Deciding Non-Compressible Blocks in Sparse Direct Solvers using Incomplete Factorization

HiPC 2021

Esragul Korkmaz, Mathieu Faverge, Grégoire Pichon and Pierre Ramet

Inria, CNRS, Univ. Bordeaux, Univ. Lyon, Bordeaux INP, ENS Lyon

Context: Sparse direct solvers

Current status for 3D problems of size n^3

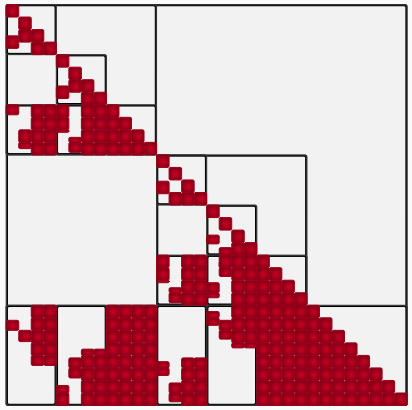
- $\Theta(n^2)$ time complexity
- $\Theta(n^{\frac{4}{3}})$ memory complexity
- BLAS Level 3 operations with computations on blocks

Exploit low-rank compression to reduce the cost

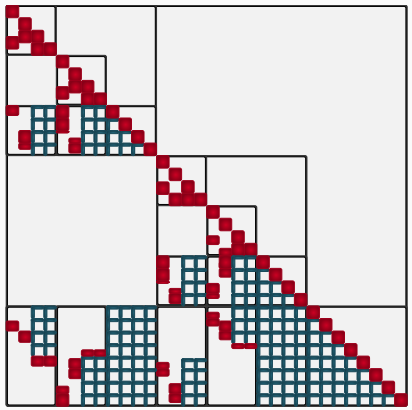
- Compress diagonal and/or off-diagonal blocks
- Use different compression formats: BLR, $\mathcal{H}$, HODLR, HSS, $\mathcal{H}^2$..
- Recently introduced in many solvers:
  MUMPS, PASTIX, STRUMPACK and many others

Block Low-Rank (BLR) compression scheme (in PaStiX)

Full rank:



Block low-rank:



Approach

- Keep the same structure as in full rank used to increase parallelism / block efficiency
- Diagonal blocks are kept dense
- Off-diagonal blocks are compressed wrt an admissibility criterion (block size)

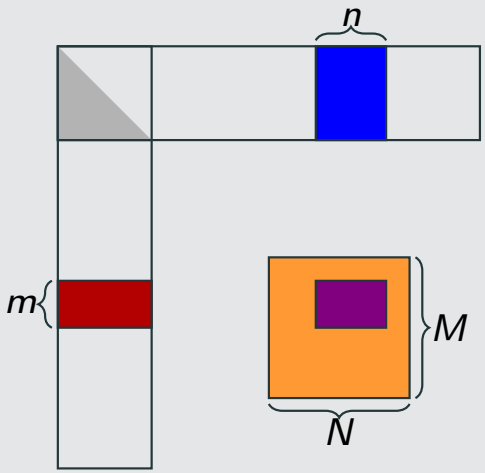
Operations

- TRSM are always performed on the **low-rank** form
- Two scenarios to apply the GEMM updates

Strategies to apply the updates

Cost of applying a $m \times n$ update to a $M \times N$ block.

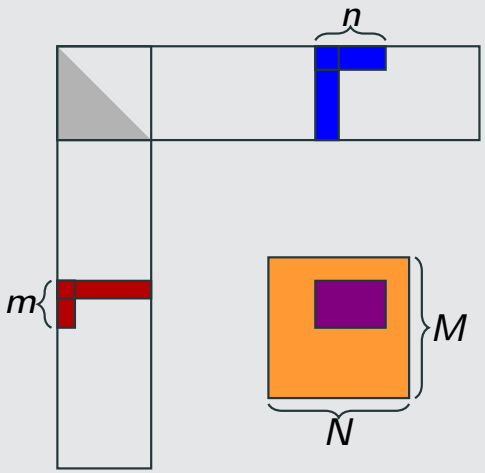
Full rank



- A single GEMM kernel
- Cost depends on mn

Non-fully structured

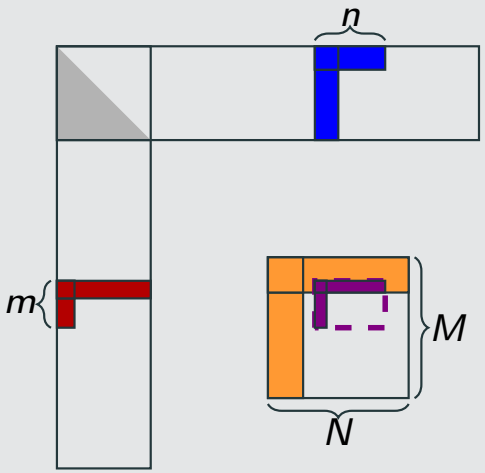
Just-In-Time



- Small GEMM kernels
- Cost depends on mn

Fully structured

Minimal Memory



- Complex operation
- Cost depends on MN

The two compression scenarios

Algorithm 1 Minimal Memory - Just in Time Scenarios

```
1: for each off-diagonal block A_ij in A do
2:     Compress(A_ij)                          /* Compress if admissible */
3: for k=1 to N_cblk do
4:     Factorize
5:     for each off-diagonal block A_ij in cblk do
6:         Compress(A_ij)                      /* Compress if admissible */
7:     Solve
8:     Update
```

Just in Time

- ALAP Compression
  - *After all Write*
  - *Before all Read tasks*
- Non fully structured updates

✗ Do not reduce the memory peak

✓ Reduce the flop count

✓ Reduce the time to solution

Minimal Memory

- ASAP Compression
  - *Before any other operation*
- Fully structured updates

✓ Reduce memory footprint

✗ May increase the flop count

✗ May increase the time to solution

Objective

Identify the high-rank blocks to reduce the fully structured updates overhead

Main ideas

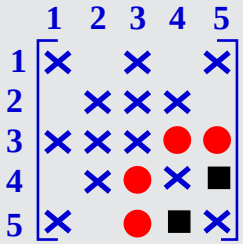
- Allow low memory overhead to save flops
- Do not attempt to compress blocks that end up full-rank
- Identify blocks with numerous updates of smaller sizes
- Identify blocks with few updates of similar sizes

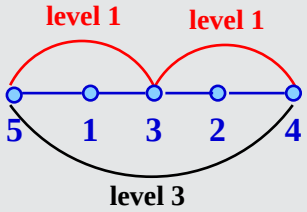
Proposed idea

Exploit the block Incomplete LU (ILU) factorization levels

Block ILU Factorization

Recalls on ILU





The levels of the fill-in values (blocks) in L somehow represent the *distance* to the original matrix A.

Easy and cheap heuristic to apply in a block-wise manner.

Low level blocks

- Have larger ranks
- Marginally improve memory footprint
- Receive many updates

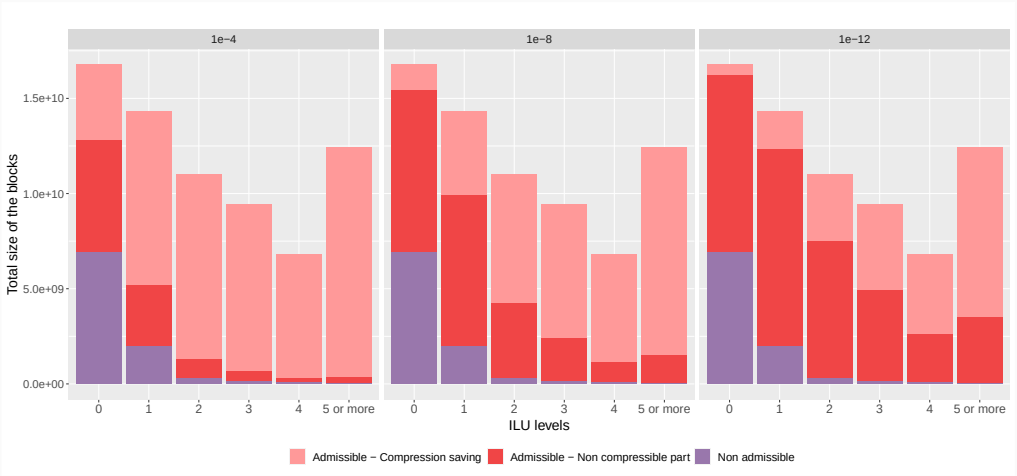
$\rightarrow$ Can be compressed ALAP

High level blocks

- Have smaller ranks
- Greatly improve memory footprint
- Receive few updates

$\rightarrow$ Can be compressed ASAP

Compression ratio wrt ILU block levels and tolerance



- Statistics on a set of 31 matrices from SuiteSparse Matrix Collection
- Dark red represents the memory used by the admissible low-rank blocks after compression
- Light red shows the amount of memory that can be saved by compressing the blocks
- Purple represents the memory used by the non-admissible blocks

- The higher the level, the higher the memory saving
- For a low tolerance criterion, good memory savings can be obtained considering only small levels of fill

Block ILU level definition

Algorithm 2 Symbolic ILU(*maxlevel*) Factorization

```
1:  /* Initialize the levels */
2:  for all L_ij in L do
3:      if A_ij ≠ 0 then
4:          lvl(L_ij) = 0
5:      else
6:          lvl(L_ij) = ∞
7:  /* Simulate the block factorization */
8:  for k = 1 to n do
9:      for i = k + 1 to n do
10:         for j = k + 1 to n do
11:             lvl(L_ij) = min(lvl(L_ij), lvl(L_ik) + lvl(L_kj) + 1)
```

BLR algorithm with the new ILU(*maxlevel*) heuristic

Algorithm 3 Minimal Memory - Just in Time - ILU(*maxlevel*) Scenarios

```
1:  for each off-diagonal block A_ij in A do
2:      if level(A_ij) > maxlevel then
3:          Compress(A_ij)                    /* Compress if admissible */
4:  for k=1 to N_cblk do
5:      Factorize
6:      for each off-diagonal block A_ij in cblk do
7:          if level(A_ij) <= maxlevel then
8:              Compress(A_ij)                /* Compress if admissible */
9:      Solve
10:     Update
```

maxlevel defines the criterion to switch from ASAP to ALAP compression according to the input matrix and tolerance

Experiments

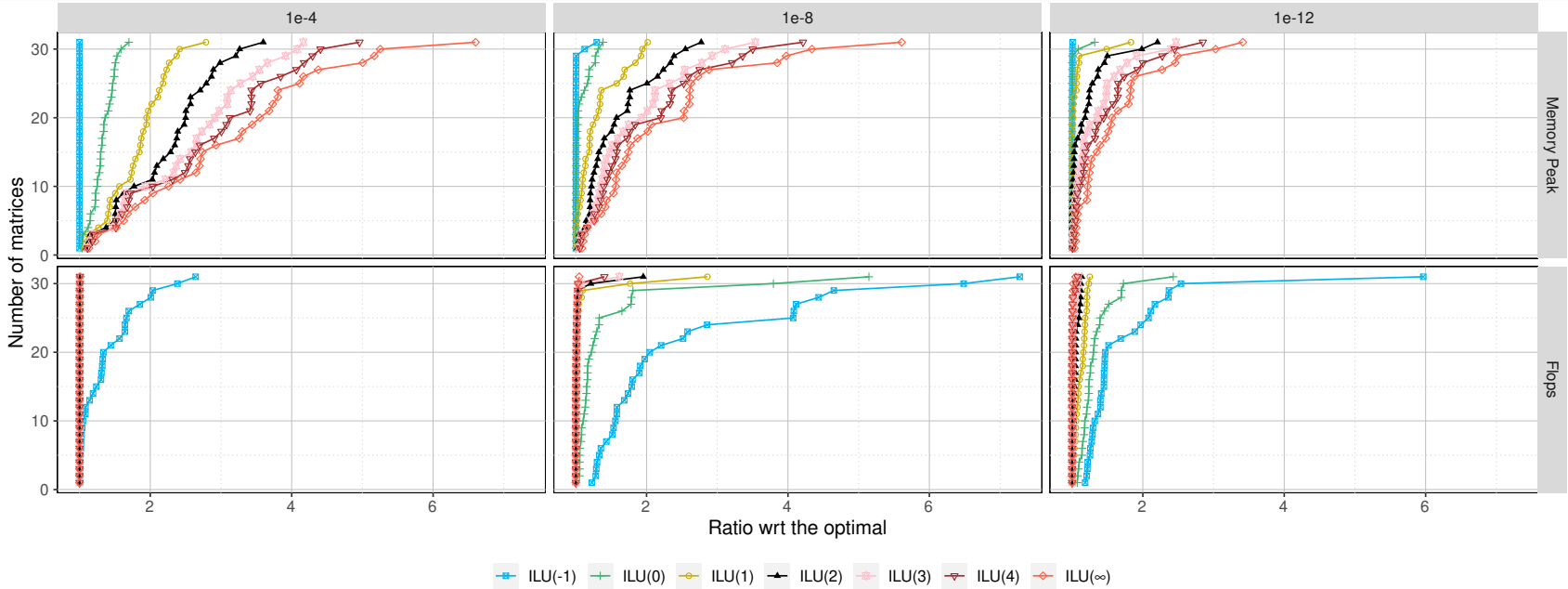
Context

- On a set of 31 matrices from the SuiteSparse Matrix Collection
- On a two INTEL Xeon E5-2680v3 12-cores running at 2.50 GHz and 128 GB of memory

Presentation of results

- Performance profile curve: each curve represents the number of matrices within a percentage overhead of the best solution for each metric and matrix
- We compare the existing approaches, *Minimal Memory* (or $ILU(-1)$) and *Just-In-Time* (or $ILU(\infty)$), with the new heuristic based on levels of fill

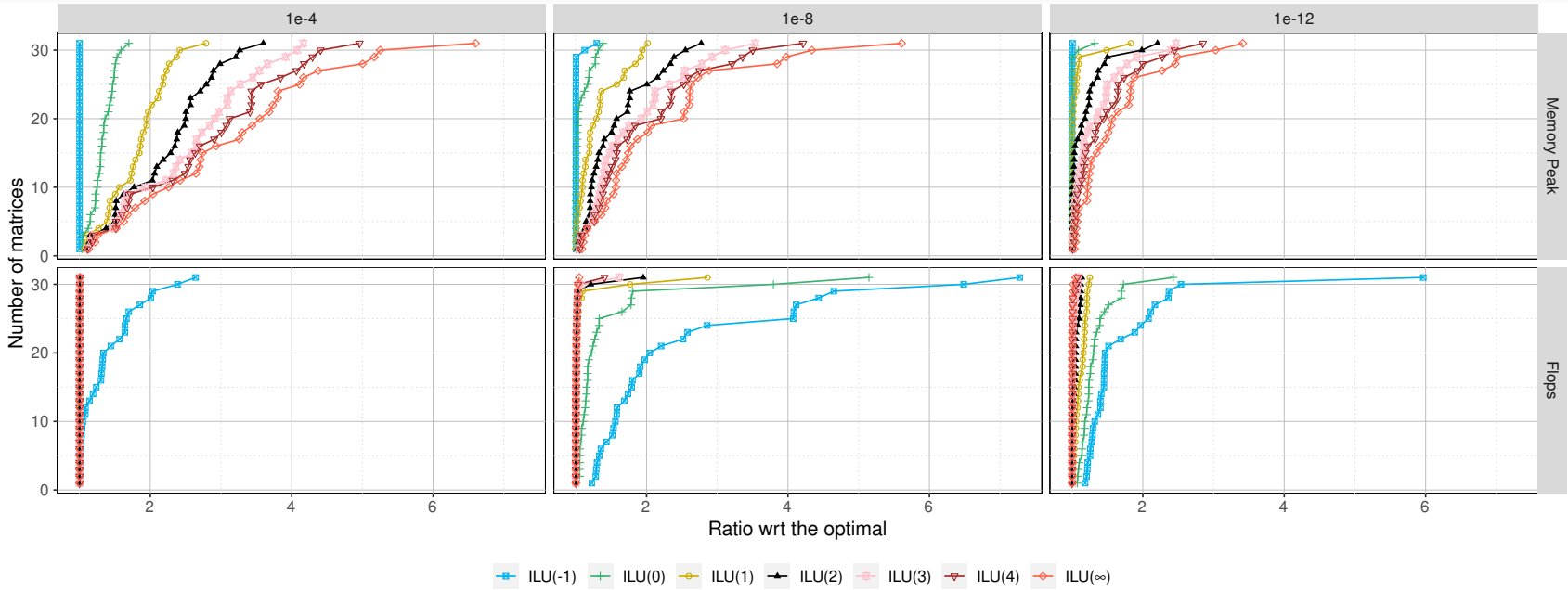
Impact of the ILU(k) Heuristic on the memory and flops



Memory consumption

- *Minimal Memory* reaches the smallest memory consumption, while *Just-In-Time* reaches the highest
- The lower the level, the higher the memory consumption
- *ILU*(1) reaches a good compromise in average

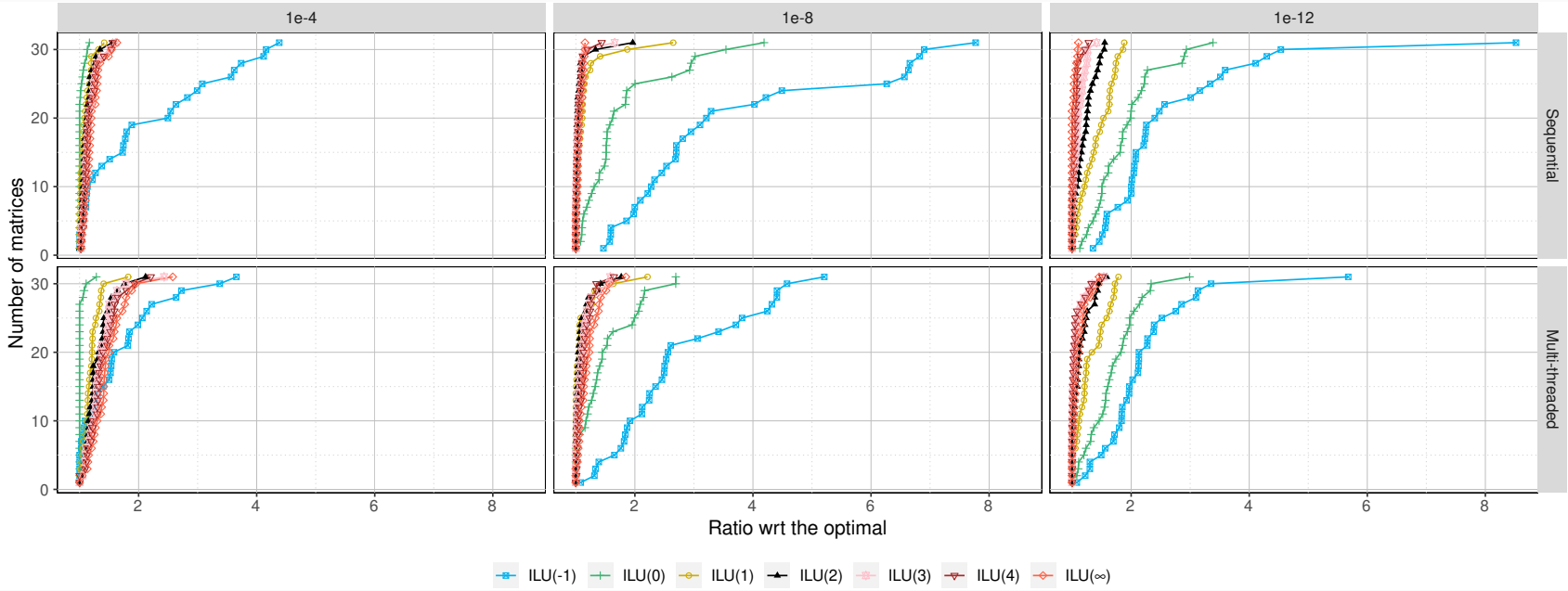
Impact of the ILU(k) Heuristic on the memory and flops



Flops

- *Just-In-Time* has the smallest flops count, while *Minimal Memory* has the highest
- The lower the level, the lower the flops count
- In many cases, the new heuristic is almost mingled with *Just-In-Time*

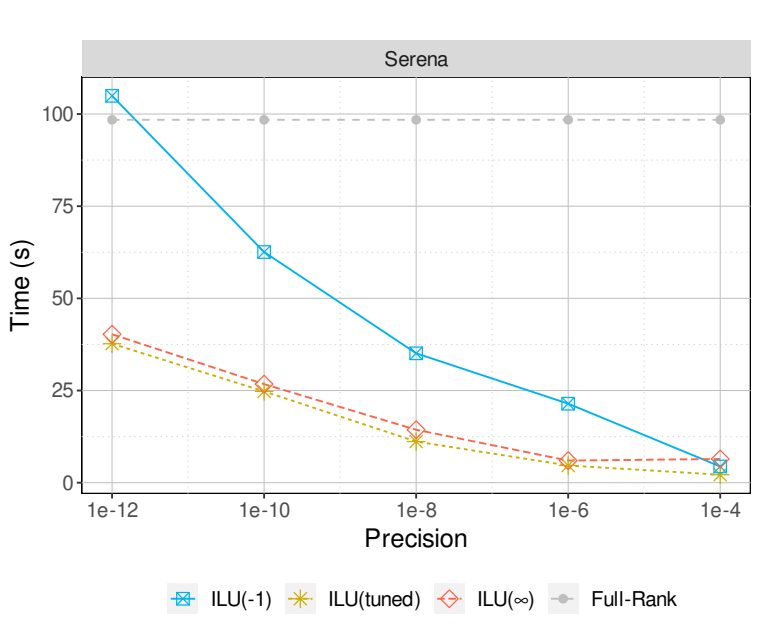
Impact of the ILU(k) Heuristic on the factorization time



Sequential vs multi-threaded

- The trend on the factorization time follows the trend on flops
- The ILU(k) heuristic behaves better than the *Just-In-Time* strategy in parallel: its execution time if often lower, especially for low tolerance

Tolerance scalability on the Serena Matrix



Tolerance	$ILU(tuned)$
$1e^{-4}$	$ILU(0)$
$1e^{-6}$	$ILU(2)$
$1e^{-8}$	$ILU(2)$
$1e^{-10}$	$ILU(4)$
$1e^{-12}$	$ILU(4)$

Table 1: Corresponding levels for $ILU(tuned)$ at each precision.

Observations

- By tuning correctly the level of fill for each tolerance, we can always outperform both the *Minimal Memory* and the *Just-In-Time* strategies
- Recall that when $ILU(k)$ is faster than *Just-In-Time*, it also reduces significantly its memory consumption

Conclusion

The ILU(k) heuristic

- Allows to reach good trade-offs between the *Minimal Memory* and the *Just-In-Time* strategies
- The new heuristic in sequential provides huge speedup and reduces flops with only a slight memory increase
- The new heuristic in parallel is even faster than *Just-In-Time*
- It is possible to tune the level of fill to always consume as low memory as *Minimal Memory* or to reduce execution time compared to *Just-In-Time*

Future work

- Infer automatically the level depending on the user's requirements
- Order unknowns to gather contributions with a similar level of fill