On the design of parallel linear solvers for large scale problems

ICIAM - August 2015 - Mini-Symposium on Recent advances in matrix computations for extreme-scale computers

M. Faverge, X. Lacoste, G. Pichon, P. Ramet, J. Roman

Outline

1. Sparse Direct Solvers
2. GPU with runtime implementation
3. Supernode Ordering Problem
4. Introducing H-Matrix in PaStiX

1

Sparse Direct Solvers

Context

Problem: solve $Ax = b$

- Cholesky: factorize $A = LL^T$ (symmetric pattern $(A + A^T)$ for LU)
- Solve $Ly = b$
- Solve $L^T x = y$

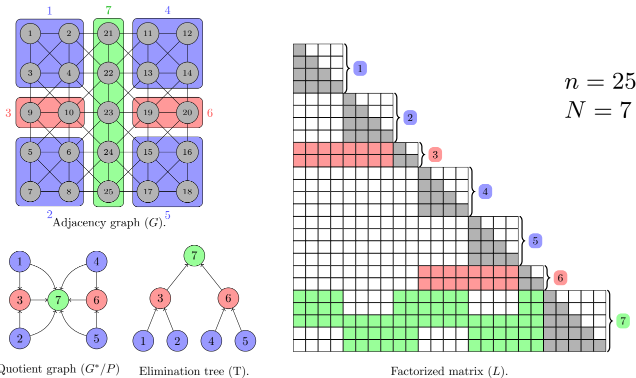
Sparse Direct Solvers: PaStiX approach

1. Order unknowns to minimize the fill-in
2. Compute a symbolic factorization to build L structure
3. Factorize the matrix in place on L structure
4. Solve the system with forward and backward triangular solves

Symbolic Factorization (1)

General approach

1. Use the nested dissection process to partition a sparse matrix
2. Use the minimum degree solution when leaves are small enough
3. Order a supernode thanks to the Reverse Cuthill-McKee algorithm



Adjacency graph (G).

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

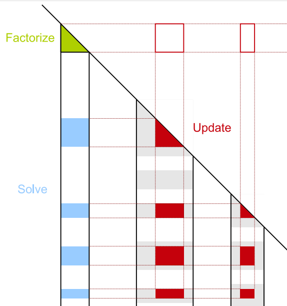
Elimination tree (T).

Factorized matrix (L).

Numerical Factorization

Algorithm to eliminate the block column k

1. Factorize the diagonal block
2. Solve off-diagonal blocks in the current column (TRSM)
3. Update the underlying matrix with the column's contribution (GEMM)



Update

- Compacted matrix-matrix product
- Update divided into the number of off-diagonal blocks receiving contributions

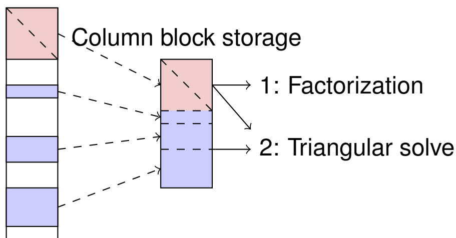
2

GPU with runtime implementation

Panel factorization

Steps

1. Factorization of the diagonal block (`xxTRF`)
2. solve $X \times b_{\text{diag}} = b_{\text{off-diag}}$, with off-diag $>$ diag (`TRSM`)



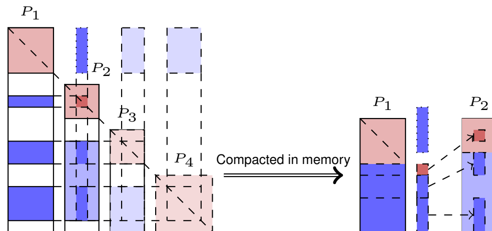
Trailing supernodes updates

The operations to perform

- One matrix-matrix (`GEMM`) operation for each block in the column block

In practice in PASTIX

1. One global `GEMM` per block in a temporary buffer for more performance
2. Scatter addition (many `AXPY`)



Panel update

A new sparse kernel for GPU

Problem

- A CUDA call $\Rightarrow$ a CUDA start-up paid
- Many `AXPY` $(A \times X + Y)$ calls $\Rightarrow$ loss of performance

Solution

- A new GPU kernel to compute all the updates from P_1 on P_2 at once

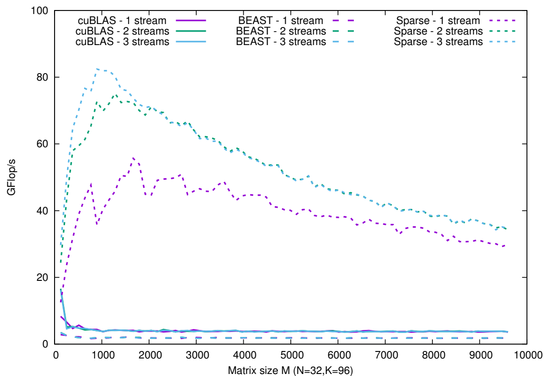
Based on auto-tuning `GEMM` CUDA kernel

- Auto-tuning done by the framework BEAST (developed for MAGMA at ICL and inspired from ATLAS)
- CUBLAS is a proprietary library

Sparse `GEMM` cuda kernel

- Added sparsity information about P_1 and P_2 using 2 arrays
- Computes an offset used when adding to the global memory

Sparse kernel vs multiple dense call (NVIDIA M2070)



- 35 to 80 GFlop/s (peak cuBLAS GEMM 250 GFlop/s)
- 20 to 40 times faster than multiple cuBLAS calls
- 40 to 80 times faster than multiple BEAST calls

DAG schedulers considered

StarPU

- RunTime Team – Inria Bordeaux Sud-Ouest
- Dynamic Task Discovery
- Computes cost models on the fly
- Multiple kernels on the accelerators
- Multiple scheduling strategies: Minimum Completion Time, Local Work Stealing, user defined...

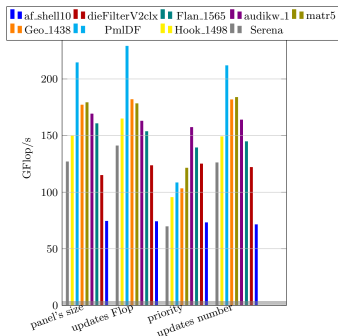
PaRSEC (formerly DAGuE)

- ICL – University of Tennessee, Knoxville
- Parameterized Task Graph
- Only the most compute intensive kernel on accelerators
- Scheduling strategy based on static performance model
- GPU multi-stream enabled

Constraining the data mapping on the GPUs

Choose which GEMM will run on GPUs

- Statically decide to map only selected panels on GPUs following a given criterion:
  - Number of updates received by the panel
  - Position in the critical path
  - Number of flops received by the panel
  - Surface of the target panel



GFlop/s following the criterion with PaRSEC (3 GPUs, 3 streams)

Matrices and Machines

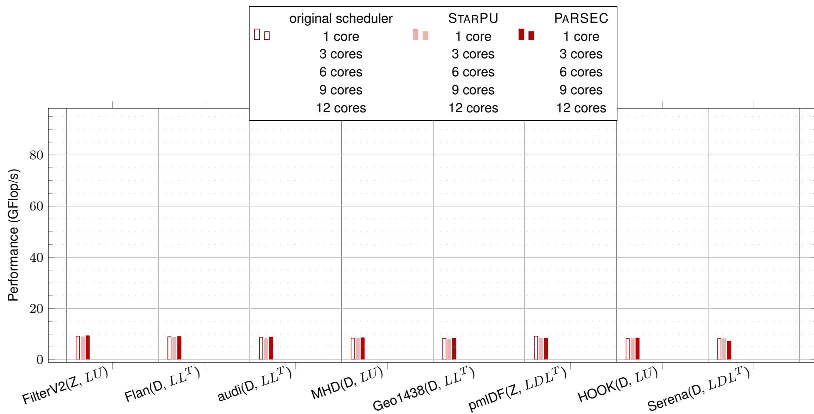
Matrix	Prec	Method	Size	nnz_A	nnz_L	TFlop
FilterV2	Z	LU	0.6e+6	12e+6	536e+6	3.6
Flan	D	LL^T	1.6e+6	59e+6	1712e+6	5.3
Audi	D	LL^T	0.9e+6	39e+6	1325e+6	6.5
MHD	D	LU	0.5e+6	24e+6	1133e+6	6.6
Geo1438	D	LL^T	1.4e+6	32e+6	2768e+6	23
Pmldf	Z	LDL^T	1.0e+6	8e+6	1105e+6	28
Hook	D	LU	1.5e+6	31e+6	4168e+6	35
Serena	D	LDL^T	1.4e+6	32e+6	3365e+6	47

Matrix description (Z: double complex, D: double).

Machine	**Processors**	**Frequency**	**GPUs**	**RAM**
Mirage	Westmere Intel Xeon X5650 (2×6)	2.67 GHz	Tesla M2070 $(\times 3)$	36 GB

CPU scaling study

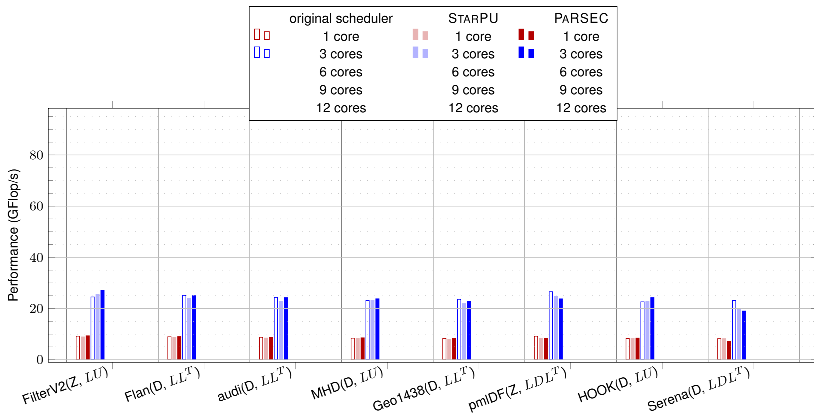
GFlop/s in numerical factorization
(2 hexa-core Intel Xeon X5650)



- Same performances as PaStiX original finely tuned scheduler
- 80 GFlop/s out of 12*4*2.67GHz = 128 GFlop/s practical peak
- Can handle heterogeneous system more easily

CPU scaling study

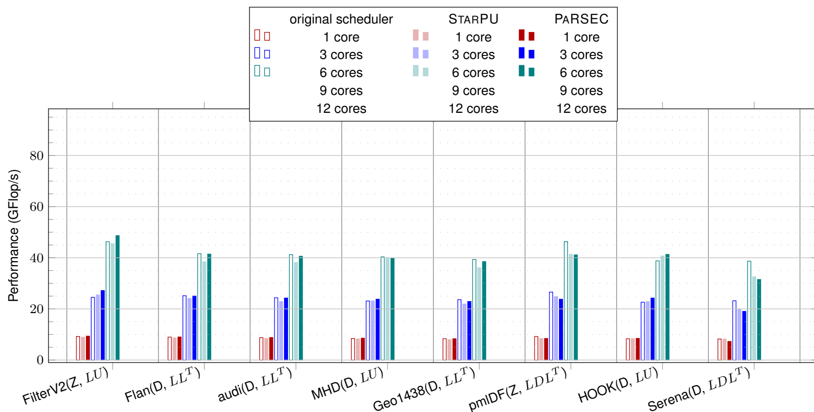
GFlop/s in numerical factorization
(2 hexa-core Intel Xeon X5650)



- Same performances as PaStiX original finely tuned scheduler
- 80 GFlop/s out of 12*4*2.67GHz = 128 GFlop/s practical peak
- Can handle heterogeneous system more easily

CPU scaling study

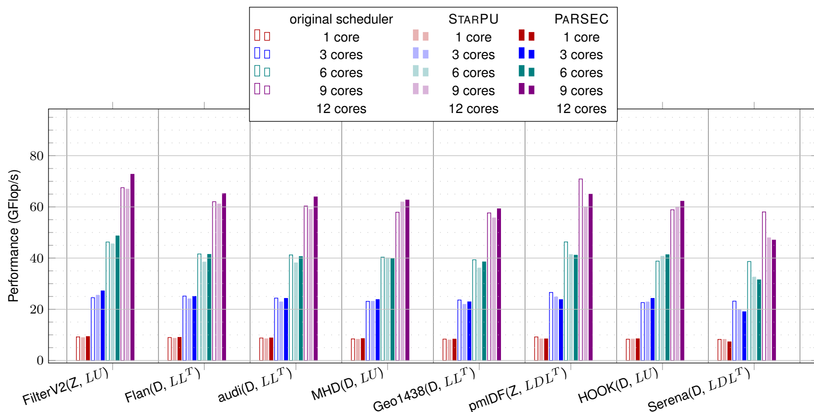
GFlop/s in numerical factorization (2 hexa-core Intel Xeon X5650)



- Same performances as PaStiX original finely tuned scheduler
- 80 GFlop/s out of 12*4*2.67GHz = 128 GFlop/s practical peak
- Can handle heterogeneous system more easily

CPU scaling study

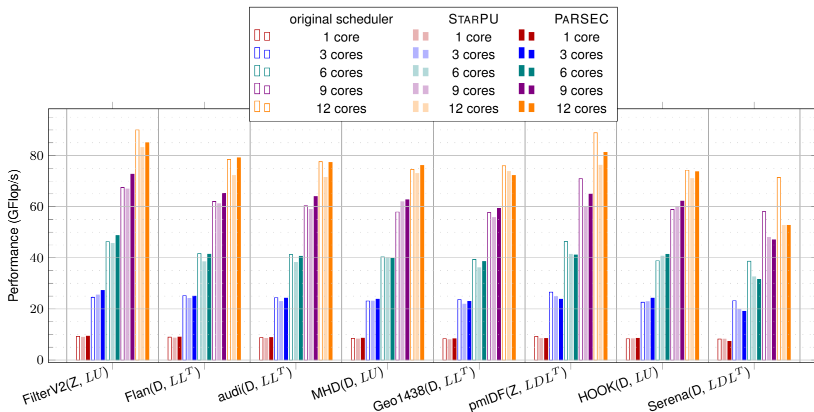
GFlop/s in numerical factorization
(2 hexa-core Intel Xeon X5650)



- Same performances as PaStiX original finely tuned scheduler
- 80 GFlop/s out of 12*4*2.67GHz = 128 GFlop/s practical peak
- Can handle heterogeneous system more easily

CPU scaling study

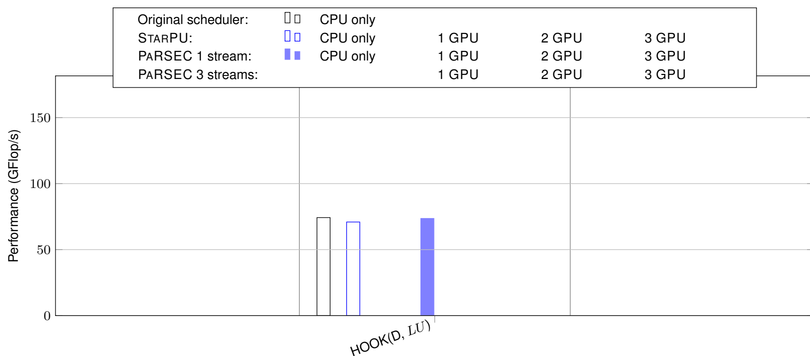
GFlop/s in numerical factorization (2 hexa-core Intel Xeon X5650)



- Same performances as PASTIX original finely tuned scheduler
- 80 GFlop/s out of 12*4*2.67GHz = 128 GFlop/s practical peak
- Can handle heterogeneous system more easily

GPU scaling study

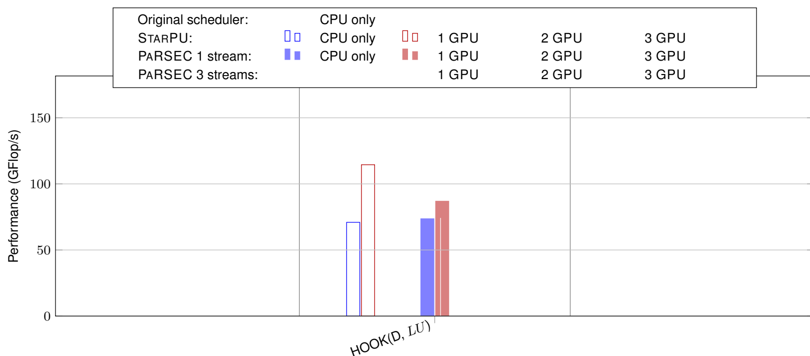
GFlop/s in numerical factorization (2 hexa-core Intel Xeon X5650 + 3 NVIDIA M2070)



- Up to 160 GFlop/s (x3.1) more using 3 GPUs with STARPU
- Up to 120 GFlop/s (x2.4) more using 3 GPUs with PARSEC (1 stream)
- Up to 155 GFlop/s (x2.7) more using 3 GPUs with PARSEC (3 streams)

GPU scaling study

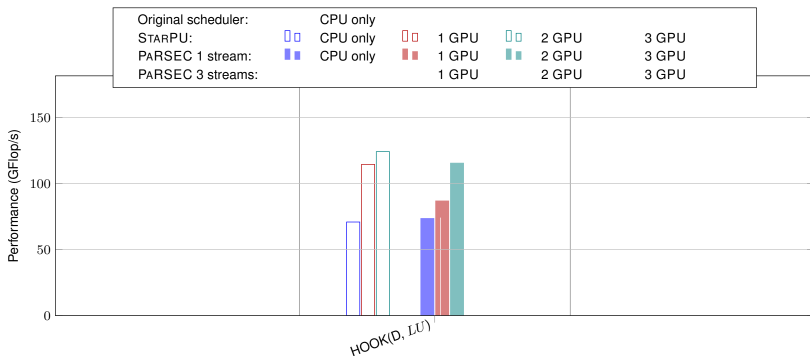
GFlop/s in numerical factorization
(2 hexa-core Intel Xeon X5650 + 3 NVIDIA M2070)



- Up to 160 GFlop/s (x3.1) more using 3 GPUs with STARPU
- Up to 120 GFlop/s (x2.4) more using 3 GPUs with PARSEC (1 stream)
- Up to 155 GFlop/s (x2.7) more using 3 GPUs with PARSEC (3 streams)

GPU scaling study

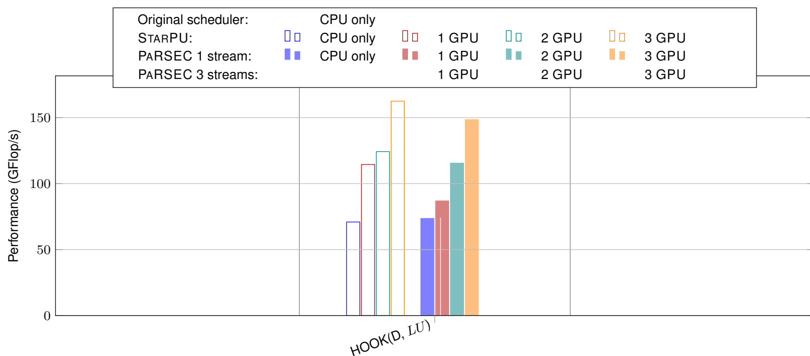
GFlop/s in numerical factorization
(2 hexa-core Intel Xeon X5650 + 3 NVIDIA M2070)



- Up to 160 GFlop/s (x3.1) more using 3 GPUs with StarPU
- Up to 120 GFlop/s (x2.4) more using 3 GPUs with PaRSEC (1 stream)
- Up to 155 GFlop/s (x2.7) more using 3 GPUs with PaRSEC (3 streams)

GPU scaling study

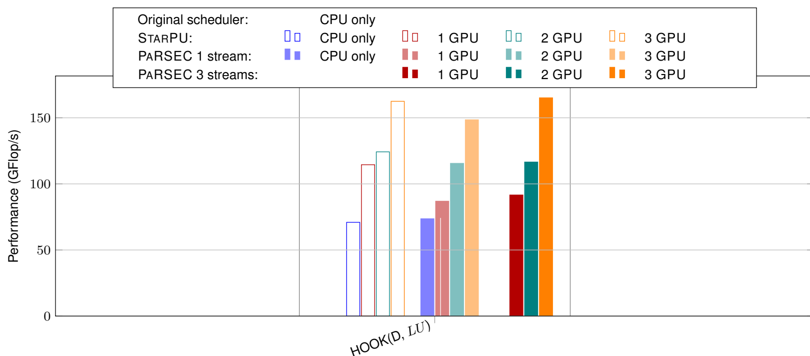
GFlop/s in numerical factorization
(2 hexa-core Intel Xeon X5650 + 3 NVIDIA M2070)



- Up to 160 GFlop/s (x3.1) more using 3 GPUs with STARPU
- Up to 120 GFlop/s (x2.4) more using 3 GPUs with PARSEC (1 stream)
- Up to 155 GFlop/s (x2.7) more using 3 GPUs with PARSEC (3 streams)

GPU scaling study

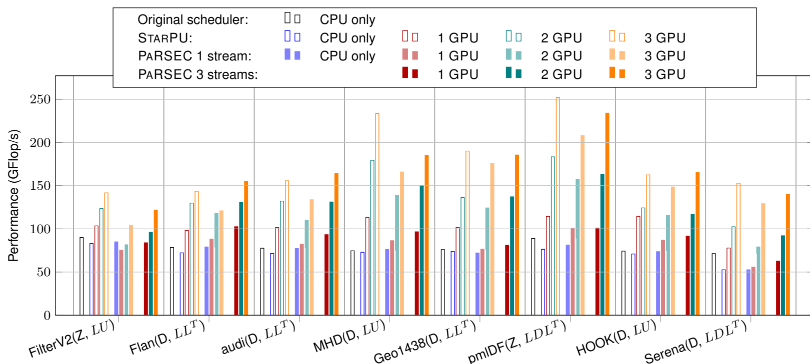
GFlop/s in numerical factorization (2 hexa-core Intel Xeon X5650 + 3 NVIDIA M2070)



- Up to 160 GFlop/s (x3.1) more using 3 GPUs with StarPU
- Up to 120 GFlop/s (x2.4) more using 3 GPUs with PaRSEC (1 stream)
- Up to 155 GFlop/s (x2.7) more using 3 GPUs with PaRSEC (3 streams)

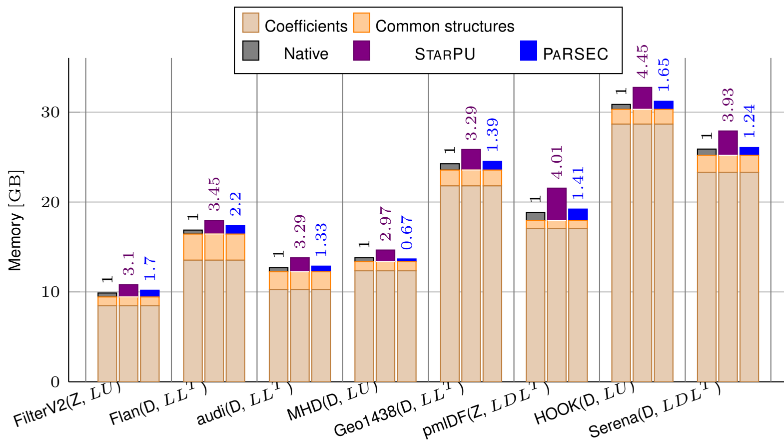
GPU scaling study

GFlop/s in numerical factorization (2 hexa-core Intel Xeon X5650 + 3 NVIDIA M2070)



- Up to 160 GFlop/s (x3.1) more using 3 GPUs with StarPU
- Up to 120 GFlop/s (x2.4) more using 3 GPUs with PaRSEC (1 stream)
- Up to 155 GFlop/s (x2.7) more using 3 GPUs with PaRSEC (3 streams)

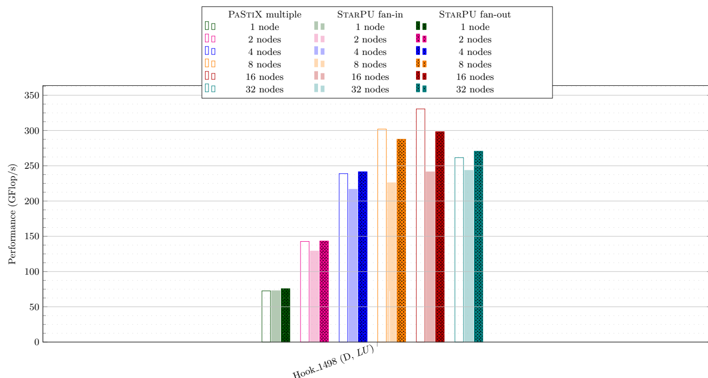
Memory consumption: footprint of the scheduler



- Negligible compared to the coefficients

Distributed results

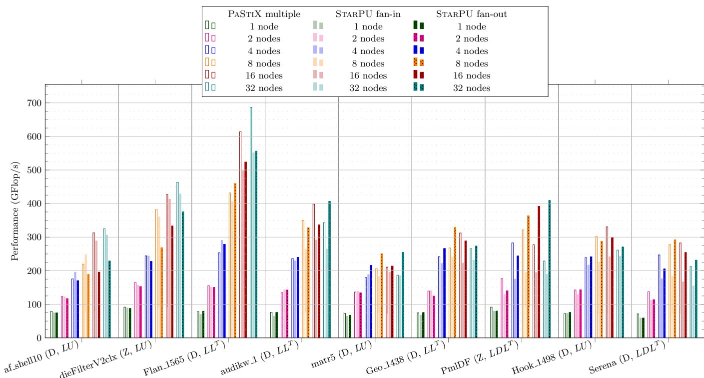
Avakas (2 hexa-core Intel Xeon x5675 per node)



- Original PaStiX slightly better

Distributed results

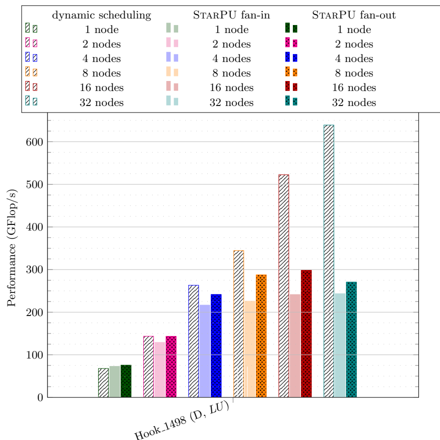
Avakas (2 hexa-core Intel Xeon x5675 per node)



- Original PaStiX slightly better

Runtimes vs PaStiX original dynamic scheduling

Avakas
(2 hexa-core Intel Xeon x5675 per node)



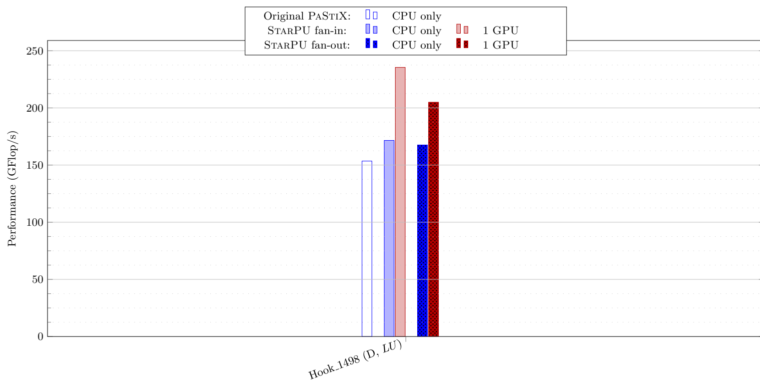
Dynamic scheduling

- Mathieu Faverge's PhD thesis
  $\Rightarrow$ Efficient hybrid MPI/P-Thread
- Work stealing
- Adaptive granularity
- More reactive communications

- Original PaStiX with dynamic scheduling still really better
  $\Rightarrow$ Need to work together with the runtime teams

Distributed heterogeneous

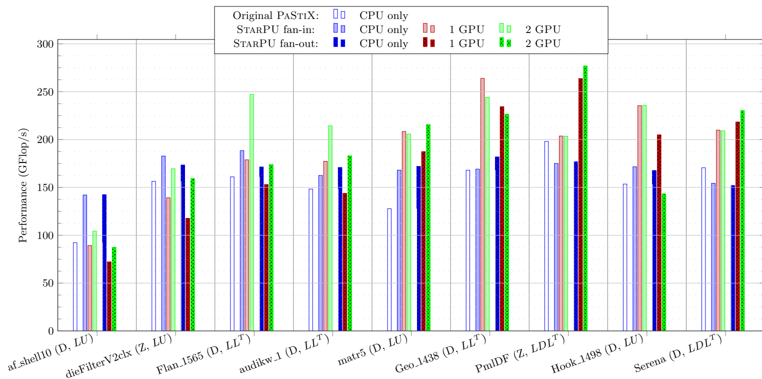
Curie
(4 MPI, 2 quad-core Westmere, 2 NVIDIA M2090)



- GPU accelerate the performance (+54% in fan-in, +33% in fan-out, with one GPU on `hook_1498`).

Distributed heterogeneous

Curie
(4 MPI, 2 quad-core Westmere, 2 NVIDIA M2090)



- GPU accelerate the performance (+54% in fan-in, +33% in fan-out, with one GPU on `hook_1498`).

Conclusion

Sparse direct solver on top of task-based runtime systems

One of the first attempts to build a complex and irregular application on top of task-based runtime systems. (QR MUMPS on task-based runtime systems, CHOLMOD and SUPERLU target GPU)

More interactions with the runtime systems

- Adaptive granularity: Terry Cojean's PhD thesis on divisible tasks in STARPU
- Data locality: Andra Hugo's PhD thesis on contexts in STARPU

3

Supernode Ordering Problem

Motivations

Amalgamation techniques

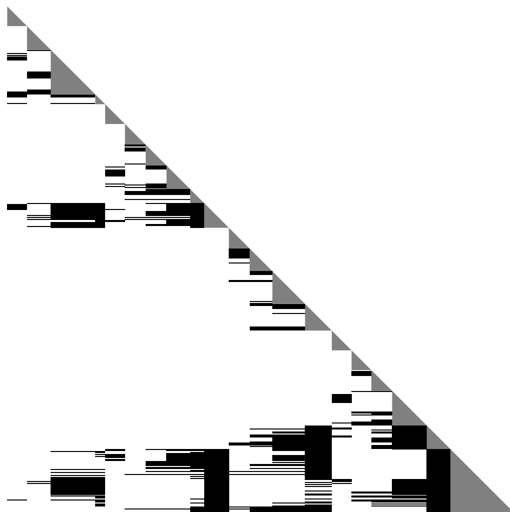
- Operations on data blocks are more efficient
- Preprocessing stages on the matrix structure before numerical operations
- Those steps can be used for several systems presenting the same initial structure, or for several right-hand-sides

Objectives

- Increase BLAS efficiency by reducing the number of off-diagonal blocks
- Reduce RUNTIME overhead, with larger tasks

The number of non-zeros, as well as the number of operations, is kept the same

Symbolic Factorization (2)



$n = 8 \times 8 \times 8$
$N = 20$

Related Work: Reverse Cuthill-McKee (RCM) algorithm

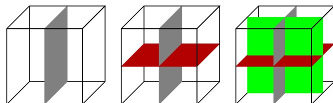
General idea - Breadth-First Search

1. Choose a peripheral vertex x, ordered as first vertex
2. Order vertexes interacting with x (neighbourhood at distance d)
3. Iterate starting with those vertexes (neighbourhood at distance $d + 1$)

Drawbacks

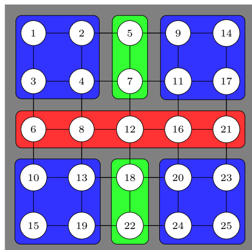
- Work on A structure instead of L structure
- Do not consider contributing supernodes, but only intra-node interactions
- Order supernodes during the nested dissection process while it could be realized after

Ordering Last Supernode

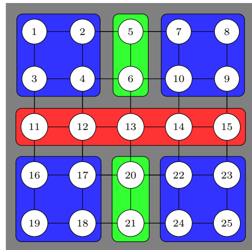


Example

- $n = 5 \times 5 \times 5$
- $N = 1 + 2 + 4 + 8 = 15$
- First separator of size 5×5



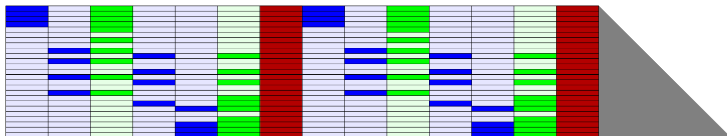
RCM



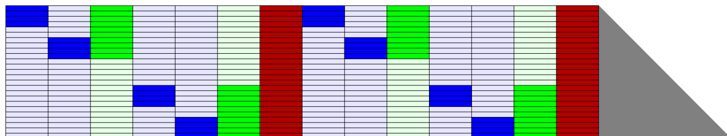
Optimal

Projection of contributing supernodes and ordering of the first separator

Resulting Symbolic Structures

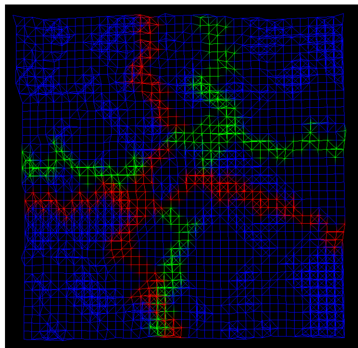


Symbolic Factorization: RCM



Symbolic Factorization: Optimal

Practical Ordering with Scotch



Last supernode of a 3D Laplacian (size 40)

Subparts A and B are partitioned differently. Thus, supernodes projection is less regular than in our previous example

Modeling of the Problem (1)

Notations

- We consider the ℓth diagonal block C_ℓ
- Contributing supernodes are included in $(C_k)_{k\in[1,\ell-1]}$

Supernode contributions to C_ℓ

$$row_{ik}^{\ell} = \begin{cases} 1 \text{ if vertex } i \text{ from } C_\ell \text{ is connected to } C_k \\ 0 \text{ otherwise} \end{cases}, k \in [1, \ell-1], i \in [1, |C_\ell|]$$

Set of contributions for line i:
$B_i^\ell = (row_{ik}^\ell)_{k\in[1,\ell-1]}$

Modeling of the Problem (2)

Metrics

- Weight of line i

$$w_i^\ell = \sum_{k=1}^{\ell-1} row_{ik}^\ell$$

- Distance between lines i and j

$$d_{i,j}^\ell = d(B_i^\ell, B_j^\ell) = \sum_{k=1}^{\ell-1} row_{ik}^\ell \oplus row_{jk}^\ell$$

with $\oplus$ the exclusive or operation.
measure the number of blocks created by line j and ended at line i

Modeling of the Problem (3)

Quality: Number of off-diagonal blocks

$$odb^{\ell} = \frac{1}{2}(w_1^{\ell} + \sum_{i=1}^{|C_{\ell}|-1} d_{i,i+1}^{\ell} + w_{|C_{\ell}|}^{\ell})$$

Optimal solution to minimize odb^{ℓ}

- Shortest Hamiltonian Path problem: find the shortest path visiting once each line, with a constraint on first and last line
- Complete symmetric graph: $d_{ij}^{\ell} = d_{ji}^{\ell}$ and $d_{ij}^{\ell} \leq d_{ik}^{\ell} + d_{kj}^{\ell}$

Proposition

Traveller Salesman Problem

- Find a cycle minimizing

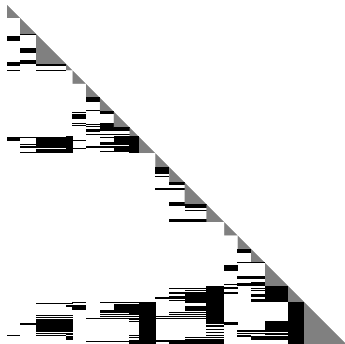
$$\sum_{i=1}^{|C_\ell|} d^{\ell}_{i,(i+1)[|C_\ell|]}$$

- Add a fictive vertex S_0, without any contribution to build a cycle instead of a path

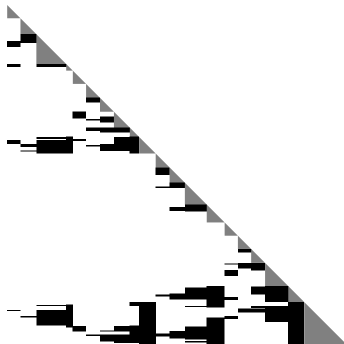
Algorithm

1. Build the set B_i^ℓ for each line i of C^ℓ
2. Compute the distance matrix
3. Insert lines to minimize the cycle length
4. Split the cycle at fictive vertex to get the path

Resulting Solution - Example



Without reordering



With reordering

Reordering on a $8 \times 8 \times 8$ laplacian

Works whatever is the initial seed

Complexity

Type	σ	Reordering	Factorization
2D	$\frac{1}{2}$	$\Theta(n\sqrt{n})$	$\Theta(n\sqrt{n})$
3D	$\frac{2}{3}$	$\Theta(n^{\frac{5}{3}})$	$\Theta(n^2)$

Complexity for regular meshes

Asymptotically faster than the numerical factorization for $\sigma > \frac{1}{2}$
Remind that RCM is well working in 2D case

Experimental Conditions

Set of matrices

- Large matrices, around 1 million unknowns (real-life meshes)
- Extracted from different applications
- Different average off-diagonal block size

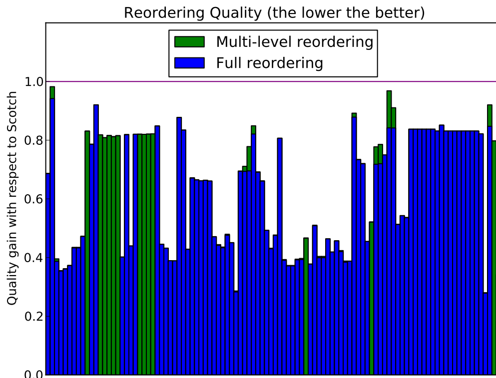
Machine - Curie TGCC

- Two quadcore INTEL `Westmere` running at 2.66 GHz
- Two NVIDIA `M2090 T20A`
- MKL `14.0.3.174`
- Cuda `5.5.22`

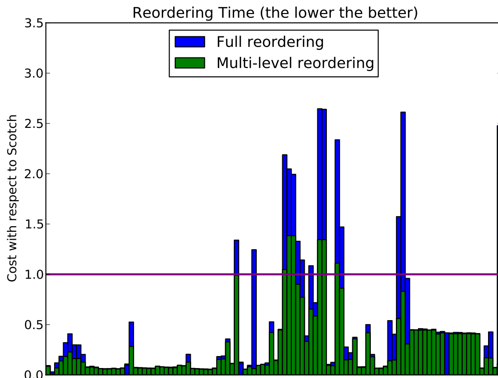
PaStiX

- Use implementation over StarPU (Xavier Lacoste thesis)
- 6 CPUs + 2 GPUs or 8 CPUs
- Large minimum block size for GPUs efficiency

Number of Off-Diagonal Blocks

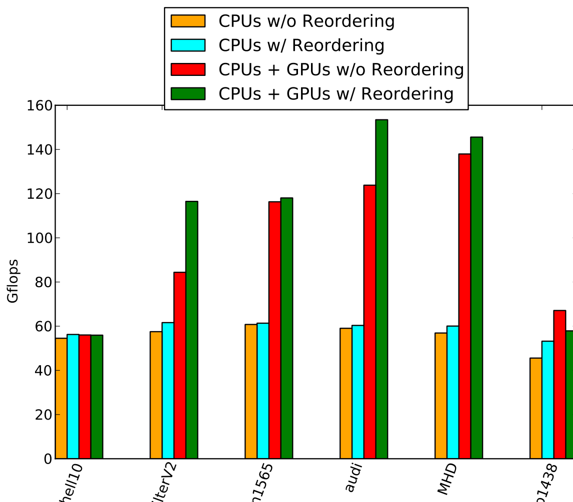


Reordering Cost (sequential)



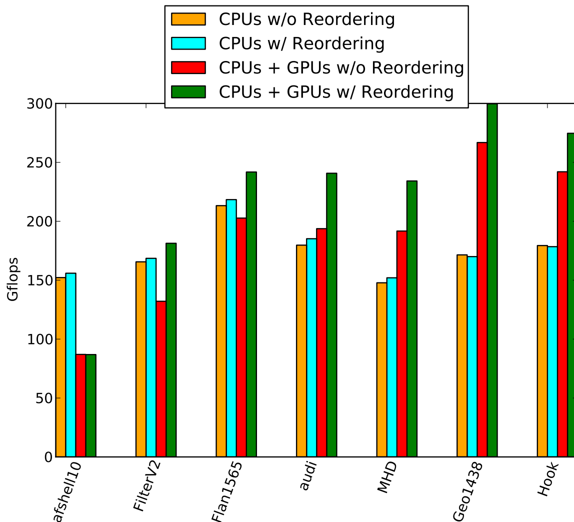
Impact on a CPU+GPU run with StarPU and 1 node (8 CPUs and 2 GPUs)

For the factorization only



Impact on a CPU+GPU run with StarPU and 4 nodes

For the factorization only



Conclusion

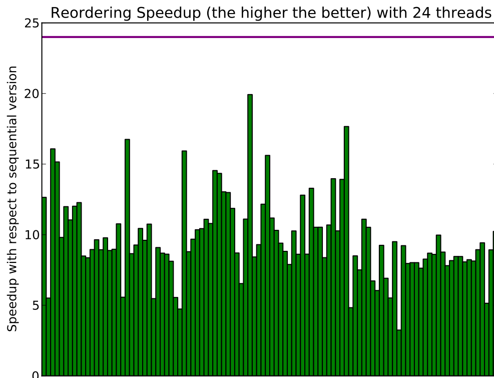
Results

- Number of off-diagonal blocks reduced by a factor between 2 and 3
- Performance gain of the factorization up to 30% in an heterogeneous context
- Theoretical and practical reordering complexity small with respect to the numerical factorization for 3D graphs
- Reduce the reordering cost with a multilevel distance computation

Perspectives

- Study such a strategy for a multifrontal solver (MUMPS)
- Implement the algorithm in a parallel context

Speedup for a multi-threaded reordering step with 24 threads



4

Introducing H-Matrix in PaStiX

Scenario

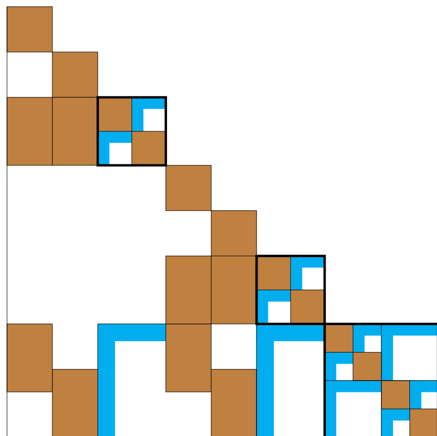
Steps

1. Use direct approach on small supernodes
2. Compress large supernodes: dense diagonal block and off-diagonal blocks
3. Compute a low-rank factorization with both dense and low-rank blocks
4. Use the resulting solution as a “good” preconditioner for iterative methods

Advantages

- Keep the inherent parallelism of the Right-Looking approach in PaStiX
- Do not update an H-structure with an H-structure

Using HODLR in PaStiX



Compress Information

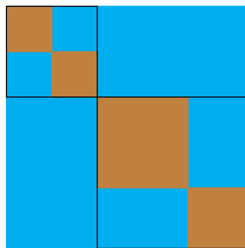
- Large diagonal block: HODLR
- Large off-diagonal blocks: low-rank

Underlying Contributions

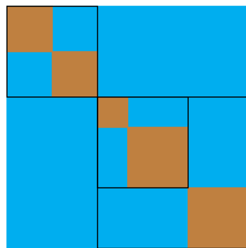
- Dense to low-rank
- Dense to HODLR
- Low-rank to low-rank
- Low-rank to HODLR

HODLR Compression

Level 1



Level 2



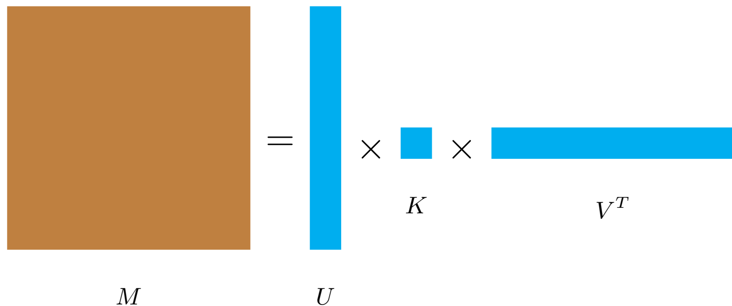
Level 3

Full Rank

Low Rank

Build an HODLR tree: dense diagonal blocks and low-rank off-diagonal blocks.
Take care to keep a connected interface (separator in the HODLR structure)

Low-rank Compression



$$M \in \mathbb{R}^{n\times n},\, U, V \in \mathbb{R}^{n\times r},\, K \in \mathbb{R}^{r\times r}$$

Storage in $2nr + r^2$ instead of n^2

Low rank compression with BDLR, ACA, SVD...

Open Issues

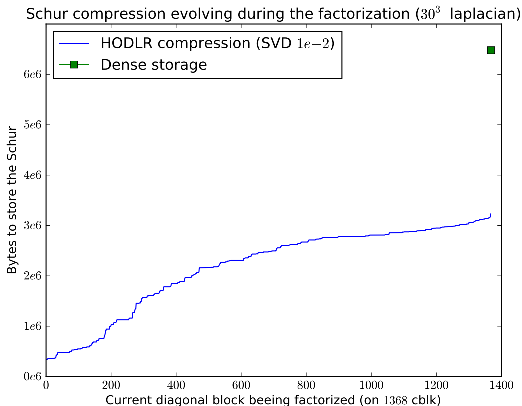
HODLR management

- Factorization and Solve relying on HODLR kernels
- Update: has to be tuned for PaStiX
- Initialization: set the HODLR tree (median bisection?)

Compression structures

- The rank may grow during the factorization. When to compress? in A? after some steps?
- How to choose the low-rank accuracy?
- How to control the number of iterations?

Compression evolution on the Last Supernode of a 30^3 laplacian

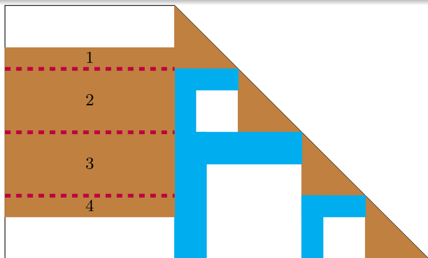


Compressed with HODLR using SVD and $1e-2$ tolerance

Apply an Update

Applying an update to an HODLR structure

- Divide the contribution to fit the HODLR tree
- Update dense or low-rank structures instead of the complete HODLR structure
- Objective: build a low-rank symbolic structure



To enhance efficiency, it is important to study the structure of off-diagonal blocks contributing to a diagonal block

Complexity results for the factorization on general 2D/3D finite element meshes

We consider a subgraph of size p with a separator C, while the original matrix is of size n.

PaStiX 2D

- OPC: $\Theta(n^{\frac{3}{2}})$
- NNZ: $\Theta(n\ ln(n))$

H-PaStiX 2D - $\sigma = 1/2$

- $OPC_C : \Theta(p^{\frac{1}{2}}) \rightarrow \Theta(n)$
- $NNZ_C : \Theta(p^{\frac{1}{2}}) \rightarrow \Theta(n)$

PaStiX 3D

- OPC: $\Theta(n^2)$
- NNZ: $\Theta(n^{\frac{4}{3}})$

H-PaStiX 3D - $\sigma = 2/3$

- $OPC_C : \Theta(p^{\frac{4}{3}} + p) \rightarrow \Theta(n^{\frac{4}{3}})$
- NNZ_C: $\Theta(p) \rightarrow \Theta(n\ ln(n))$

Similar results for HSS (Randomized Sparse Direct Solvers, by Jianlin Xia), except for OPC in 3D: HSS is in $\Theta(n^{\frac{4}{3}}\ ln(n))$

Experiments with the Schur Complement

Context

- Compress only the last diagonal block
- Proof of concept
- Defining the interface to an H-Matrix library

Strategy

1. Eliminate the first block-columns
2. Apply updates to the last supernode (in dense)
3. Compress this supernode
4. Factorize this supernode
5. In the forward/backward steps, replace the two solves on the last supernode by a single HODLR solve
6. Refine with GMRES

Experiments with a regular grid: laplacian of size $30 \times 30 \times 30$

Method	Tolerance	Compression	Guess	Nb iterations	Norm
ACA	1e-1	4.25	1.76e-01	20	1.01e-08
	1e-2	2.37	1.55e-01	20	1.93e-12
	1e-3	1.61	1.16e-01	12	1.84e-13
	1e-4	1.36	5.50e-02	8	6.82e-13
	1e-5	1.26	3.37e-05	3	1.30e-14
	1e-6	1.18	1.17e-06	3	3.87e-15
BDLR	1e-1	2.48	3.12e-02	10	2.16e-13
	1e-2	2.00	9.31e-03	7	1.87e-14
	1e-3	1.65	1.31e-03	5	2.03e-14
	1e-4	1.42	1.40e-04	4	3.41e-15
	1e-5	1.30	7.84e-06	3	4.16e-15
	1e-6	1.23	1.13e-06	3	2.89e-15
SVD	1e-1	2.99	2.45e-02	9	1.85e-13
	1e-2	2.13	3.81e-03	6	3.65e-14
	1e-3	1.72	2.26e-04	4	1.33e-14
	1e-4	1.48	1.77e-05	3	3.06e-14
	1e-5	1.34	1.19e-06	3	3.39e-15
	1e-6	1.26	9.49e-08	2	3.39e-14

Experiments with a real-life matrix (AUDI)

Method	Tolerance	Compression	Guess	Nb iterations	Norm
ACA	1e-1	6.94	2.96e-03	20	4.94e-06
	1e-2	2.94	7.22e-04	20	3.68e-07
	1e-3	2.00	5.66e-05	12	8.50e-13
	1e-4	1.68	2.91e-06	5	4.36e-14
	1e-5	1.50	9.38e-08	3	1.94e-13
	1e-6	1.34	9.05e-09	2	5.87e-13
BDLR	1e-1	4.53	1.20e-03	20	1.84e-06
	1e-2	3.19	3.27e-04	20	1.41e-07
	1e-3	2.43	9.77e-03	15	1.65e-13
	1e-4	1.97	4.02e-06	7	1.68e-13
	1e-5	1.70	3.13e-07	4	3.50e-13
	1e-6	1.52	1.53e-08	3	4.40e-14
SVD	1e-1	5.07	7.48e-04	20	2.16e-07
	1e-2	3.24	1.34e-04	20	1.65e-10
	1e-3	2.45	9.74e-06	7	9.55e-13
	1e-4	1.99	8.27e-07	4	2.75e-13
	1e-5	1.73	8.42e-08	3	4.61e-14
	1e-6	1.54	8.55e-09	2	3.14e-13

Numerical Results

Accuracy of the resulting solution depends on:

- The compression method (SVD is the best)
- The HODLR tree (structure and depth)
- The given tolerance to HODLR code
- The number of iterations allowed in the iterative process
- The block size (we used 30 in our experiments)

Preliminary experiments

- SVD better than BDLR, better than ACA
- Small number of iterations to reach PaStiX original accuracy

Conclusion

Perspectives

- Linear solver for $2D$ general meshes
- A $\Theta(n^{\frac{4}{3}})$ solver for $3D$ general meshes
- Memory improvements
- OPC gain, but probably loss in effectiveness (BLAS)
- Large challenging problems
- Factorization tolerance may depend on the number of right-hand-sides

Future Work

- Develop black-box approach
- Study HODLR tree with respect to off-diagonal contributions
- Verify the condition $x = \frac{d-2}{d-1}$ in practice

Thanks !