Trade-offs between execution time and memory consumption when using low-rank compression

Loris Marchal, Thibault Marette, Grégoire Pichon, & Frédéric Vivien

CNRS, INRIA, UCBL & ENS Lyon

TOPAL WG: February, 3rd 2022
`https://hal.inria.fr/hal-03517124/document`

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

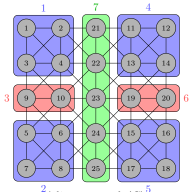
- Solve the problem at a reduced precision
- Can favor memory consumption (everything is compressed as soon as possible)
- Or execution time (using temporary full-rank memory spaces)

Outline

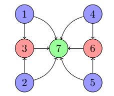
Block Symbolic Factorization

General approach

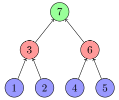
1. Build a partition with the nested dissection process
2. Compress information on data blocks
3. Compute the block elimination tree using the block quotient graph



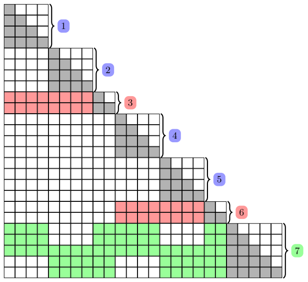
Adjacency graph (G).



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



Elimination tree (T).

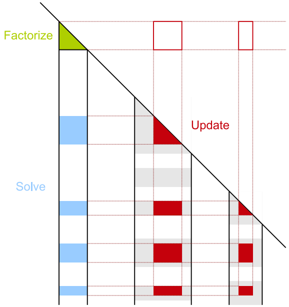


Factorized matrix (L).

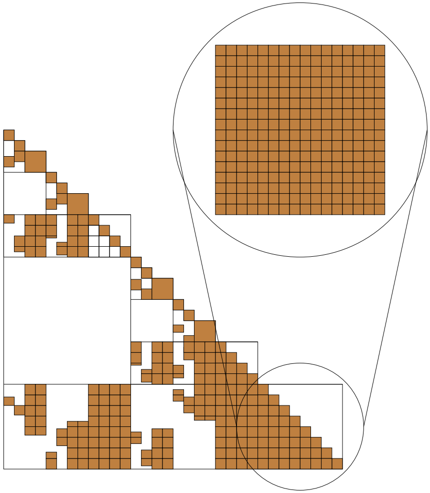
Block Numerical Factorization

Algorithm to eliminate the k^{th} supernode

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



BLR compression – Symbolic factorization



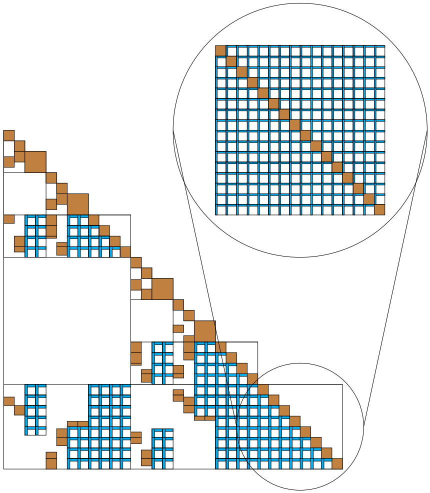
Approach

- 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



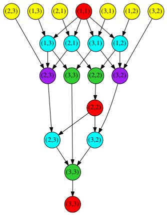
Approach

- 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

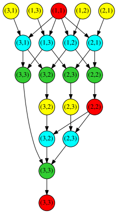
MM vs JIT: DAG of tasks



Minimal Memory

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

🟨	*Compression*
🟥	*Factorize*
🟦	*Solve*
🟪	*Low-rank update*
🟩	*Dense update*

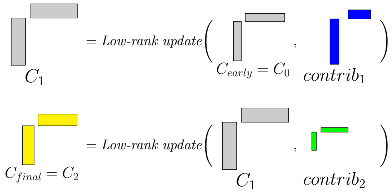


Just-In-Time

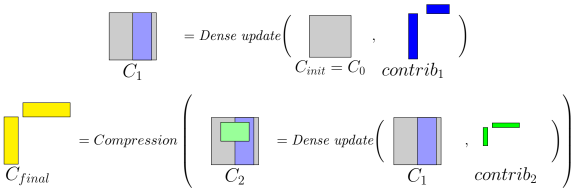
- Compress each block when fully updated
- Update full-rank blocks

MM vs JIT: updates with two contributions

Low-rank updates (*Minimal Memory*):



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


C_1
$= \textit{Dense update}\Big($
$C_{init} = C_0$
$contrib_1$
$\Big)$
C_{final}
$= \textit{Compression}\Big($
C_2
$= \textit{Dense update}\Big($
C_1
$contrib_2$
$\Big)\Big)$

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

Outline

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**

General approach

Algorithm

- For a given memory constraint $\mathcal{M}$, 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 $\frac{T_i - t_i}{S_i - s_i}$

Assumptions

- $i \in [1 : n], S_i > s_i$ and $T_i > t_i$
- Otherwise, if $S_i \leq s_i$ it is always better to execute the task in *lazy* mode and if $T_i \leq t_i$ it is always better to use the *early* mode.

Equivalence with Knapsack (1/2)

Knapsack problem

Let $\mathcal{I}$ be a set of n items. Each item has a value v_i and a weight w_i. The objective is to fit some of the items in a bag of weight capacity $\mathcal{W}$, while maximizing the value of the objects inside the bag.

We associate a variable $x_i \in \{0, 1\}$ to each $J_i \in [1 : n]$.
Let

- $x_i = 1$ if the task J_i is executed in *lazy* mode,
- $x_i = 0$ if the task J_i is executed in *early* mode.

Therefore, the ILP formulation is:

$$\text{minimize} \quad \sum_{i=1}^{n}(x_i t_i) + \sum_{i=1}^{n}((1 - x_i)T_i) \tag{1}$$

$$\text{subject to} \quad \sum_{i=1}^{n}(x_i S_i) + \sum_{i=1}^{n}((1 - x_i)s_i) \leq \mathcal{M} \tag{2}$$

$$\text{and} \quad \forall i \in \{1, n\},\ x_i \in \{0, 1\} \tag{3}$$

Equivalence with Knapsack (2/2)

We have the following relations:

$$(1) \iff \textit{maximize} \sum_{i=1}^{n} x_i(T_i - t_i) - \sum_{i=1}^{n} T_i \iff \textit{maximize} \sum_{i=1}^{n} x_i(T_i - t_i)$$

$$(2) \iff \sum_{i=1}^{n} x_i(S_i - s_i) \leq \mathcal{M} - \sum_{i=1}^{n} s_i$$

Thanks to these two equivalences, we just showed that it is exactly a linear formulation of the Knapsack problem:

$$\text{maximize} \quad \sum_{i=1}^{n} x_i v_i \quad \text{subject to} \quad \sum_{i=1}^{n} x_i w_i \leq \mathcal{W} \quad \text{and} \quad \forall i \in \{1, n\}, x_i \in \{0, 1\}$$

with the following transformation:

- $\forall i \in [1:n], v_i = T_i - t_i$ and $\forall i \in [1:n], w_i = S_i - s_i$
- $\mathcal{W} = \mathcal{M} - \sum_{i=1}^{n} s_i$

Therefore, our problem is NP-complete

Approximation quality – theory

Algorithm 1 Greedy approximation algorithm

```
1: Sort tasks by non-increasing $\frac{T_i - t_i}{S_i - s_i}$ values
2: Greedily add tasks to a set $S$ while the sum of their weights $w_i = S_i - s_i$
   does not exceed $\mathcal{M} - \sum_{i=1}^{n} s_i$
```

Our algorithm is a $(1 + 2\varepsilon\rho)$-approximation of our problem

- $\varepsilon = \max_i S_i / (\mathcal{M} - \sum_j s_j)$ (ratio size largest blocks wrt remaining memory)
- $\rho = (\sum_i T_i)/(\sum_i t_i)$ (overhead of MM wrt JIT)

Approximation quality – in practice

$(1 + 2\varepsilon\rho)$-approximation of our problem

- $\varepsilon = \max_i S_i / (\mathcal{M} - \sum_j s_j)$
- $\rho = (\sum_i T_i)/(\sum_i t_i)$

Practical values leading to a 1.02-approximation

- $\rho \leq 10$, it corresponds to the ratio between the execution times of the *Minimal Memory* and *Just-In-Time* strategies
- $\varepsilon \leq 0.001$
  1. Block size is lower than 256 (splitting) thus $\max_i S_i \approx 0.5 \text{ MB}$
  2. Let us assume that $\mathcal{M} \geq 1.1 \times \sum_j s_j$ and that the overall memory is larger than 5GB
  3. $$\mathcal{M} - \sum_j s_j \geq 0.1 \sum_j s_j \geq 0.5GB$$

Models to estimate T_i, t_i and s_i (1/2)

The mode of each block has to be chosen before starting the factorization. Unfortunately, the time and the memory consumption depend on the rank of the matrix. The rank depends on numerical properties and cannot be known before the factorization.

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

Models to estimate T_i, t_i and s_i (2/2)

Memory consumption model

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

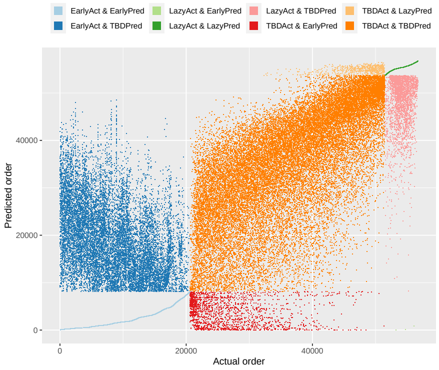
Time model: sum of update's time

- 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

Outline

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)

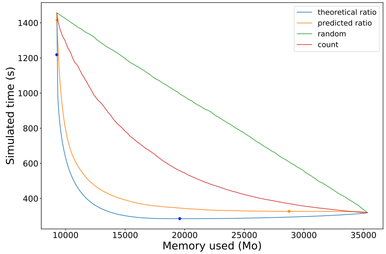
Full algorithm for the *memory-aware* strategy

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**
6. If the memory increases during the factorization and would exceed the memory limit, we compress the next block in the previous order (we switch this block to *early* mode). This is required for actual runs of the solver as we cannot know the exact evolution of the memory consumption of low-rank blocks before the actual factorization.

Simulation, train=Serena, test=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 (train 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

Open research

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