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

Sparse Days 2022

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 of sparse direct solvers on 3D problems

- $\Theta(n^2)$ time complexity
- $\Theta(n^{\frac{4}{3}})$ memory complexity

Low-rank representations to reduce these costs

- Different representations:
  - Non-hierarchical format like BLR
  - Hierarchical format like $\mathcal{H}$, HODLR, HSS, $\mathcal{H}^2$
- Recently introduced in many solvers:
  - MUMPS, PASTIX, STRUMPACK and many others

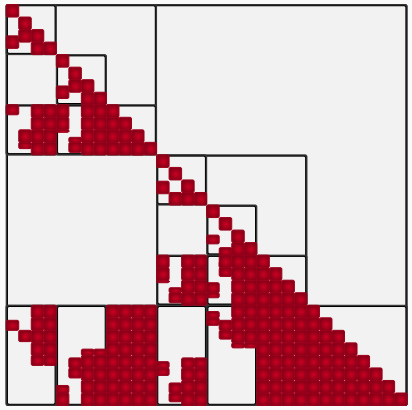
Sparse supernodal direct solver PaStiX

PaStiX 6.2.1 `http://gitlab.inria.fr/solverstack/pastix`

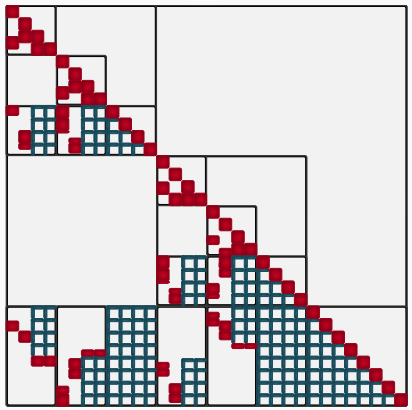
- Support shared/distributed memory with different schedulers:
  - Sequential
  - Static/Dynamic
  - PaRSEC/StarPU runtime systems with GPU support
- Cholesky, *LDL* and *LU* factorizations
- GMRES, CG, BiCG iterative refinements
- Wrappers:
  - F90/Python/Julia
  - PETSc (ongoing work)
- Non-hierarchical representation: block low-rank (BLR)

Block Low-Rank (BLR) representation (in PaStiX)

Full rank:



Block low-rank:



Approach

- Diagonal blocks are kept dense
- Off-diagonal blocks are compressed wrt an admissibility criterion (block size)

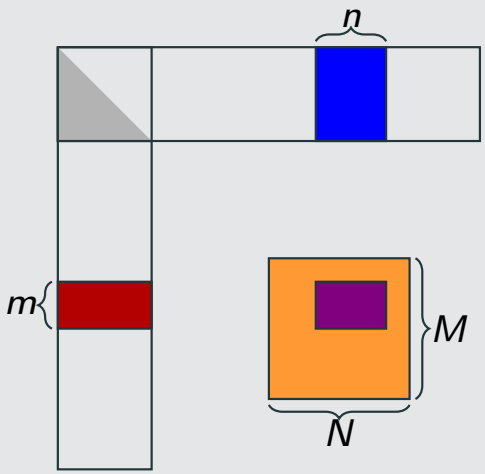
Operations

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

Strategies to apply the updates ($C = C - AB$)

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

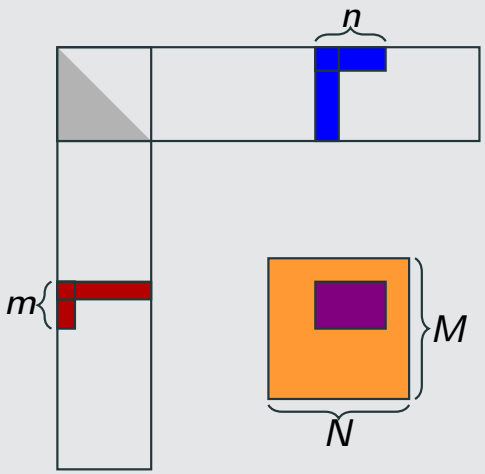
Full rank



- A single GEMM kernel
- Cost depends on mn

Non-fully structured

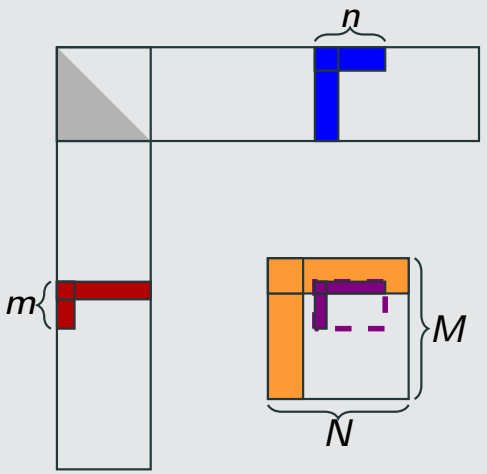
Just-In-Time



- Small GEMM kernels
- Cost depends on mn

Fully structured

Minimal Memory



- Complex operation with padding
- Cost depends on MN
- Recompression (RRQR or SVD) is needed

The two compression scenarios

Algorithm 1 Minimal Memory (MM) - Just in Time (JIT) 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 (JIT)

- Late 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 (MM)

- Early Compression
  - *Before any other operation*
- Fully structured updates
- ✓ Reduce memory footprint
- ✗ May increase the flop count
- ✗ May increase the time to solution

Objective

Take advantage of JIT and MM at the same time

- Early compression of highly compressible blocks to reduce memory
- Late compression of poorly compressible blocks to reduce flop/time

How to determine the compressibility of the blocks?

- Marchal et al. [1] introduced models to determine the block compressibility at run-time.

Our approach

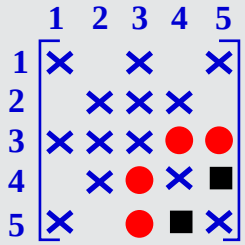
Exploit the fill-in levels concept of Incomplete LU (ILU) factorization

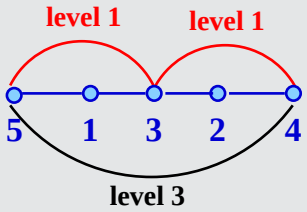
---

[1] L. Marchal, T. Marette, G. Pichon, and F. Vivien. Trading performance for memory in sparse direct solvers using low-rank compression. *Future Generation Computer Systems*, 130:307-320, 2022.

Block ILU Factorization

Recalls on ILU





- The fill-in level of an entry (in L) somehow illustrates the importance of its *magnitude*.
- Higher the fill-in level of a block, higher the distance of the interactions it represent in L.
- Easy and cheap heuristic to apply in a block-wise manner.

Low level blocks

- Have larger ranks
- Marginally improve memory footprint

$\rightarrow$ Suitable for late compression

High level blocks

- Have smaller ranks
- Greatly improve memory footprint

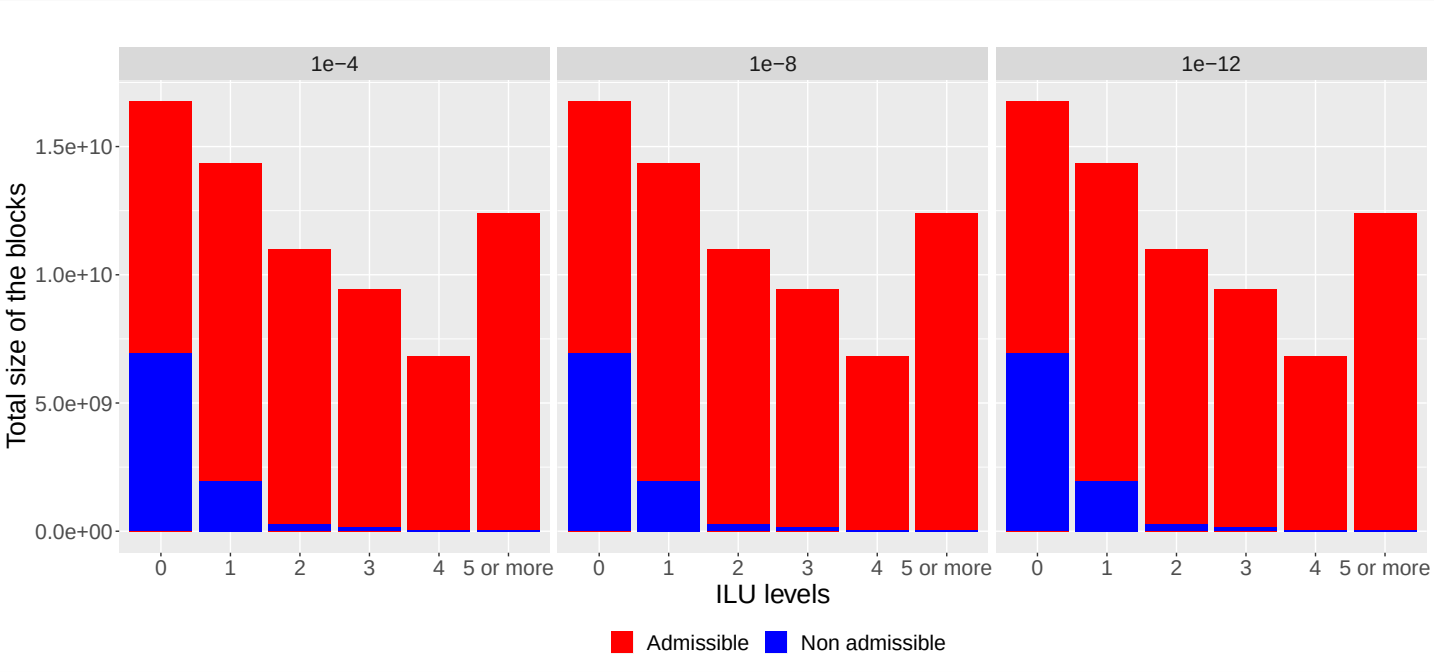
$\rightarrow$ Suitable for early compression

Cholesky based block ILU level definition

Algorithm 2 Block-wise symbolic ILU(*maxlevel*) factorization

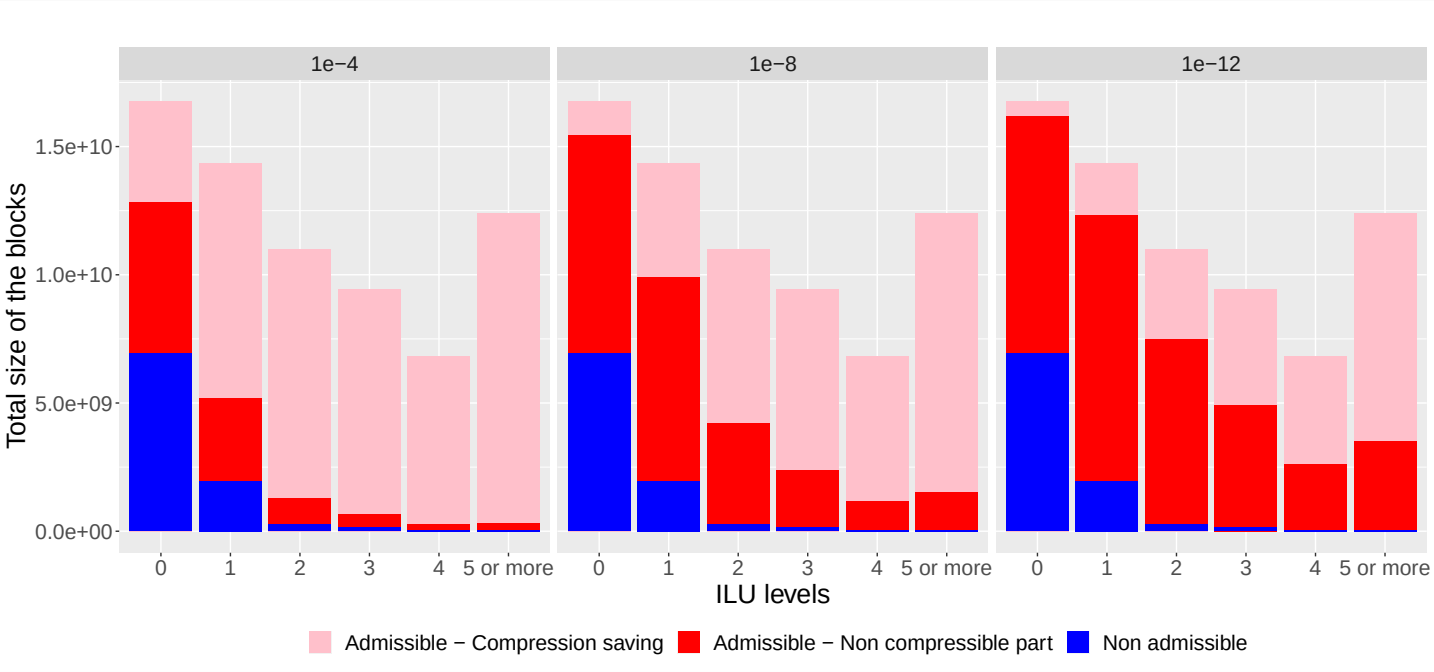
1: /* *Initialize the levels* */
2: **for all** block A_{ij} in A **do**
3: **if** $A_{ij} \neq 0$ **then**
4: $lvl(A_{ij}) = 0$
5: **else**
6: $lvl(A_{ij}) = \infty$
7: /* *Simulate the block factorization* */
8: **for all** column block A_{*k} in A **do**
9: **for all** block A_{ik} in A_{*k} **do**
10: **for all** block A_{jk} in A_{*k} (with $j > i$) **do**
11: $lvl(A_{ij}) = min(lvl(A_{ij}),\ lvl(A_{ik}) + lvl(A_{jk}) + 1)$

Admissibility of blocks at each ILU block level and tolerance



- Statistics on a set of 31 matrices from SuiteSparse Matrix Collection
- Red represents the memory used by the admissible blocks
- Blue represents the memory used by the non-admissible blocks

Compression ratio wrt ILU block levels and tolerance



- Statistics on a set of 31 matrices from SuiteSparse Matrix Collection
- Red shows the memory used by the compressed admissible blocks
- Pink shows the saved memory by compressing 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

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 early to late compression according to the input matrix and tolerance

Experiments

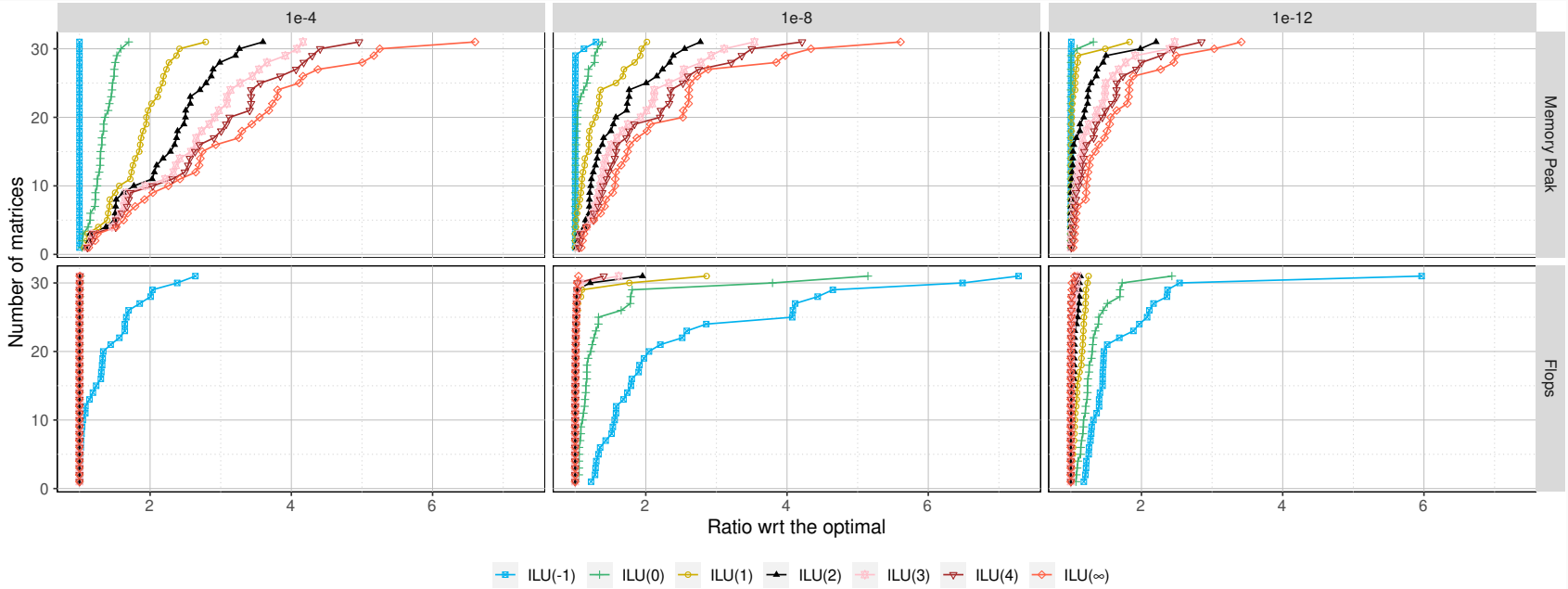
Context

- 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

- We compare our heuristic with different *maxlevel* values ($ILU(maxlevel)$)
- $ILU(-1)$ stands for the existing *Minimal Memory* scenario
- $ILU(\infty)$ stands for the existing *Just-In-Time* scenario
- If a curve is closer to $x = 1$, this method is better on average

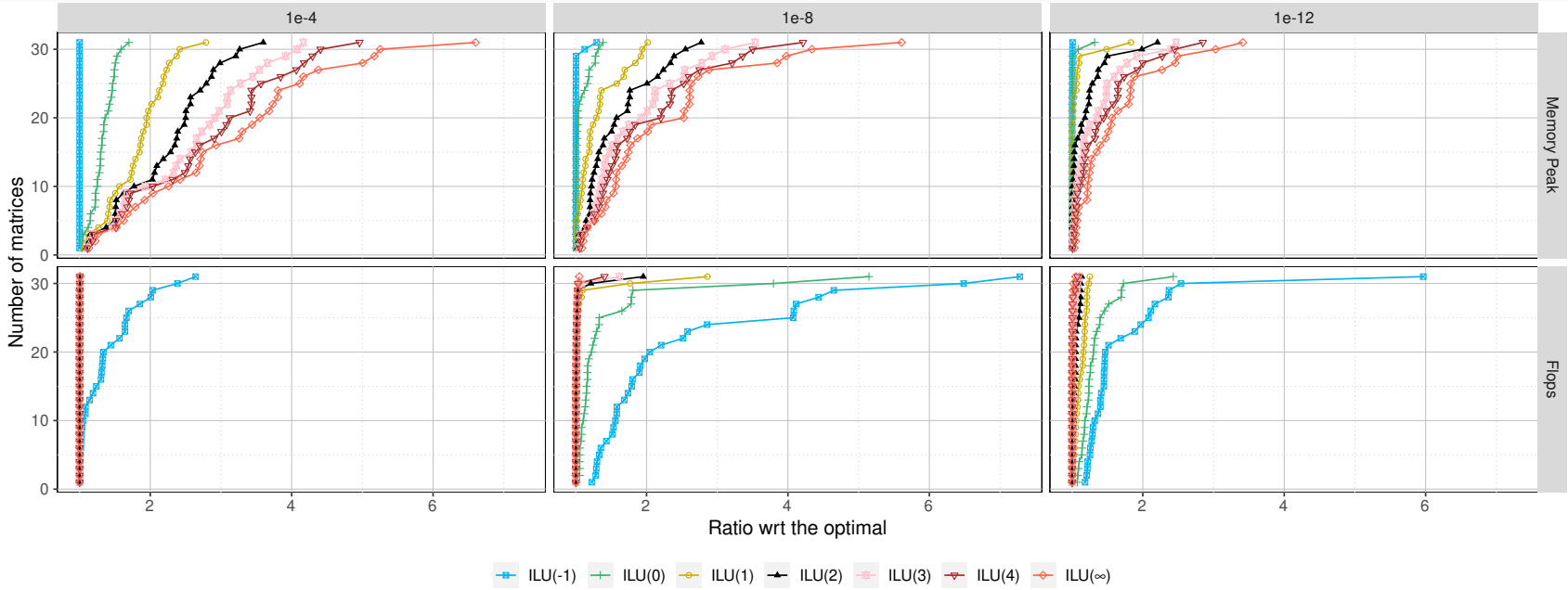
Impact of the ILU(k) Heuristic on the memory and flops



Memory consumption

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

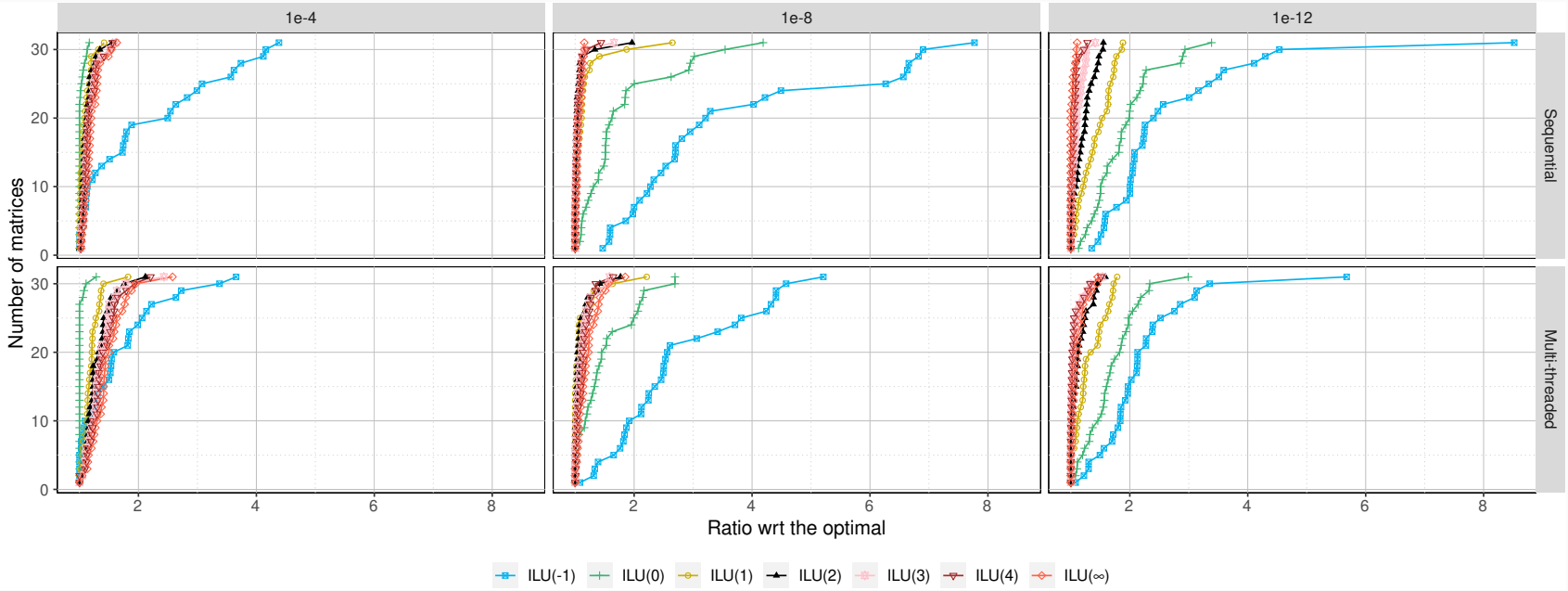
Impact of the ILU(k) Heuristic on the memory and flops



Flops

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

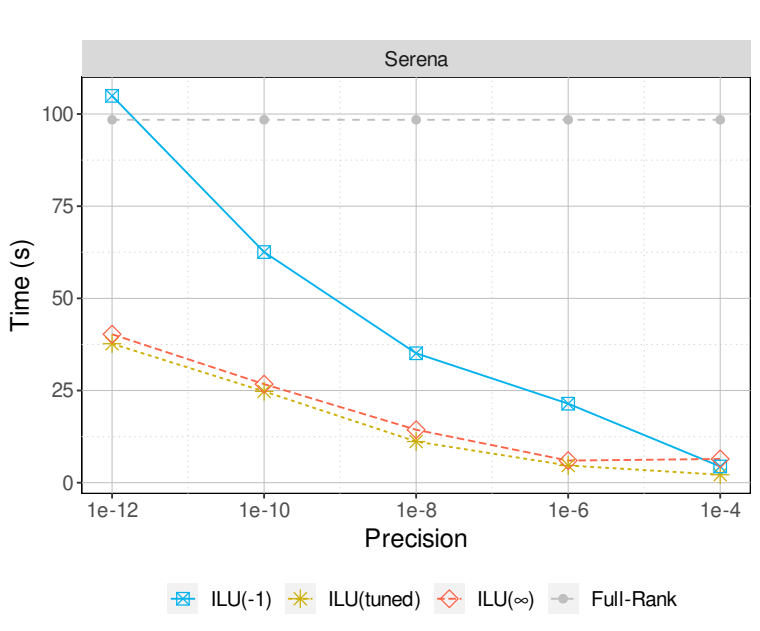
Impact of the ILU(k) Heuristic on the factorization time



Sequential vs multi-threaded (24 threads)

- 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 is 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

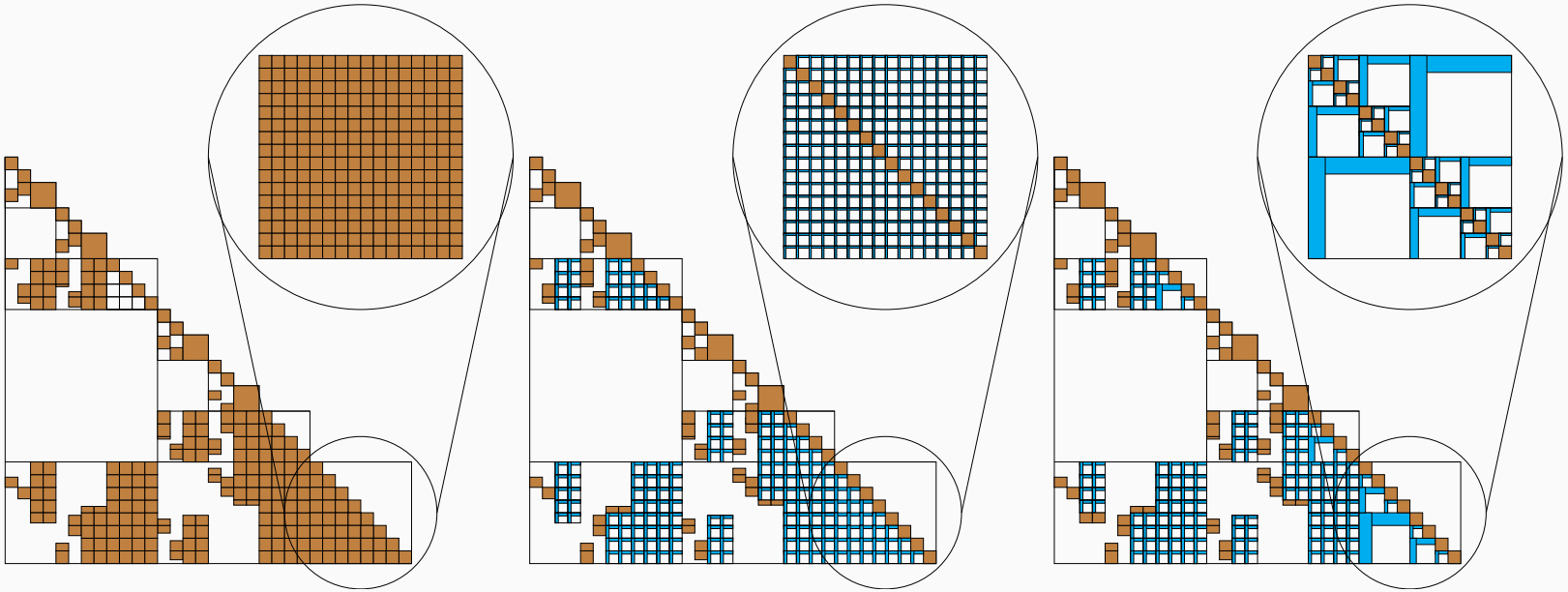
- Allows to reach good trade-offs between the *Minimal Memory* and the *Just-In-Time* strategies
- The new heuristic in sequential provides huge speedup and reduces flops with only a slight memory increase
- The new heuristic in parallel is even faster than *Just-In-Time*, with improved memory

Future work

- Infer automatically the level depending on the user's requirements and matrices
- Inject ILU based information to the run-time preselection method offered by Marchal et al.
- Reduce cost by recompressing accumulated updates

Thank You!

Low-rank representations



- Full-rank format: Too much time and memory
- Low-rank approximation ($A \approx U \times V^T$) on the blocks of:
  - Non-hierarchical Format: Simpler to implement and more flexible in parallel environments
  - Hierarchical Format: Less computational complexity and memory consumption