



HAL
open science

Utilisation de la compression Block Low-Rank pour accélérer un solveur direct creux supernodal

Grégoire Pichon, Eric Darve, Mathieu Faverge, Pierre Ramet, Jean Roman

► To cite this version:

Grégoire Pichon, Eric Darve, Mathieu Faverge, Pierre Ramet, Jean Roman. Utilisation de la compression Block Low-Rank pour accélérer un solveur direct creux supernodal. COMPAS 2018 - Conférence d'informatique en Parallélisme, Architecture et Système, Jul 2018, Toulouse, France. hal-01956959

HAL Id: hal-01956959

<https://inria.hal.science/hal-01956959v1>

Submitted on 16 Dec 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Utilisation de la compression Block Low-Rank pour accélérer un solveur direct creux supernodal

30 Juin 2018 – Compas

Grégoire Pichon^a, Eric Darve^b, Mathieu Faverge^a, Pierre Ramet^a, Jean Roman^a

^aInria, Bordeaux INP, CNRS, Université de Bordeaux

^bStanford University

Introduction

Solveurs direct creux pour les problèmes 3D de taille n

- Complexité en temps en $\Theta(n^2)$
- Complexité mémoire en $\Theta(n^{\frac{4}{3}})$
- Opérations de type BLAS 3: performance correcte

Introduction de la compression Low-Rank

- Compression des gros blocs à une précision donnée
- Peu d'impact sur le parallélisme de PASTIX
- La stratégie *Minimal Memory* permet de réduire l'empreinte mémoire
- La stratégie *Just-In-Time* permet de réduire le temps de résolution

Objectif: intégrer de manière algébrique des techniques de compression de données au solveur PASTIX.

1

Contexte

Solveur direct creux

Problème – résoudre $Ax = b$

- Cholesky: factorise $A = LL^T$ (structure symétrique $(A + A^T)$ pour LU)
- Résolution de $Ly = b$
- Résolution de $L^T x = y$

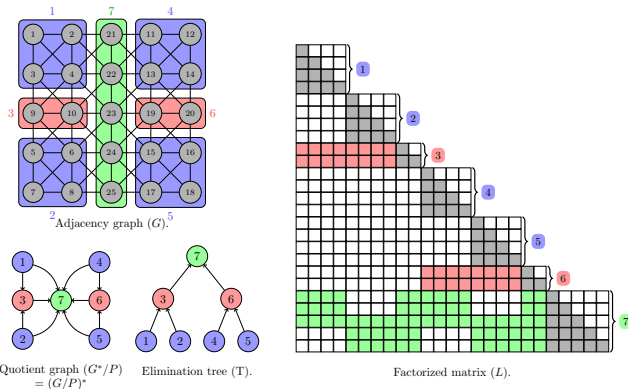
Approche suivie dans PASTIX

1. Numérotation des inconnues pour minimiser le remplissage
2. Factorisation symbolique: calcul de la structure de L
3. Factorisation de la matrice en place sur la structure de L
4. Résolution de systèmes triangulaires

Factorisation Symbolique

Approche globale

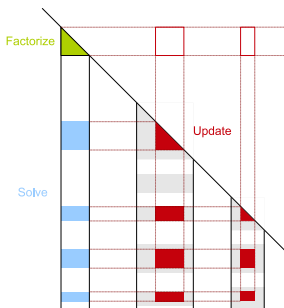
1. Construction d'une partition avec la dissection emboîtée
2. Gestion des inconnues par blocs
3. Construction de l'arbre d'élimination des blocs d'inconnues



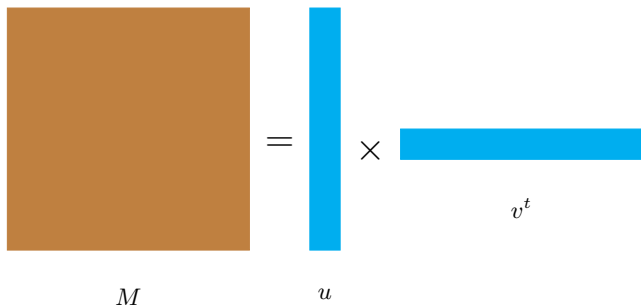
Factorisation numérique

Algorithme pour éliminer le supernoœud k

1. **Factorize** le bloc diagonal (POTRF/GETRF)
2. **Solve** sur les blocs extra-diagonaux du supernoœud (TRSM)
3. **Update** de la matrice sous-jacente avec la contribution du supernoœud (GEMM)



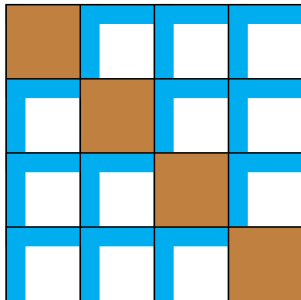
Compression Low-Rank



$$M \in \mathbb{R}^{n \times n}; u, v \in \mathbb{R}^{n \times r}$$

Stockage de $2nr + r^2$ au lieu de n^2
Différents noyaux de compression: SVD, RRQR, BDLR, ACA...

Compression Block Low-Rank



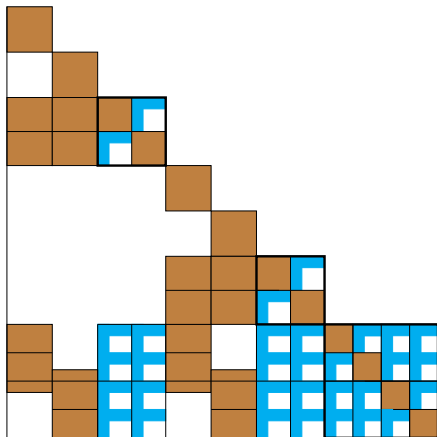
 Full Rank

 Low Rank

2

Opérations Block Low-Rank

Factorisation Symbolique avec compression BLR



Les gros blocs extra-diagonaux sont low-rank, de la forme uv^t .

Quelques solveurs concurrents...

Solveurs Block Low-Rank

- BLR-MUMPS permet de réduire le temps de factorisation (similaire à notre stratégie *Just-In-Time* mais dans une approche multifrontale)
- Solveur dense développé au LSTC avec des techniques de recompression

D'autres approches pour le creux

- Strumpack par P. Ghysels et al. utilise le format HSS
- HODLR introduit par E. Darve et al.
- \mathcal{H} -LU proposé par Hackbusch et al.

Présentation à suivre d'Aurélien Falco!

Algorithme Block Low-Rank

Idée générale

- Les gros supernoeuds sont découpés en un ensemble de petits supernoeuds
- Les gros blocs extra-diagonaux sont ensuite compressés

Opérations

- Les blocs diagonaux sont denses
- L'opération TRSM est effectuée sur des blocs low-rank
- L'opération d'Update est réalisée entre des blocs low-rank. La contribution se fait vers des blocs denses (*Just-In-Time*) ou des blocs low-rank (*Minimal Memory*)

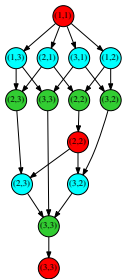
Techniques de compression

- $\|B - u_B v_B^t\|_2 \leq \tau$
- SVD, RRQR actuellement (autres méthodes: ACA, BDLR...)

Deux nouvelles stratégies

Stratégies

Yellow	Compression
Red	GETRF
Cyan	TRSM
Purple	LR2LR
Green	LR2GE



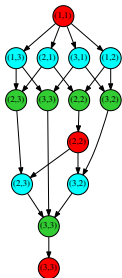
Full-rank

Deux nouvelles stratégies

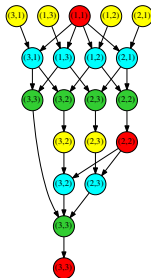
Stratégies

- *Just-In-Time* pour réduire le temps de résolution

Yellow	Compression
Red	GETRF
Cyan	TRSM
Purple	LR2LR
Green	LR2GE



Full-rank



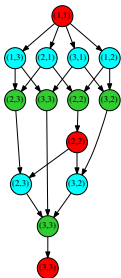
Just-In-Time

Deux nouvelles stratégies

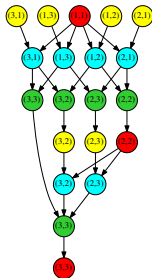
Stratégies

- *Just-In-Time* pour réduire le temps de résolution
- *Minimal Memory* pour réduire la consommation mémoire et résoudre de plus gros problèmes

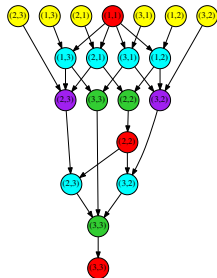
Yellow	Compression
Red	GETRF
Cyan	TRSM
Purple	LR2LR
Green	LR2GE



Full-rank








Just-In-Time

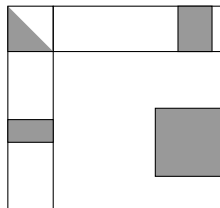
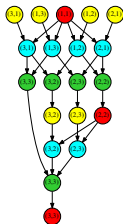


Minimal Memory

Stratégie *Just-In-Time*






1. Élimination de chaque supernoeud
 - 1.1 Factorisation du bloc diagonal dense
Compression des blocs extra-diagonaux du supernoeud
 - 1.2 Application d'un Solveur sur les blocs LR
 - 1.3 Update LR sur des matrices denses (*LR2GE* extend-add)
2. Résolution de systèmes triangulaires avec des blocs LR

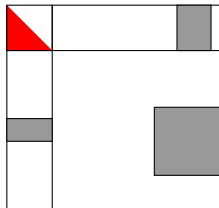
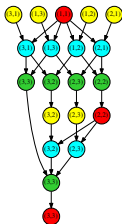
	Compression
	GETRF (Facto)
	TRSM (Solve)
	LR2LR (Update)
	LR2GE (Update)



Stratégie *Just-In-Time*






1. Élimination de chaque supernoeud
 - 1.1 Factorisation du bloc diagonal dense
Compression des blocs extra-diagonaux du supernoeud
 - 1.2 Application d'un Solveur sur les blocs LR
 - 1.3 Update LR sur des matrices denses (*LR2GE* extend-add)
2. Résolution de systèmes triangulaires avec des blocs LR

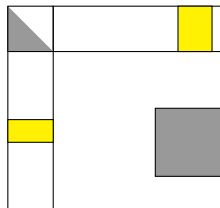
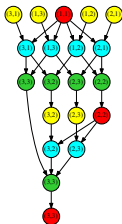
	Compression
	GETRF (Facto)
	TRSM (Solve)
	LR2LR (Update)
	LR2GE (Update)



Stratégie *Just-In-Time*






1. Élimination de chaque supernoœud
 - 1.1 Factorisation du bloc diagonal dense
Compression des blocs extra-diagonaux du supernoœud
 - 1.2 Application d'un Solveur sur les blocs LR
 - 1.3 Update LR sur des matrices denses (*LR2GE* extend-add)
2. Résolution de systèmes triangulaires avec des blocs LR

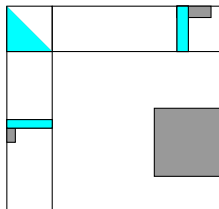
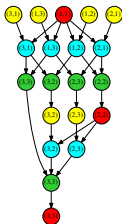
	Compression
	GETRF (Facto)
	TRSM (Solve)
	LR2LR (Update)
	LR2GE (Update)



Stratégie *Just-In-Time*






1. Élimination de chaque supernoeud
 - 1.1 Factorisation du bloc diagonal dense
Compression des blocs extra-diagonaux du supernoeud
 - 1.2 Application d'un Solve sur les blocs LR
 - 1.3 Update LR sur des matrices denses (*LR2GE* extend-add)
2. Résolution de systèmes triangulaires avec des blocs LR

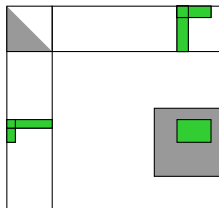
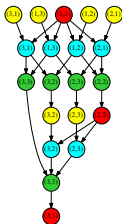
	Compression
	GETRF (Facto)
	TRSM (Solve)
	LR2LR (Update)
	LR2GE (Update)



Stratégie *Just-In-Time*

1. Élimination de chaque supernoeud
 - 1.1 Factorisation du bloc diagonal dense
Compression des blocs extra-diagonaux du supernoeud
 - 1.2 Application d'un Solveur sur les blocs LR
 - 1.3 Update LR sur des matrices denses (*LR2GE* extend-add)
2. Résolution de systèmes triangulaires avec des blocs LR

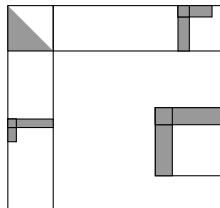
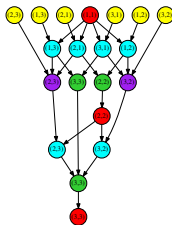
	Compression
	GETRF (Facto)
	TRSM (Solve)
	LR2LR (Update)
	LR2GE (Update)



Stratégie *Minimal Memory*

1. Compression des gros blocs extra-diagonaux de A (en exploitant leur caractère creux)
2. Élimination de chaque supernoeud
 - 2.1 Factorisation du bloc diagonal dense
 - 2.2 Application d'un Solve sur les blocs LR
 - 2.3 Update LR sur des matrices LR ($LR2LR$ extend-add)
3. Résolution de systèmes triangulaires avec des blocs LR

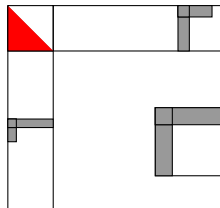
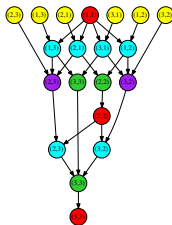
Yellow	Compression
Red	GETRF (Facto)
Cyan	TRSM (Solve)
Purple	LR2LR (Update)
Green	LR2GE (Update)



Stratégie *Minimal Memory*

1. Compression des gros blocs extra-diagonaux de A (en exploitant leur caractère creux)
2. Élimination de chaque supernoeud
 - 2.1 Factorisation du bloc diagonal dense
 - 2.2 Application d'un Solve sur les blocs LR
 - 2.3 Update LR sur des matrices LR ($LR2LR$ extend-add)
3. Résolution de systèmes triangulaires avec des blocs LR

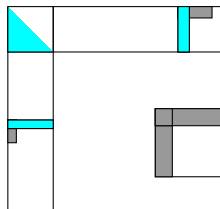
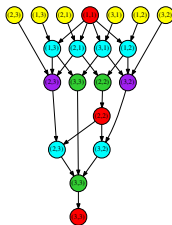
Yellow	Compression
Red	GETRF (Facto)
Cyan	TRSM (Solve)
Purple	LR2LR (Update)
Green	LR2GE (Update)



Stratégie *Minimal Memory*

1. Compression des gros blocs extra-diagonaux de A (en exploitant leur caractère creux)
2. Élimination de chaque supernoeud
 - 2.1 Factorisation du bloc diagonal dense
 - 2.2 Application d'un Solve sur les blocs LR
 - 2.3 Update LR sur des matrices LR ($LR2LR$ extend-add)
3. Résolution de systèmes triangulaires avec des blocs LR

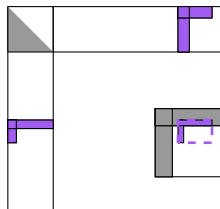
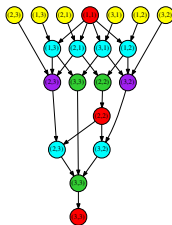
Yellow	Compression
Red	GETRF (Facto)
Cyan	TRSM (Solve)
Purple	LR2LR (Update)
Green	LR2GE (Update)



Stratégie *Minimal Memory*

1. Compression des gros blocs extra-diagonaux de A (en exploitant leur caractère creux)
2. Élimination de chaque supernoeud
 - 2.1 Factorisation du bloc diagonal dense
 - 2.2 Application d'un Solve sur les blocs LR
 - 2.3 Update LR sur des matrices LR ($LR2LR$ extend-add)
3. Résolution de systèmes triangulaires avec des blocs LR

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



Comparaison des deux stratégies

Consommation mémoire

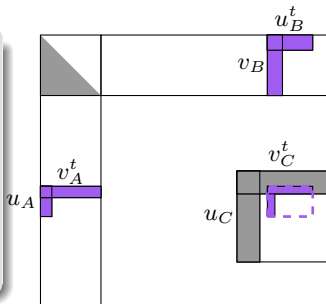
- La stratégie *Minimal Memory* n'alloue jamais les facteurs denses
- La stratégie *Just-In-Time* réduit la taille finale des facteurs mais utilise de manière intermédiaire les facteurs sous leur stockage dense

Procédure d'Update

- La stratégie *Minimal Memory* implique des recompressions coûteuses réalisées avec le noyau *LR2LR*
- La stratégie *Just-In-Time* continue d'appliquer des mises à jour d'une matrice dense à moindre coût via le noyau *LR2GE*

Focus sur le noyau LR2LR

- Mise à jour de la matrice C avec la contribution du couple de blocs (A, B)
- On forme une matrice $u_{AB}v_{AB}^t$ qui sera ajoutée à $u_Cv_C^t$
- $u_{AB}v_{AB}^t = (u_A(v_A^t v_B))u_B^t$ ou $u_{AB}v_{AB}^t = u_A((v_A^t v_B)u_B^t)$



Noyau LR2LR utilisant la SVD

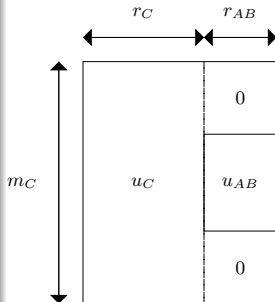
La structure LR $u_C v_C^t$ reçoit la contribution $u_{AB} v_{AB}^t$.

Algorithme

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

- 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: $R_1 R_2^t = u \sigma v^T \quad \Theta((r_C + r_{AB})^3)$

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



Avec RRQR la complexité est réduite à $\Theta(n(r_C + r_{AB})r_C^*)$: on arrête les calculs dès que le rang est trouvé

3

Expériences

Contexte expérimental

Machine du supercalculateur Plafrim

- 2 INTEL Xeon E5 – 2680v3 at 2.50 GHz
- 128 Go
- 24 coeurs

Matrices 3D issues de The SuiteSparse Matrix Collection

- *Audi*: mécanique des structures (943 695 dofs)
- *Atmosmodj*: modèle atmosphérique (1 270 432 dofs)
- *Geo1438*: modèle de la terre (1 437 960 dofs)
- *Hook*: modèle d'un crochet (1 498 023 dofs)
- *Serena*: simulation d'un réservoir de gaz (1 391 349 dofs)
- + Laplaciens: problème de Poisson (stencil à 7 points)

Le parallélisme est obtenu en suivant l'ordonnancement statique de PASTIX pour les architectures multi-threadées.

Configuration du solveur

Paramètres d'entrée

- Tolérance τ : valeur absolue normalisée pour chaque bloc
- Méthodes de compression: SVD ou RRQR
- Stratégie suivie: *Minimal Memory* ou *Just-In-Time*
- Taille de blocage: les blocs volumineux sont découpés en blocs de taille comprise entre 128 et 256

Stratégie *Minimal Memory*

- Les blocs sont compressés au début
- Chaque contribution est source d'une recompression

Stratégie *Just-In-Time*

- Les blocs sont compressés juste avant qu'un supernoeud soit éliminé
- Ces blocs ne sont jamais décompressés

Coûts opératoires sur la matrice Atmosmodj avec $\tau = 10^{-8}$

	Dense	<i>Just-In-Time</i>		<i>Minimal Memory</i>	
		RRQR	SVD	RRQR	SVD
Temps de factorisation (s)					
Compression	-	49.53	418.5	15.20	180.9
Factorization (GETRF)	0.9635	1.000	1.003	1.074	1.104
Solve (TRSM)	15.80	6.970	6.526	11.16	6.946
Update					
Produit LR	-	64.10	91.15	193.1	94.36
Addition LR	-	-	-	774.6	6523
Udpate dense (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>
Temps de solve (s)	2.43	1.54	1.8	2.22	1.29
Taille finale des facteurs (Go)	15.9	7.4	6.86	11.4	6.76
Pic mémoire (Go)	15.9	15.9	15.9	11.4	6.76

Coûts opératoires sur la matrice Atmosmodj avec $\tau = 10^{-8}$

	Dense	<i>Just-In-Time</i>		<i>Minimal Memory</i>	
		RRQR	SVD	RRQR	SVD
Temps de factorisation (s)					
Compression	-	49.53	418.5	15.20	180.9
Factorization (GETRF)	0.9635	1.000	1.003	1.074	1.104
Solve (TRSM)	15.80	6.970	6.526	11.16	6.946
Update					
Produit LR	-	64.10	91.15	193.1	94.36
Addition LR	-	-	-	774.6	6523
Udpate dense (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>
Temps de solve (s)	2.43	1.54	1.8	2.22	1.29
Taille finale des facteurs (Go)	15.9	7.4	6.86	11.4	6.76
Pic mémoire (Go)	15.9	15.9	15.9	11.4	6.76

Coûts opératoires sur la matrice Atmosmodj avec $\tau = 10^{-8}$

	Dense	<i>Just-In-Time</i>		<i>Minimal Memory</i>	
		RRQR	SVD	RRQR	SVD
Temps de factorisation (s)					
Compression	-	49.53	418.5	15.20	180.9
Factorization (GETRF)	0.9635	1.000	1.003	1.074	1.104
Solve (TRSM)	15.80	6.970	6.526	11.16	6.946
Update					
Produit LR	-	64.10	91.15	193.1	94.36
Addition LR	-	-	-	774.6	6523
Update dense (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>
Temps de solve (s)	2.43	1.54	1.8	2.22	1.29
Taille finale des facteurs (Go)	15.9	7.4	6.86	11.4	6.76
Pic mémoire (Go)	15.9	15.9	15.9	11.4	6.76

Coûts opératoires sur la matrice Atmosmodj avec $\tau = 10^{-8}$

	Dense	<i>Just-In-Time</i>		<i>Minimal Memory</i>	
		RRQR	SVD	RRQR	SVD
Temps de factorisation (s)					
Compression	-	49.53	418.5	15.20	180.9
Factorization (GETRF)	0.9635	1.000	1.003	1.074	1.104
Solve (TRSM)	15.80	6.970	6.526	11.16	6.946
Update					
Produit LR	-	64.10	91.15	193.1	94.36
Addition LR	-	-	-	774.6	6523
Udpate dense (GEMM)	418.7	47.94	47.03	-	-
Total	436	169	564	995	6806
Temps de solve (s)	2.43	1.54	1.8	2.22	1.29
Taille finale des facteurs (Go)	15.9	7.4	6.86	11.4	6.76
Pic mémoire (Go)	15.9	15.9	15.9	11.4	6.76

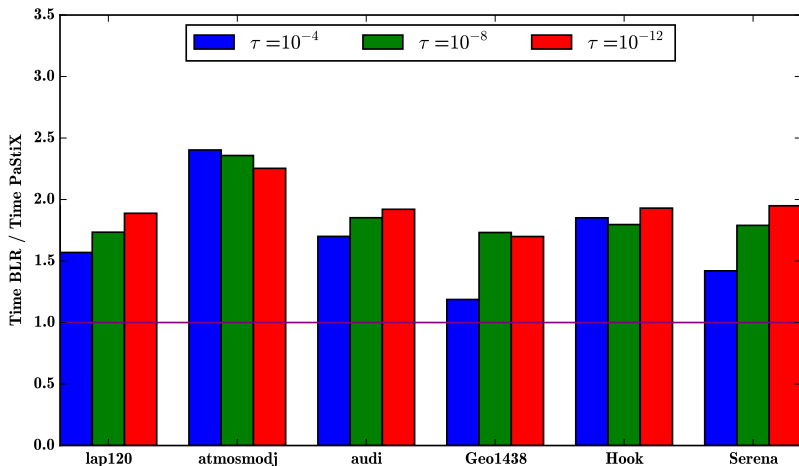
Coûts opératoires sur la matrice Atmosmodj avec $\tau = 10^{-8}$

	Dense	<i>Just-In-Time</i>		<i>Minimal Memory</i>	
		RRQR	SVD	RRQR	SVD
Temps de factorisation (s)					
Compression	-	49.53	418.5	15.20	180.9
Factorization (GETRF)	0.9635	1.000	1.003	1.074	1.104
Solve (TRSM)	15.80	6.970	6.526	11.16	6.946
Update					
Produit LR	-	64.10	91.15	193.1	94.36
Addition LR	-	-	-	774.6	6523
Udpate dense (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>
Temps de solve (s)	2.43	1.54	1.8	2.22	1.29
Taille finale des facteurs (Go)	15.9	7.4	6.86	11.4	6.76
Pic mémoire (Go)	15.9	15.9	15.9	11.4	6.76

Coûts opératoires sur la matrice Atmosmodj avec $\tau = 10^{-8}$

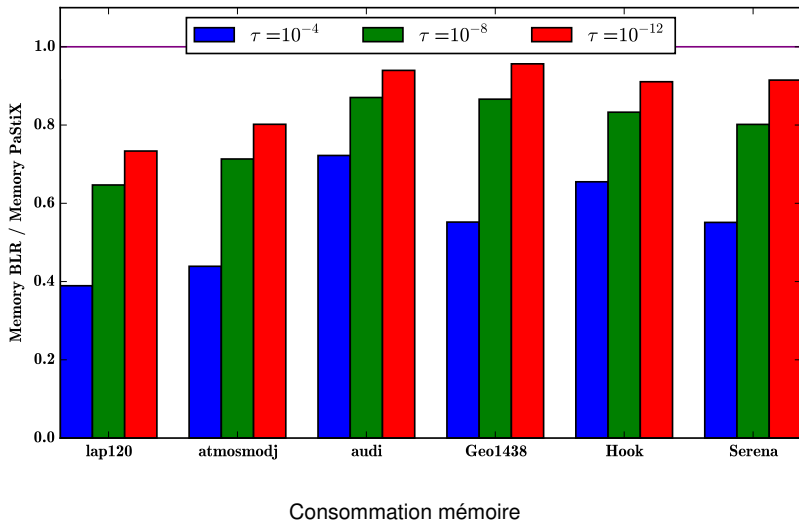
	Dense	<i>Just-In-Time</i>		<i>Minimal Memory</i>	
		RRQR	SVD	RRQR	SVD
Temps de factorisation (s)					
Compression	-	49.53	418.5	15.20	180.9
Factorization (GETRF)	0.9635	1.000	1.003	1.074	1.104
Solve (TRSM)	15.80	6.970	6.526	11.16	6.946
Update					
Produit LR	-	64.10	91.15	193.1	94.36
Addition LR	-	-	-	774.6	6523
Update dense (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>
Temps de solve (s)	2.43	1.54	1.8	2.22	1.29
Taille finale des facteurs (Go)	15.9	7.4	6.86	11.4	6.76
Pic mémoire (Go)	15.9	15.9	15.9	11.4	6.76

Comportement de RRQR/*Minimal Memory*: Performance

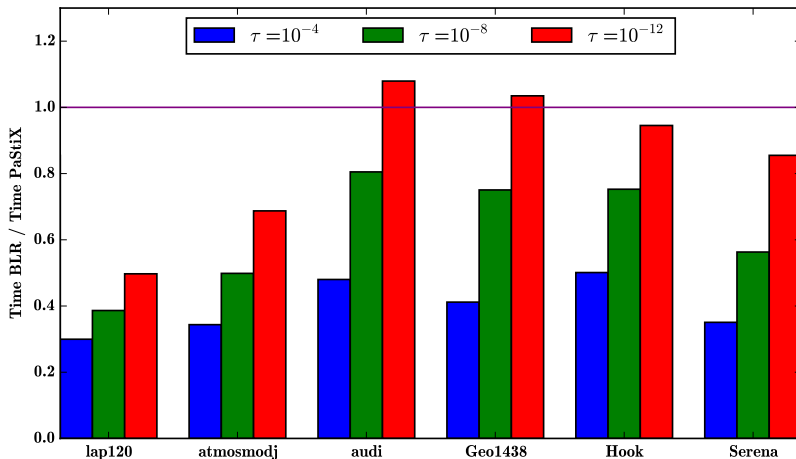


Performance de la factorisation

Comportement of RRQR/*Minimal Memory*: Mémoire

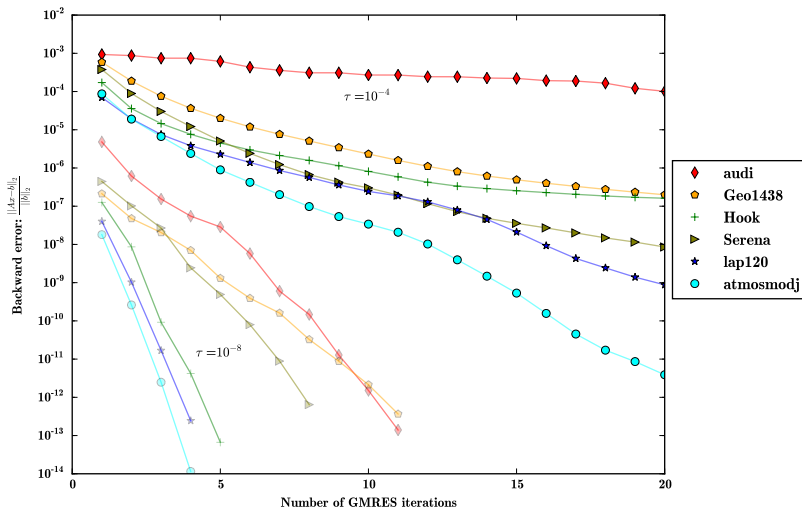


Performance de RRQR/*Just-In-Time*

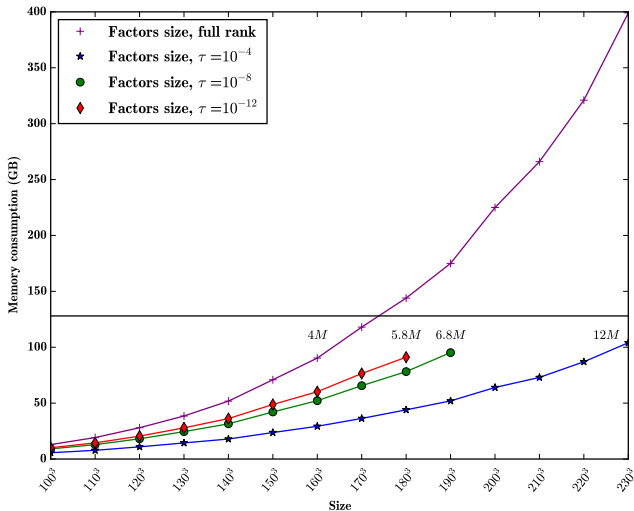


Performance de la factorisation

Convergence de RRQR/Minimal Memory



Scalabilité sur des Laplaciens: Consommation Mémoire de RRQR/*Minimal Memory*



Conclusion

Solveur Block Low-Rank

- Le parallélisme de PASTIX est conservé
- Principale différence par rapport aux concurrents due à l'approche supernodale, qui permet de réduire au maximum la consommation mémoire

Travaux futurs

- Mise en place d'une stratégie hybride pour allier les avantages de *Minimal Memory* et *Just-In-Time*
- Étude de techniques de renumérotation (dissection emboîtée) pour améliorer les taux de compression
- Introduction de nouveaux noyaux de compression
- Mise en place d'une version distribuée

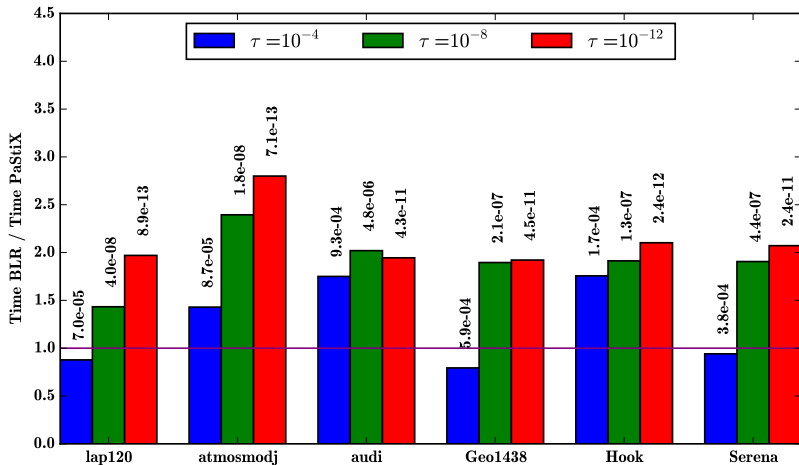
PASTIX 6.0.0alpha est désormais disponible!

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

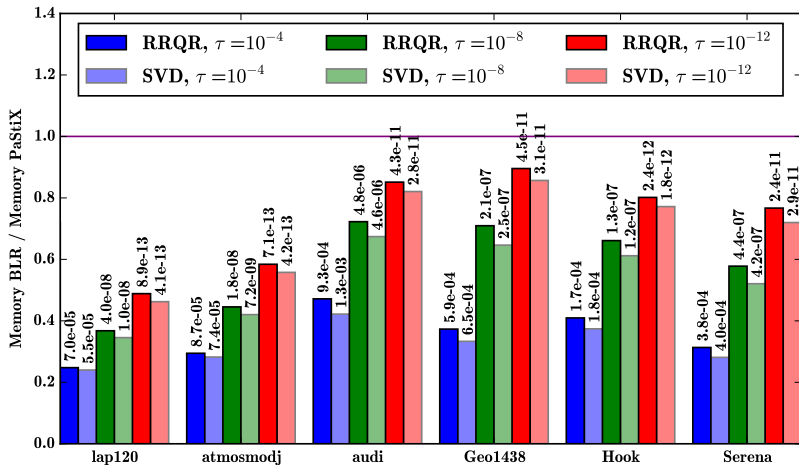
- Support des architectures en mémoire partagée avec différents ordonnanceurs:
 - ▶ séquentiel
 - ▶ ordonnancement statique
 - ▶ support d'exécution PaRSEC (StarPU en cours de finalisation)
- Support Low-rank
- Factorisation Cholesky et LU

Merci.

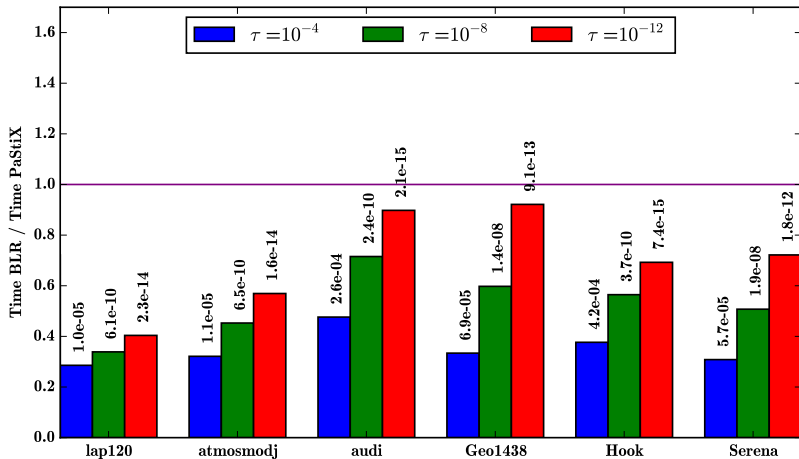
Performance of RRQR/*Minimal Memory*



Memory footprint of RRQR/*Minimal Memory*



Performance of RRQR/*Just-In-Time*



Compression techniques

Compressing A

- Absolute tolerance: tol
- Norm of the block being compressed: $\|A\|_2$
- Relative tolerance: $tol_A = \sqrt{tol} \times \|A\|_2$

Truncation criteria

- SVD: $A = u\sigma v^t$. Keep k singular values such that $\sigma_{k+1} < tol_A$
- RRQR: $A = Q_k R_k$. Stop when $\|\tilde{A}(k+1, :)\|_2 < tol_A$

On exit, we are considering the backward-error: $\frac{\|Ax-b\|_2}{\|b\|_2}$

Extend-add: SVD Recompression

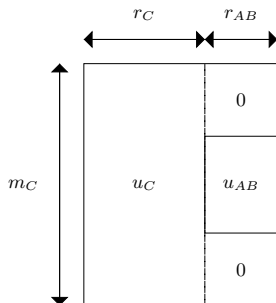
A low-rank structure $u_1v_1^t$ receives a low-rank contribution $u_2v_2^t$.

Algorithm

$$A = u_1v_1^t + u_2v_2^t = ([u_1, u_2]) \times ([v_1, v_2])^t$$

- QR: $[u_1, u_2] = Q_1R_1$ $\Theta(m(r_1 + r_2)^2)$
- QR: $[v_1, v_2] = Q_2R_2$ $\Theta(n(r_1 + r_2)^2)$
- SVD: $R_1R_2^t = u\sigma v^T$ $\Theta((r_1 + r_2)^3)$

$$A = (Q_1u\sigma) \times (Q_2v)^t$$



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

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

Orthogonalize u_2 with respect to u_1 :

$$u_2^* = u_2 - u_1(u_1^t u_2) \quad \Theta(mr_1 r_2)$$

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 :

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

RRQR with truncation in $\Theta(n(r_1 + r_2)r_1^*)$. Less stable?

Extend-add with algebraic methods for HODLR

BEGIN scenario

- Recompression is expensive
- The largest off-diagonal block is of size $\Theta(n^{\frac{2}{3}})$ and rank $\Theta(n^{\frac{1}{3}})$: each recompression requires $\Theta(n^{\frac{4}{3}})$ operations
- This operation has to be realized many times

END scenario

- Allocating the largest off-diagonal block in dense format is asymptotically as expensive as the full dense solver

How to use algebraic methods to save both time and memory?

Extend-Add in other software (cf F.H. Rouet talk at SIAM PP'16)

Strategies

- MUMPS-BLR [Amestoy et al., 2012]: contribution blocks not compressed
- HSS [Jia et al., 2009]: HSS extend-add for 2D problems, slow, hard to parallelize, hard to extend to algebraic
- HSS [Wang et al., 2015]: geometric, contribution blocks not compressed
- HSS [Ghysels et al., 2015]: algebraic code with randomized sampling