Trading Performance for Memory in Sparse Direct Solvers using Low-Rank Compression

Loris Marchal, Thibault Marette, Grégoire Pichon and Frédéric Vivien

ROMA team, LIP, Lyon

Sparse Days, 24 November 2020

Introduction

Sparse direct solvers

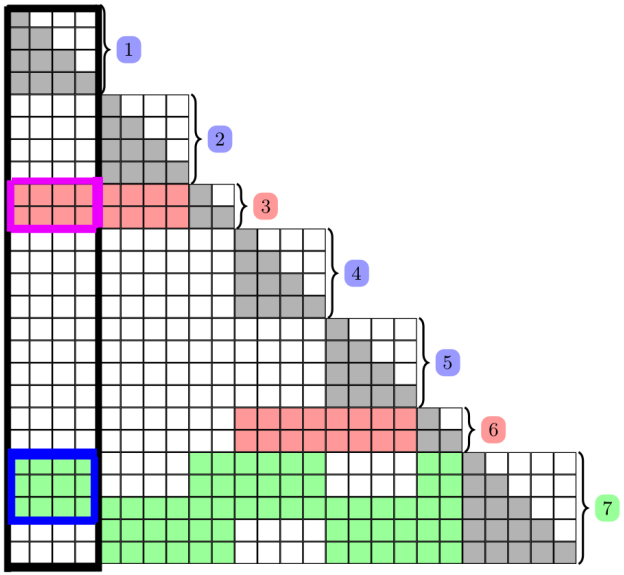
- High time and memory complexities
- Can solve systems made of millions of unknowns on top of distributed heterogeneous architectures

Low-rank compression

- Solve the problem at a reduced precision
- Can favor memory consumption (everything is compressed as soon as possible)
- Or execution time (using temporary full-rank memory spaces)

Symbolic factorization

Ordering performed with for instance Nested Dissection to minimize the fill-in



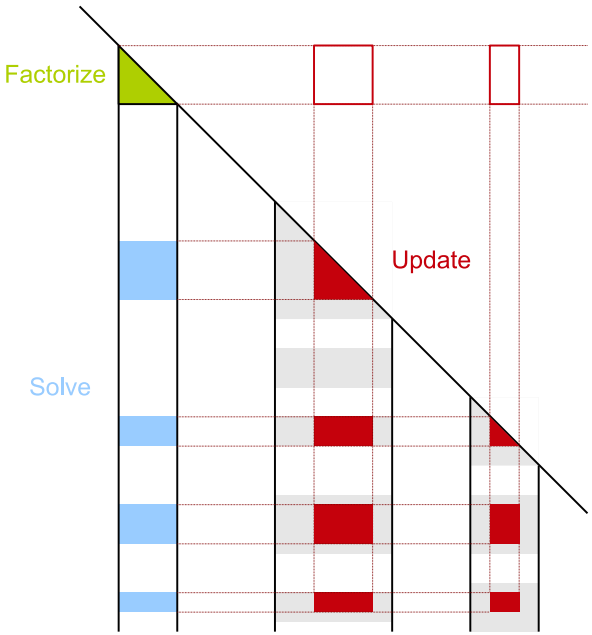
Matrix organisation after symbolic factorization

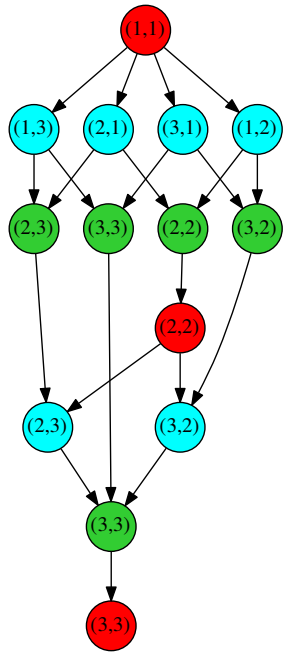
- The matrix is split into supernodes
- The grey blocks are called diagonal blocks.
- The blocks that are not diagonal blocks are called off-diagonal blocks.

Numerical factorization

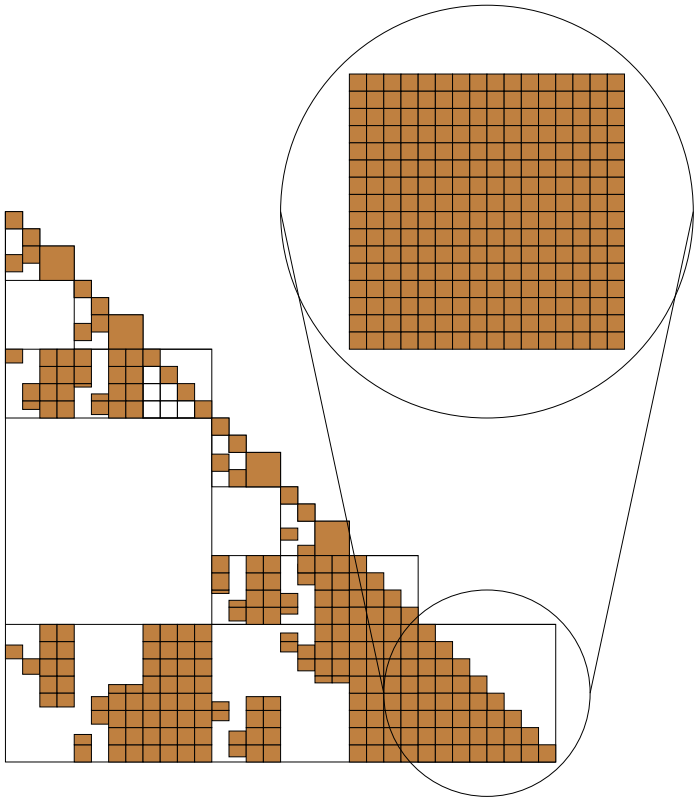
Algorithm to eliminate the k^{th} column block

1. Factorize the diagonal block
2. Solve off-diagonal blocks in the current column block
3. Update the trailing matrix with the column block contribution

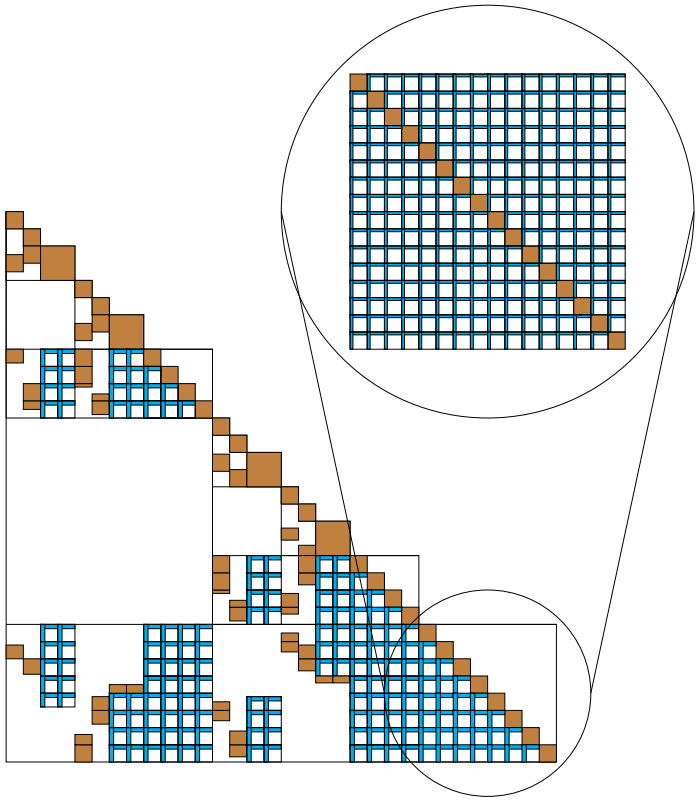




BLR compression – Symbolic factorization



Full-rank matrix.

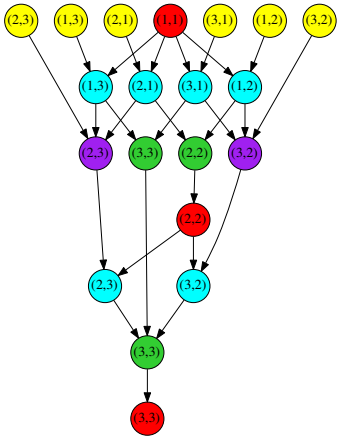


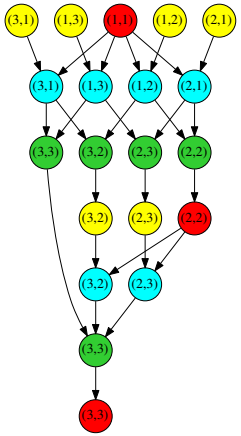
Block Low-Rank matrix.

Matrix compression

A low-rank compression of quality ε of any matrix $A \in \mathbb{R}^{m \times n}$ is a couple of two matrices $U \in \mathbb{R}^{m \times r_\varepsilon}$ and $V \in \mathbb{R}^{n \times r_\varepsilon}$ such that $||A - UV^T|| \leq \varepsilon ||A||$.

MM vs JIT: DAG of tasks





Minimal Memory

- Compress all off-diagonal blocks before starting the factorization
- Update low-rank blocks

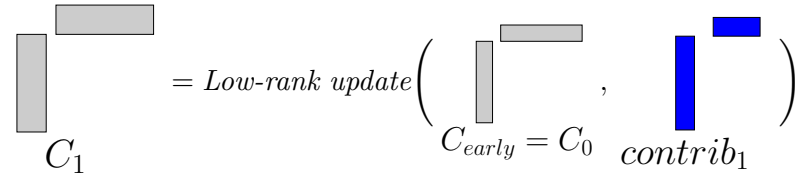
	Compression
	Factorize
	Solve
	Low-rank update
	Dense update

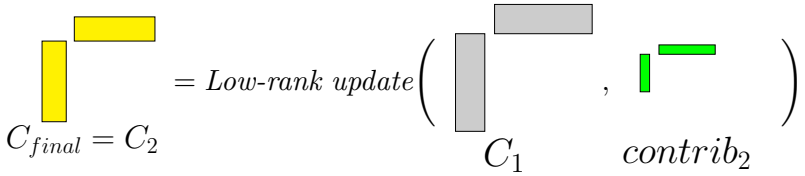
Just-In-Time

- Compress each block when fully updated
- Update full-rank blocks

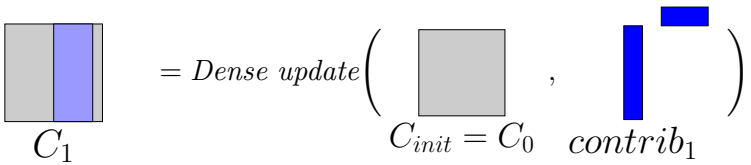
MM vs JIT: updates with two contributions

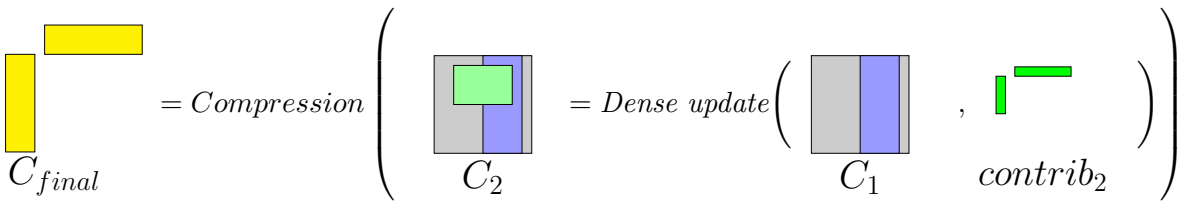
Low-rank updates (*Minimal Memory*):





Full-rank updates (*Just-In-Time*):





Both strategies have the same contributions as **inputs** and the same final low-rank matrix as **output**

Limitations

Minimal Memory

- Never use the blocks in their full-rank form: consumes as little memory as possible
- Expensive low-rank updates to maintain low-rank structures

Just-In-Time

- Efficient updates
- Compress blocks during the factorization: more memory consuming

The objective of this talk is to propose a **memory-aware** strategy that uses as much memory as possible to speedup updates while remaining under a memory constraint.

Modelization

Each block can be considered independently

Idea: two possible modes

- *early* mode (as in the *Minimal Memory* strategy): execution time T_i (sum of the updates) and memory $s_i = r_i \times (m_i + n_i)$.
- *lazy* mode (as in the *Just-In-Time* strategy): execution time t_i (sum of the updates) and memory $S_i = m_i \times n_i$;
- Execute a set of blocks in *early* mode to respect the **memory constraint**
- Execute other blocks in *lazy* mode to perform **efficient operations**

Algorithm

- For a given memory constraint, choose the sets for being as fast as possible
- This algorithm is equivalent to Knapsack: we inherit its NP-hardness and all approximation algorithms with the same approximation factor
- Sort blocks in a greedy approach (2-approximation) accordingly to $\frac{T_i - t_i}{S_i - s_i}$

Models to estimate T_i, t_i and s_i

Issue

- The mode of each block has to be chosen before starting the factorization
- Time and memory for each mode depend on the rank
- The rank depends on numerical properties: cannot be known in advance

Memory consumption model

- We made a linear regression for the rank ($s_i = r_i \times (m_i + n_i)$)
- 1) the initial rank, 2) the height m_i, 3) the width n_i, 4) the surface $m_i n_i$ and 5) the number of updates the block receives.

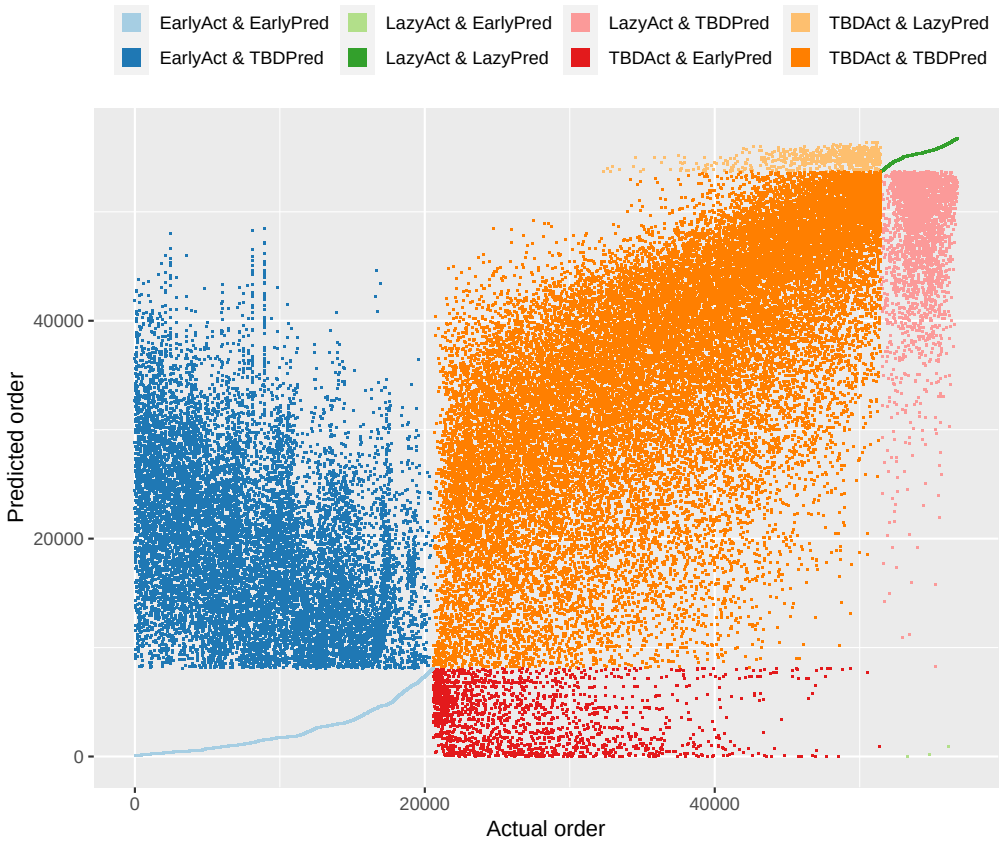
Time model: sum of update's time

- We made a linear regression with the different parameters, knowing the theoretical complexity of an update
- When a rank appears, we use the five parameters given above instead

Results: actual vs predicted orders of blocks

Three categories

- Blocks that are always better in *early* mode ($T_i \leq t_i$)
- Blocks blocks to treat with Knapsack, sorted by $\frac{T_i - t_i}{S_i - s_i}$
- Blocks that are always better in *lazy* mode ($S_i \leq s_i$)



Conclusions

- Training with one matrix and testing with another
- General trend
- Imprecise

Experimental context

Solver / machine

- PaStiX, used in sequential
- Intel Xeon E5-4620, using MKL 2018

Matrices

- Geo1438: geomechanical model of earth (1 437 960 non-zeroes)
- Hook1498: model of a steel hook (1 498 023 non-zeroes)
- Serena: gas reservoir simulation (1 391 349 non-zeroes)

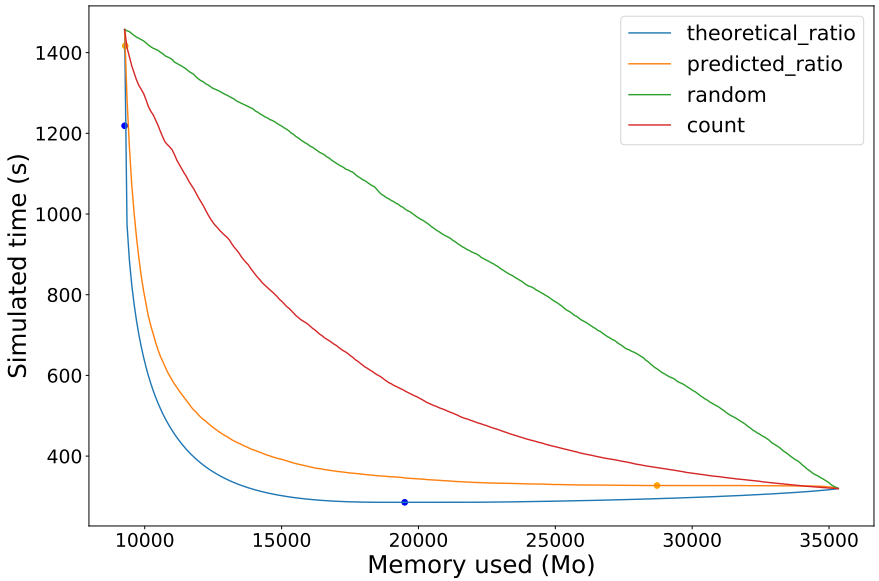
Full algorithm for the *memory-aware* strategy

1. Run the factorization using *Just-In-Time* and *Minimal Memory* strategies for the training matrix and **train** the time and the memory models
2. Use the models for the **test** matrix
3. **Select** blocks that should always be treated in ***lazy* mode** ($s_i \geq S_i$) as well as blocks that should always be treated in ***early* mode** ($t_i \geq T_i$);
4. **Sort** the remaining blocks by decreasing value of $\frac{T_i - t_i}{S_i - s_i}$;
5. Choose a sufficient number of blocks (following the order) to perform in *early* mode in order to **respect the memory constraint** and keep remaining blocks in *lazy* mode in order to **perform efficient updates**

Simulation, training with Serena, testing with Geo1438, $tol = 10^{-8}$

Ratios depicted

- Decreasing **theoretical ratio** $\frac{T_i - t_i}{S_i - s_i}$
- Decreasing **predicted ratio** $\frac{T_i^* - t_i^*}{S_i - s_i^*}$
- Decreasing number of updates (**count**) received by a block
- **Random** order, for baseline comparison.



Conclusions

- Excellent trade-off between time and memory
- Much better than a naive approach
- Close to the best solution, knowing perfectly all information

Results on real execution (training with Serena, $tol = 10^{-8}$)

Matrix	Strategy	Memory (GB)	Time(s) With pred (s)	Time(s) Opt time (s)
Geo1438	*Just-In-Time*	43.2	555.9	
	Minimal Memory	14.7	1591.7	
	memory-aware	minimum	1190.2	1149.1
		19	724.1	647.0
		23	663.3	576.0
		27	618.6	556.5
		∞	578.3	553.9
Hook1498	*Just-In-Time*	27.2	407.3	
	Minimal Memory	11.8	1863.7	
	memory-aware	minimum	1056.1	991.5
		16	506.4	465.3
		20	431.9	417.0
		24	416.5	410.0
		∞	415.5	412.3
Serena	*Just-In-Time*	46.7	534.2	
	Minimal Memory	13.3	1876.2	
	memory-aware	minimum	1300.5	1270.9
		18	654.0	606.0
		22	579.1	543.5
		26	552.7	529.1
		∞	539.8	527.1

Implementation

- Blocks are first sorted
- Dynamic memory controller
- Memory can increase due to rank growth
- Memory can decrease when blocks are compressed

Conclusion

A *memory-aware* strategy

- Proof of concept for the sequential case
- Implemented into the PaStiX solver
- Allow to reach the best of both worlds ?
- Interesting trade-offs: with 30% extra memory, divide time by 3

Future work

- Parallel experiments with a parallel memory controller
- Consider the critical path to better choose the mode of each block

See our research report: https://hal.inria.fr/hal-02976233