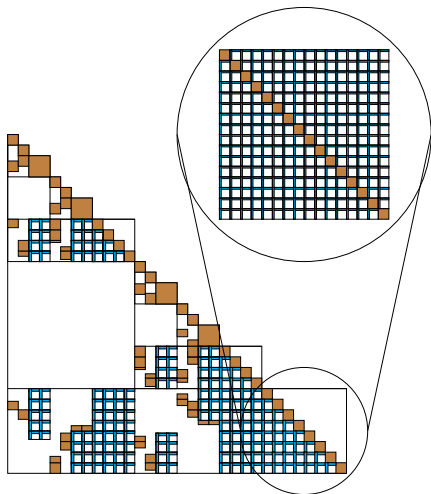
Approach

- 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

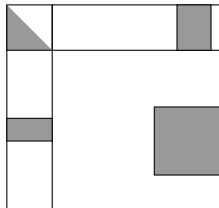
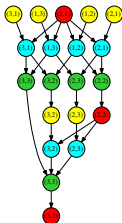
- Dense diagonal blocks
- TRSM are performed on **low-rank** off-diagonal blocks
- GEMM are performed between **low-rank** off-diagonal blocks

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)

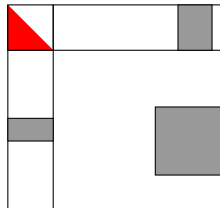
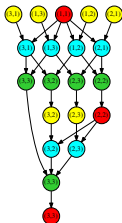


Strategy *Just-In-Time*

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






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

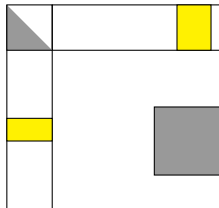
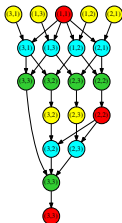


Strategy *Just-In-Time*

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)

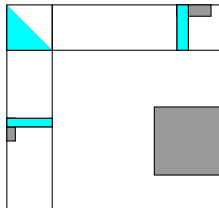
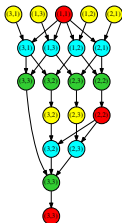


Strategy *Just-In-Time*

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)

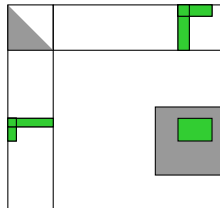
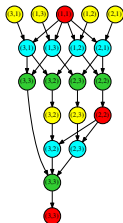


Strategy *Just-In-Time*

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)



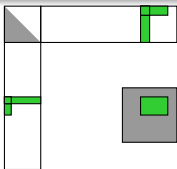
Summary of the *Just-In-Time* strategy

Advantages

- The most expensive operation, the update, 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

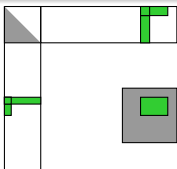
Summary of the *Just-In-Time* strategy

Advantages

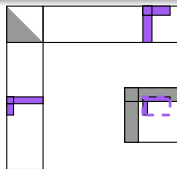
- The most expensive operation, the update, is faster using *LR2GE* kernel
- The formation of the dense update and its application is not expensive
- The size of the factors is reduced, as well as the solve cost

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



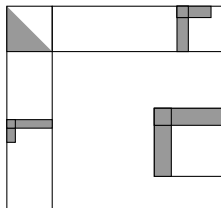
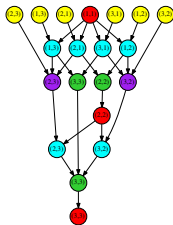
Minimal Memory

Strategy *Minimal Memory*

Compress A

1. Compress large off-diagonal blocks in A (exploiting sparsity)
2. Eliminate each column block
 - 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)

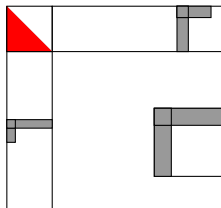
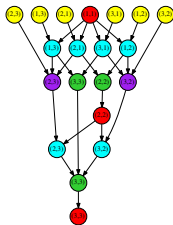


Strategy *Minimal Memory*

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)

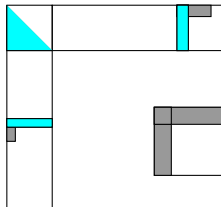
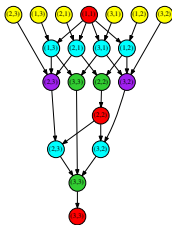


Strategy *Minimal Memory*

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)

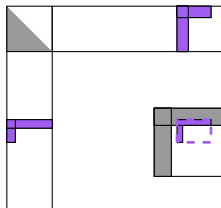
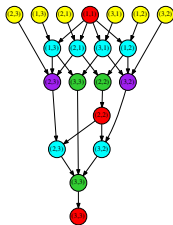


Strategy *Minimal Memory*

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)



LR2LR kernel using SVD

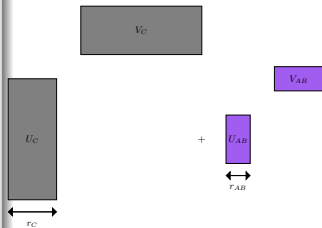
A low-rank structure $U_C V_C^t$ receives a low-rank contribution $U_{AB} V_{AB}^t$

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 the target C
Slightly more complex algorithm for RRQR, that still requires zeroes padding

LR2LR kernel using SVD

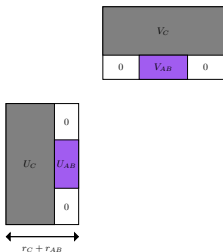
A low-rank structure $U_C V_C^t$ receives a low-rank contribution $U_{AB} V_{AB}^t$

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 the target C
Slightly more complex algorithm for RRQR, that still requires zeroes padding

LR2LR kernel using SVD

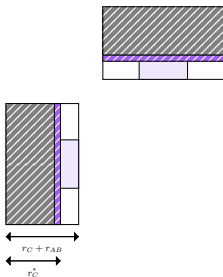
A low-rank structure $U_C V_C^t$ receives a low-rank contribution $U_{AB} V_{AB}^t$

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 the target C
Slightly more complex algorithm for RRQR, that still requires zeroes padding

Experimental setup

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

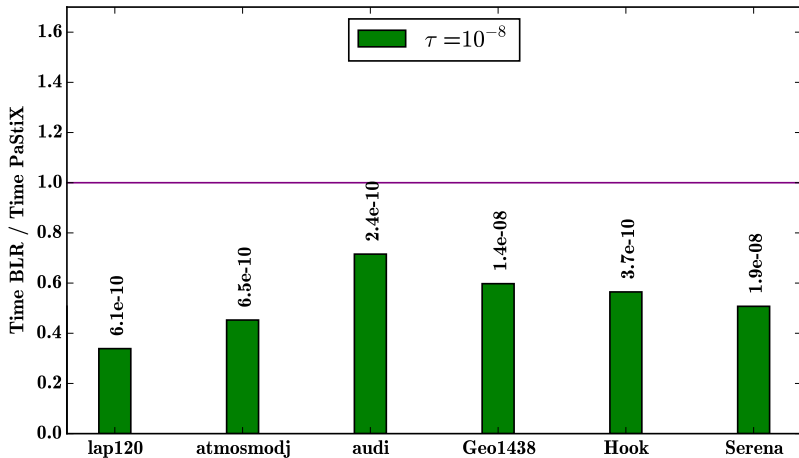
- 128 GB
- 24 threads
- Parallelism is obtained following PASTIX static scheduling for multi-threaded architectures (PARSEC version will be discussed later)

Entry parameters

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

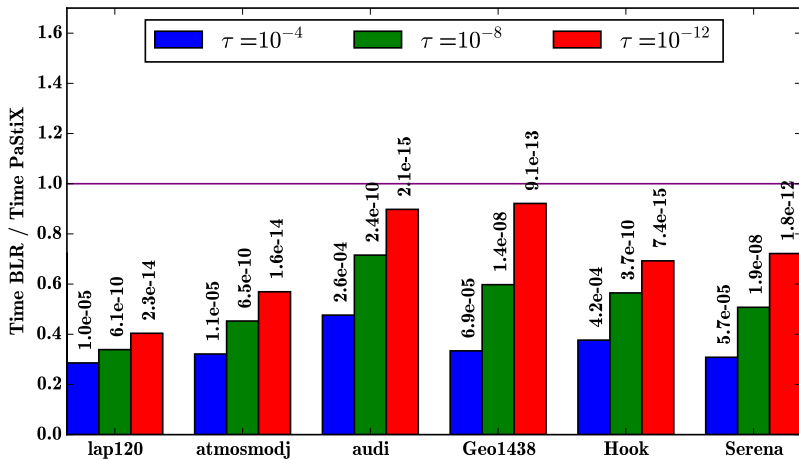
Matrix	Fact.	Size	Field
<i>lap120</i>	<i>LDL^t</i>	120 ³	Poisson problem
<i>Atmosmodj</i>	<i>LU</i>	1 270 432	atmospheric model
<i>Audi</i>	<i>LL^t</i>	943 695	structural problem
<i>Geo1438</i>	<i>LL^t</i>	1 437 960	geomechanical model of earth
<i>Hook</i>	<i>LDL^t</i>	1 498 023	model of a steel hook
<i>Serena</i>	<i>LDL^t</i>	1 391 349	gas reservoir simulation

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



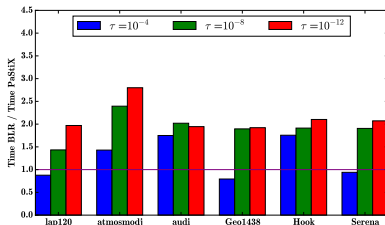
Factorization time reduced by a factor of 2 for $\tau = 10^{-8}$

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



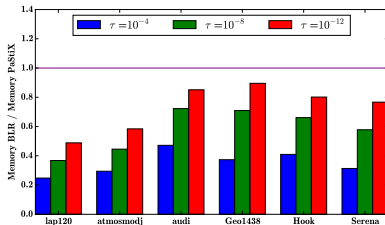
Factorization time reduced by a factor of 2 for $\tau = 10^{-8}$

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

Summary of the solver

- Fully algebraic with a supernodal approach
- Low-rank assembly with the *Minimal Memory* strategy
- 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

Related work

BLR

MUMPS BLR compression in a multifrontal solver with dense assembly, which can perform pivoting

Hierarchical

H-LIB Hierarchical matrices for sparse by Hackbusch et al. which does not exploit all structural zeroes

Hmat-OSS A. Falco's thesis that uses hierarchical matrices together with a symbolic factorization

CHOLMOD Supernodal solver using randomization with fixed rank

STRUMPACK HSS by Ghysels et al. with randomized sampling

Many other solvers

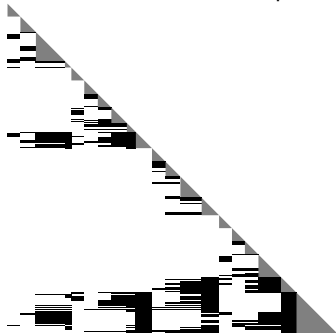
- Weak admissibility: HODLR, HSS
- Nested bases: HSS, \mathcal{H}^2
- Often used as preconditioners

3

Ordering for low-rank compression in sparse solvers

Ordering for sparse matrices

One can reorder separators without modifying the fill-in

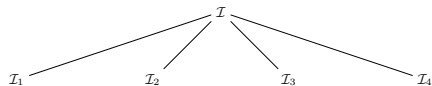


Symbolic factorization of a $8 \times 8 \times 8$ Laplacian

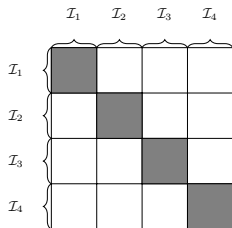
Enhance ordering

- Modify the partitioning process to enhance compressibility
- Reorder unknowns within a separator:
 - ▶ To enhance clustering within the separator
 - ▶ To enhance the coupling part

BLR clustering for a dense matrix



Cluster tree



Flat clustering

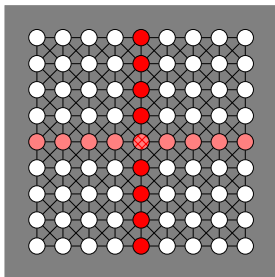
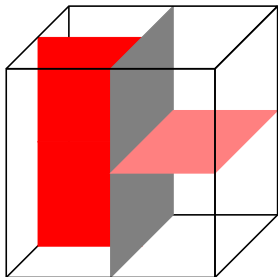
$\sigma \times \tau$ is admissible $\iff \max(\text{diam}(\sigma), \text{diam}(\tau)) \leq \eta \text{dist}(\sigma, \tau)$

Can be computed with the knowledge of geometry; otherwise, k-way partitioning can be used for clustering separators

Towards sparse ordering dedicated to low-rank

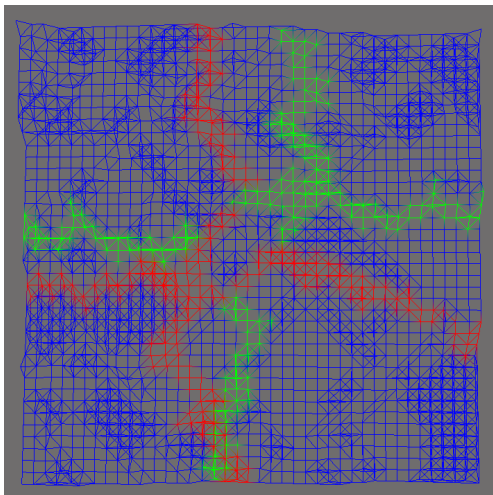
Nested dissection

1. Partition $V = A \cup B \cup C$
2. Order C with larger indices: $V_A < V_C$ and $V_B < V_C$
3. Apply the process recursively on A and B



As A and B are processed independently, their interaction with C is not the same, even for a perfect nested dissection of a $N \times N \times N$ cube

Example on a cube partitioned with SCOTCH



First separator of a $40 \times 40 \times 40$ Laplacian partitioned with SCOTCH

Dedicated ordering for low-rank compression

The ordering of unknowns within a separator does not impact the memory consumption nor the number of operations

Reordering strategy

- Minimize the number of off-diagonal blocks
- Reduce the number of elemental low-rank updates

Clustering heuristic

- Form well-separated clusters
- Increase the overall compressibility

Partitioning strategy

- Align separators
- Obtain more regular patterns, as in geometric solvers

Dedicated ordering for low-rank compression

The ordering of unknowns within a separator does not impact the memory consumption nor the number of operations

Reordering strategy

- Minimize the number of off-diagonal blocks
- Reduce the number of elemental low-rank updates

Clustering heuristic

- Form well-separated clusters
- Increase the overall compressibility

Partitioning strategy

- Align separators
- Obtain more regular patterns, as in geometric solvers

Reordering problem

Existing approaches

- Ordering of separators on the local graph
- Reverse Cuthill-McKee performs a Breadth First Search to order unknowns
- It is not designed for direct solvers since matrices of separators will become full with the fill-in

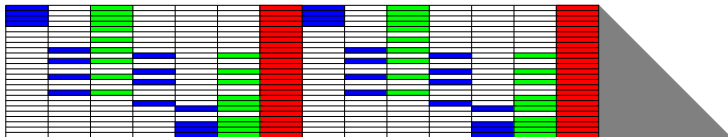
Proposition

- Ordering of separators to minimize the number of off-diagonal blocks
- Does not impact memory consumption or the number of operations
- Reduce the number of low-rank updates
- Increase granularity
 - ▶ Enhance the use of heterogeneous architectures (GPUs, Xeon Phi)
 - ▶ Reduce the overhead associated with runtime systems

Modeling of the problem

Proposition

- Define a distance between rows: the number of differences between off-diagonal blocks
- Express the problem as a Traveling Salesman Problem (TSP) to sort rows in order to minimize the overall distance
- Use heuristics to perform TSP with low complexity



Modeling of the problem

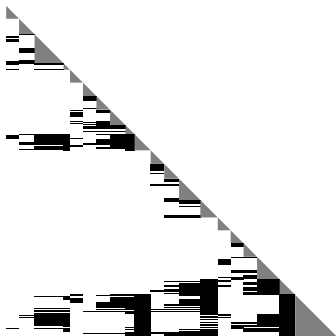
Proposition

- Define a distance between rows: the number of differences between off-diagonal blocks
- Express the problem as a Traveling Salesman Problem (TSP) to sort rows in order to minimize the overall distance
- Use heuristics to perform TSP with low complexity

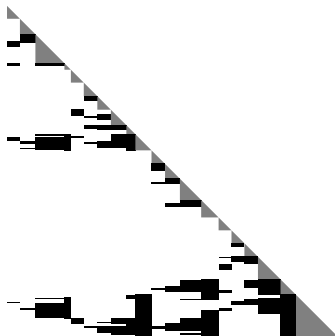
1				
2				
3	↕ +1	↕ +1	↕ +0	↕ +0
4				

	1	2	3	4
1	0	-	-	-
2	3	0	-	-
3	3	2	0	-
4	1	4	2	0

Resulting solution - Example



Without reordering (RCM)



With reordering

Reordering on a $8 \times 8 \times 8$ Laplacian

Works with any initial seed

Experimental setup

Set of matrices

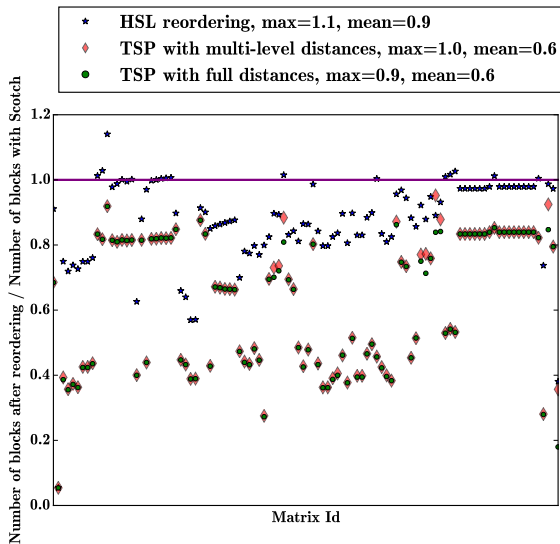
- 104 matrices issued from the SuiteSparse Matrix Collection
- Matrices with $500,000 \leq N \leq 10,000,000$
- Web and DNA matrices were removed

Strategy studied

- TSP
- TSP with multi-level distances: heuristic to reduce the cost of TSP when computing distances between rows
- Reordering in HSL, developed at STFC

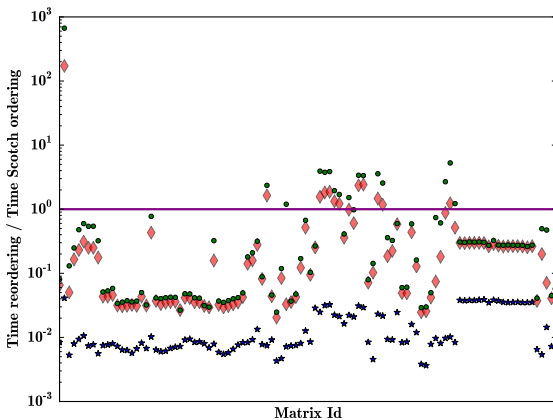
Another strategy introduced by M. Jacquelin et al. (2018) is faster than TSP while quality is close to TSP in most cases

Number of off-diagonal blocks



Reordering cost (sequential)

- TSP with full distances, max=668.0, mean=7.0(0.6)
- ◆ TSP with multi-level distances, max=172.0, mean=2.0(0.3)
- * HSL reordering, max=4e-02, mean=1e-02



Summary

Results

- Reduce the number of off-diagonal blocks and thus the overhead associated with low-rank updates
- Lead to larger data blocks suitable for modern architectures
- Always increase performance wrt SCOTCH

Architecture	Nb. units	Mean gain	Max gain
Westmere	12 cores	2%	6%
Xeon E5	24 cores	7%	13%
Fermi	12 cores + 1 to 3 M2070	10%	20%
Kepler	24 cores + 1 to 4 K40	15%	40%
Xeon Phi	64 cores	20%	40%

Performance gain for the full-rank factorization when using PARSEC runtime system with TSP instead of SCOTCH

Dedicated ordering for low-rank compression

The ordering of unknowns within a separator does not impact the memory consumption nor the number of operations

Reordering strategy

- Minimize the number of off-diagonal blocks
- Reduce the number of elemental low-rank updates

Clustering heuristic

- Form well-separated clusters
- Increase the overall compressibility

Partitioning strategy

- Align separators
- Obtain more regular patterns, as in geometric solvers

Suitable clustering strategy

Let us consider the last separator A_{22} and its interactions with previous supernodes A_{11} :

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

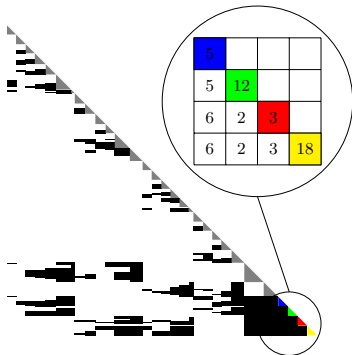
Existing approaches

- K-way partitioning orders correctly A_{22}
- The reordering strategy reduces the number of off-diagonal blocks in A_{12} and A_{21}

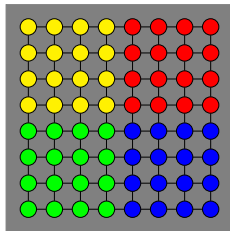
We introduce a new strategy to couple the advantages of both approaches, with suitable structure for both intra and inter separators blocks

Clustering techniques: existing solutions

- k-way partitioning (dense, multifrontal)



(a) Symbolic factorization

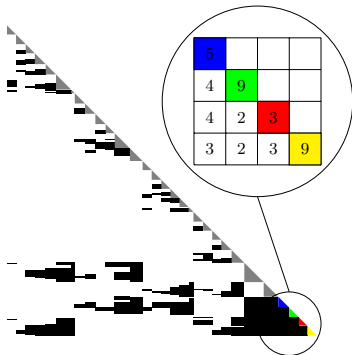


(b) First separator clustering

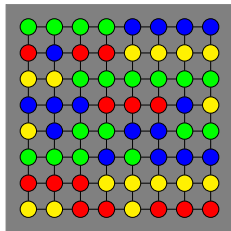
$8 \times 8 \times 8$ Laplacian partitioned using SCOTCH and k-way clustering on the first separator

Clustering techniques: existing solutions

- k-way partitioning (dense, multifrontal)
- Supernode reordering techniques (supernodal)



(a) Symbolic factorization



(b) First separator clustering

$8 \times 8 \times 8$ Laplacian partitioned using SCOTCH and Reordering clustering on the first separator

New heuristic based on projections

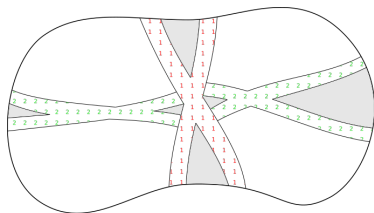
Pre-selected vertices of a separator are vertices connected to a close child in the elimination tree

For each separator we have:

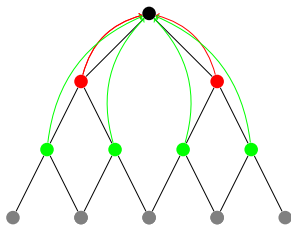
$$A_{22} = \begin{pmatrix} A_{kk} & A_{ks} \\ A_{sk} & A_{ss} \end{pmatrix}$$

where A_{ss} corresponds to pre-selected vertices.

Only interactions between non pre-selected vertices A_{kk} are compressed

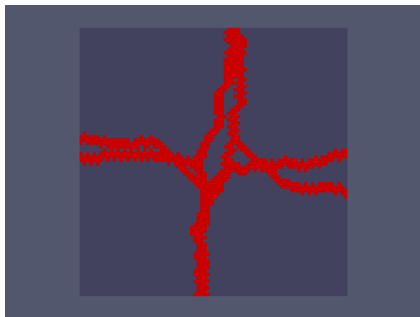


- Trace of the 1st separator
- Trace of the 2nd separator
- #nodes > threshold
- #nodes < threshold



Example - $80 \times 80 \times 80$ Laplacian matrix

Interactions with pre-selected vertices (in red) are not compressed

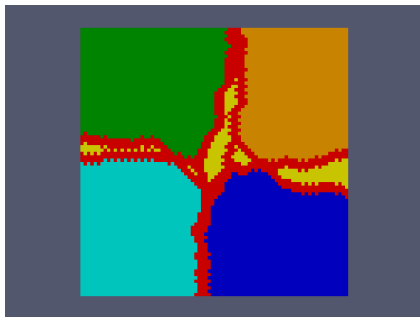


1. Compute pre-selected vertices

Projections heuristic on last separator of size 80×80

Example - $80 \times 80 \times 80$ Laplacian matrix

Interactions with pre-selected vertices (in red) are not compressed

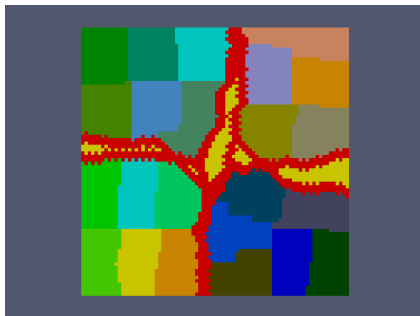


1. Compute pre-selected vertices
2. Isolate connected components
3. Merge small components

Projections heuristic on last separator of size 80×80

Example - $80 \times 80 \times 80$ Laplacian matrix

Interactions with pre-selected vertices (in red) are not compressed



Projections heuristic on last separator of size 80×80

1. Compute pre-selected vertices
2. Isolate connected components
3. Merge small components
4. Apply k-way partitioning on large components
5. Perform TSP reordering on each cluster

Experimental setup

Architecture

- 2 INTEL Xeon E5 – 2680v3 at 2.50 GHz
- 128 GB
- 24 threads

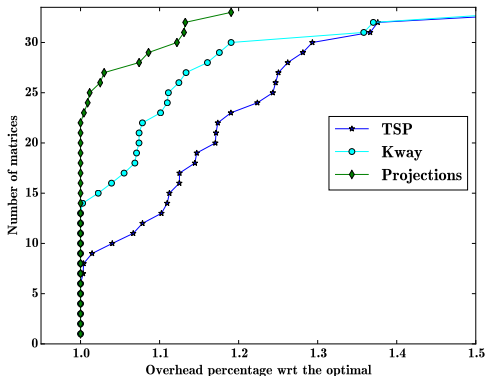
Set of matrices

- 33 square matrices from the SuiteSparse Matrix Collection
- real or complex
- $50K \leq N \leq 5M$
- $nb_{ops} > 1$ TFlops

Strategies studied

- Reordering with TSP
- K-way partitioning using SCOTCH + TSP on each cluster
- The new Projections strategy + TSP on each cluster

Performance profile of the factorization time

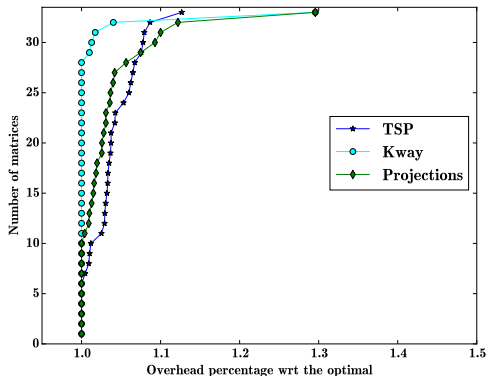


Method

- For each matrix, percentage wrt the best method
- Percentage are sorted increasingly
- A curve close to $x = 1$ is close to optimal

Minimal Memory, with $\tau = 10^{-8}$

Performance profile of the memory consumption



Method

- For each matrix, percentage wrt the best method
- Percentage are sorted increasingly
- A curve close to $x = 1$ is close to optimal

Minimal Memory, with $\tau = 10^{-8}$

Summary

Results

- Reduces factorization time, especially when using the *Minimal Memory* strategy
- Slight increase of the memory consumption since some interactions are not compressed

Impact on preprocessing stats

- Ordering time controlled: 70% overhead with respect to SCOTCH ordering
- Slight increase in the number of off-diagonal blocks: mean 1.05, max 1.35 for both *K-way* and *Projections* strategies

4

Conclusion and future work

Summary of the work presented in this talk

Block Low-Rank solver

- Allows us to see how to use the symbolic structure and the extend-add issues in a supernodal context
- The BLR version follows the parallelism of the full-rank version of PASTIX

Ordering strategies

- Reordering strategy to reduce the number of off-diagonal blocks
- Clustering heuristic to enhance low-rank compressibility

Summary of the work not presented in this talk

Aligning separators

- Ongoing work to align separators
- Should reduce the number of off-diagonal blocks
- Should increase compressibility

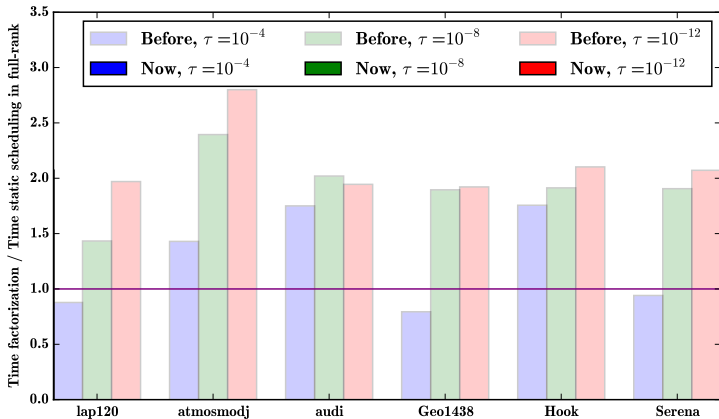
Runtime systems

- Work over the PARSEC and STARPU runtime systems
- Enhance load balancing

Integration into HORSE

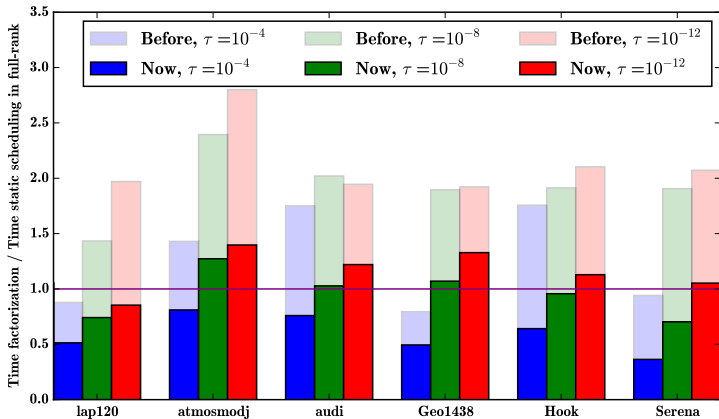
- Full-rank version replaced by the low-rank strategies in a domain decomposition solver
- Demonstrates gains in a large real-life application

Behavior of RRQR/*Minimal Memory* with PARSEC and Projections



Adding Projections and the use of PARSEC runtime system reduces factorization time

Behavior of RRQR/*Minimal Memory* with PARSEC and Projections



Adding Projections and the use of PARSEC runtime system reduces factorization time

Future work

Extensions of this work

- Experiment the impact of ordering strategies in other solvers (MUMPS for instance)
- Ongoing work to better align separators

Future work

- Hierarchical matrices
- Distributed version (load balancing)
- Low-rank operations on GPUs
- Complexity study for general 2D/3D meshes
- Use other low-rank compression kernels, such as randomization
- Compare with other preconditioners such as ILU

Thank you.

Questions?