Hybrid methods, Hybrid architectures, Hybrid compressions for sparse direct solvers

Linear Algebra Seminar at Stanford

P. Ramet
HiePACS team
Inria Bordeaux Sud-Ouest
LaBRI Bordeaux University

November 22, 2013

Guideline

1 Introduction

Motivations

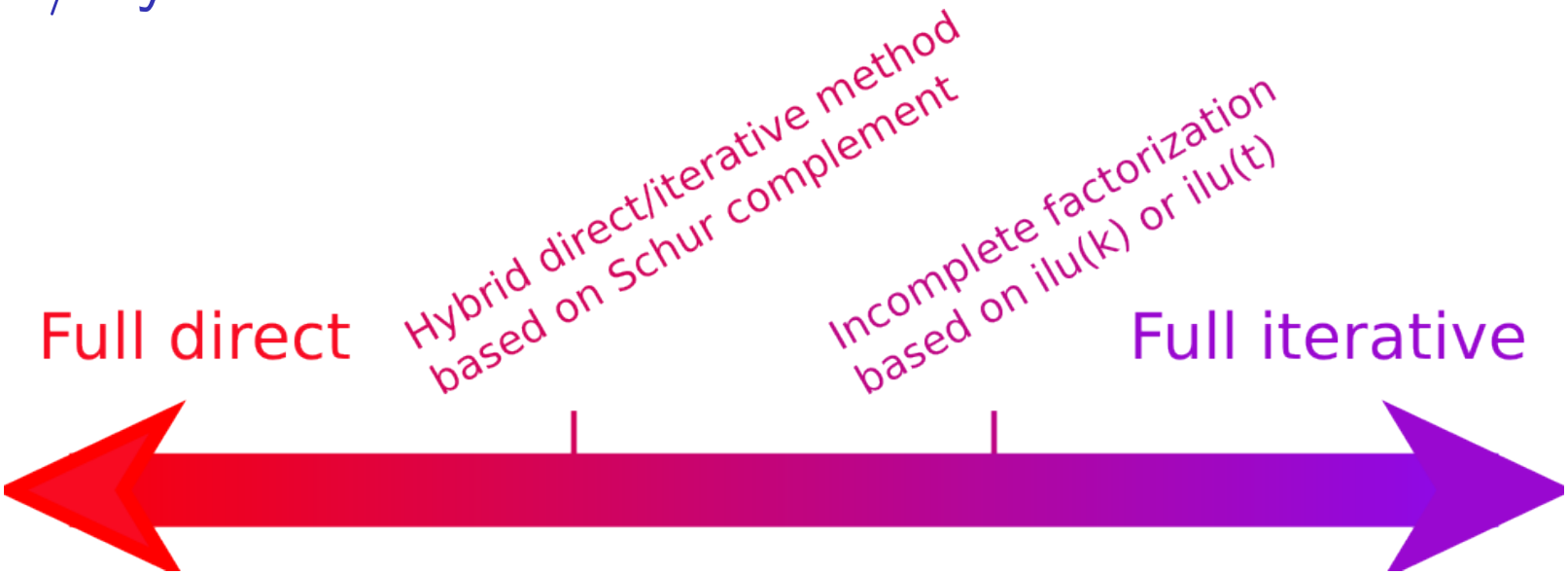
- solve linear systems of equations $\rightarrow$ key algorithmic kernel

$$\textit{Continuous problem}$$
$$\downarrow$$
$$\textit{Discretization}$$
$$\downarrow$$
$$\textit{Solution of a linear system } Ax = b$$

- Main parameters:
  - Numerical properties of the linear system (symmetry, positive definite, conditioning, ...)
  - Size and structure:
    - Large $> 10^6 \times 10^6$ (square / rectangular)
    - Dense or Sparse (structured / unstructured)
    - Target computer (sequential / parallel)

$\rightarrow$ *Algorithmic choices are critical*

Mixed/Hybrid direct-iterative methods



The "spectrum" of linear algebra solvers

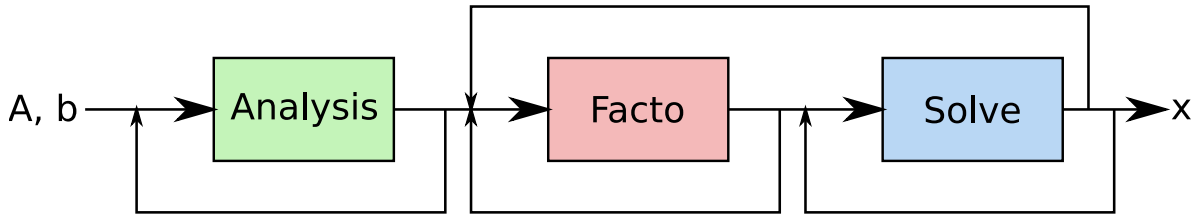
- Robust/accurate for general problems
- BLAS-3 based implementation
- Memory/CPU prohibitive for large 3D problems
- Limited parallel scalability

- Problem dependent efficiency/controlled accuracy
- Only mat-vec required, fine grain computation
- Less memory consumption, possible trade-off with CPU
- Attractive "build-in" parallel features

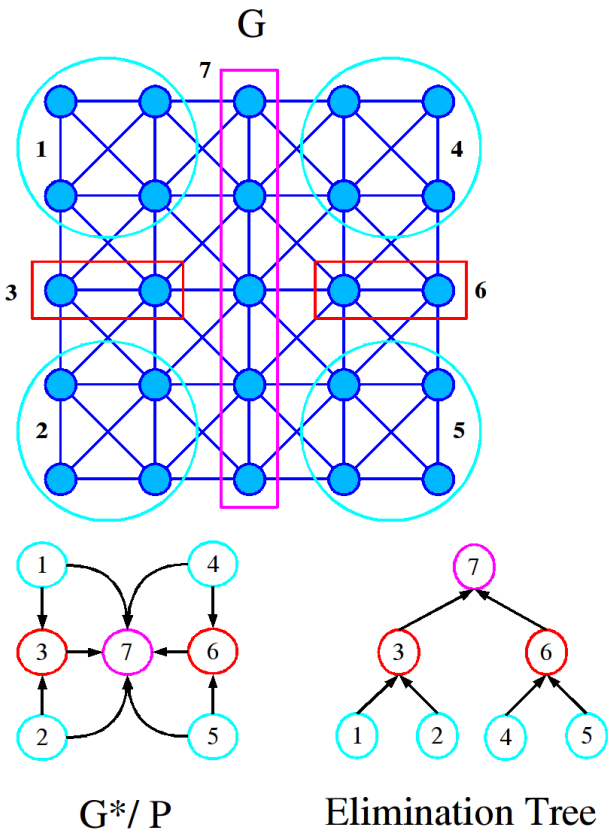
2
Direct sparse factorization

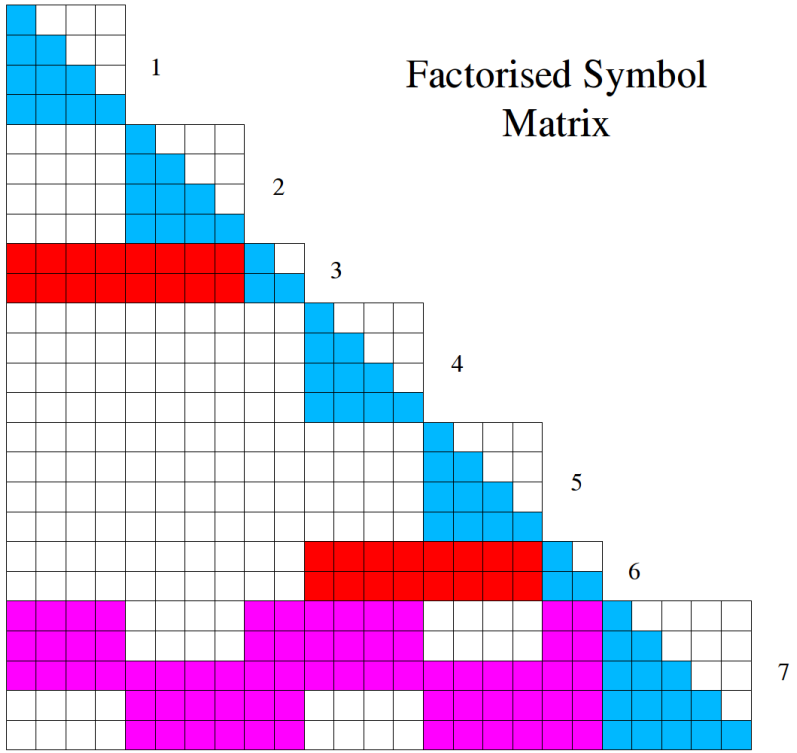
Major steps for solving sparse linear systems

1. Analysis: matrix is preprocessed to improve its structural properties ($A'x' = b'$ with $A' = P_n P D_r A D_c Q P^T$)
2. Factorization: matrix is factorized as $A = LU$, LL^T or LDL^T
3. Solve: the solution x is computed by means of forward and backward substitutions



Direct Method and Nested Dissection [A. George 73]





Symbolic factorization

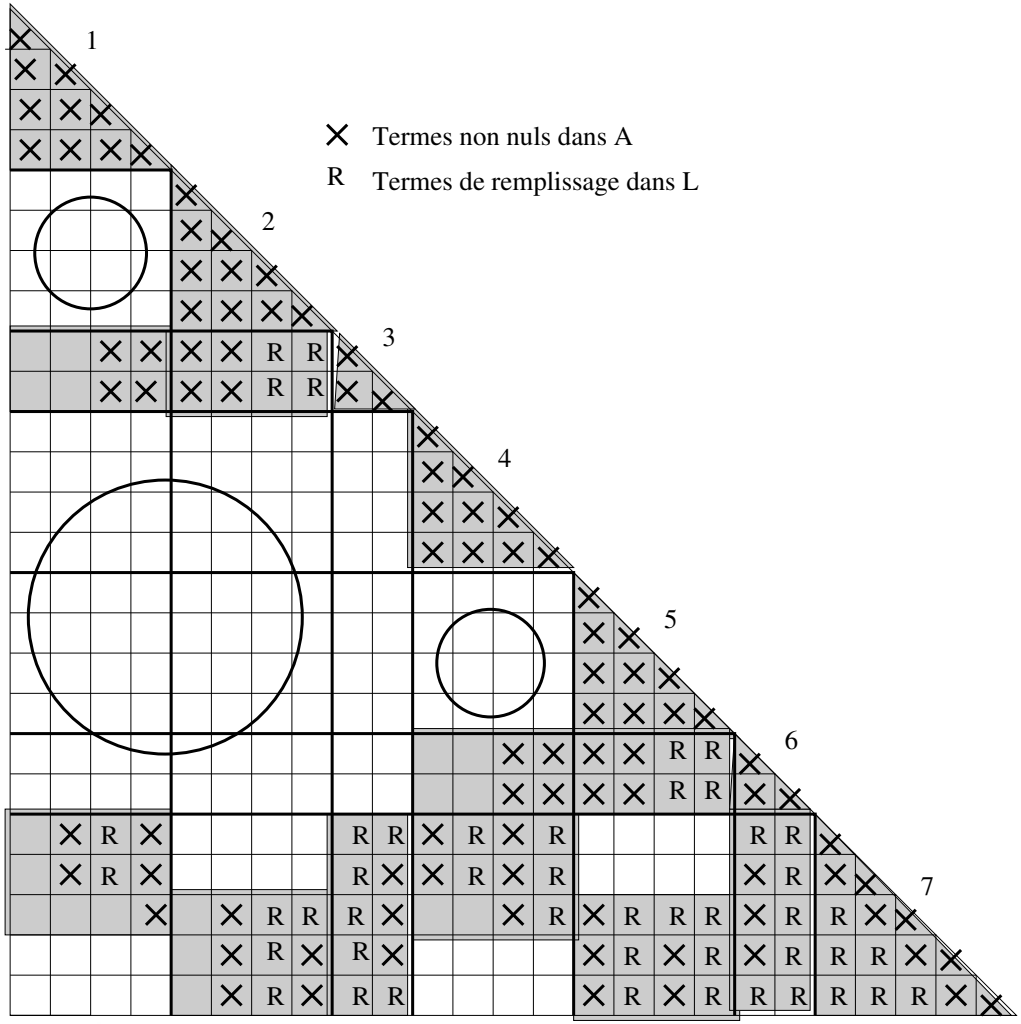
The goal of this algorithm is to build the non-zero pattern of L (and U). We will consider the symmetric case (graph of $A + A^t$ if A has an unsymmetric NZ pattern). In this case the symbolic factorization is really cheaper than the factorization algorithm.

Fundamental property

The symbolic factorization relies on the elimination tree of A.

Quotient graph and block elimination tree

Property of the elimination graph : $Q(G,P)^* = Q(G^*,P)$



Supernodal methods

Definition

A *supernode* (or supervariable) is a set of contiguous columns in the factors **L** that share essentially the same sparsity structure.

- All algorithms (ordering, symbolic factor., factor., solve) generalized to block versions.
- Use of efficient matrix-matrix kernels (improve cache usage).
- Same concept as *supervariables* for elimination tree/minimum degree ordering.
- Supernodes and pivoting: pivoting inside a supernode does not increase fill-in.

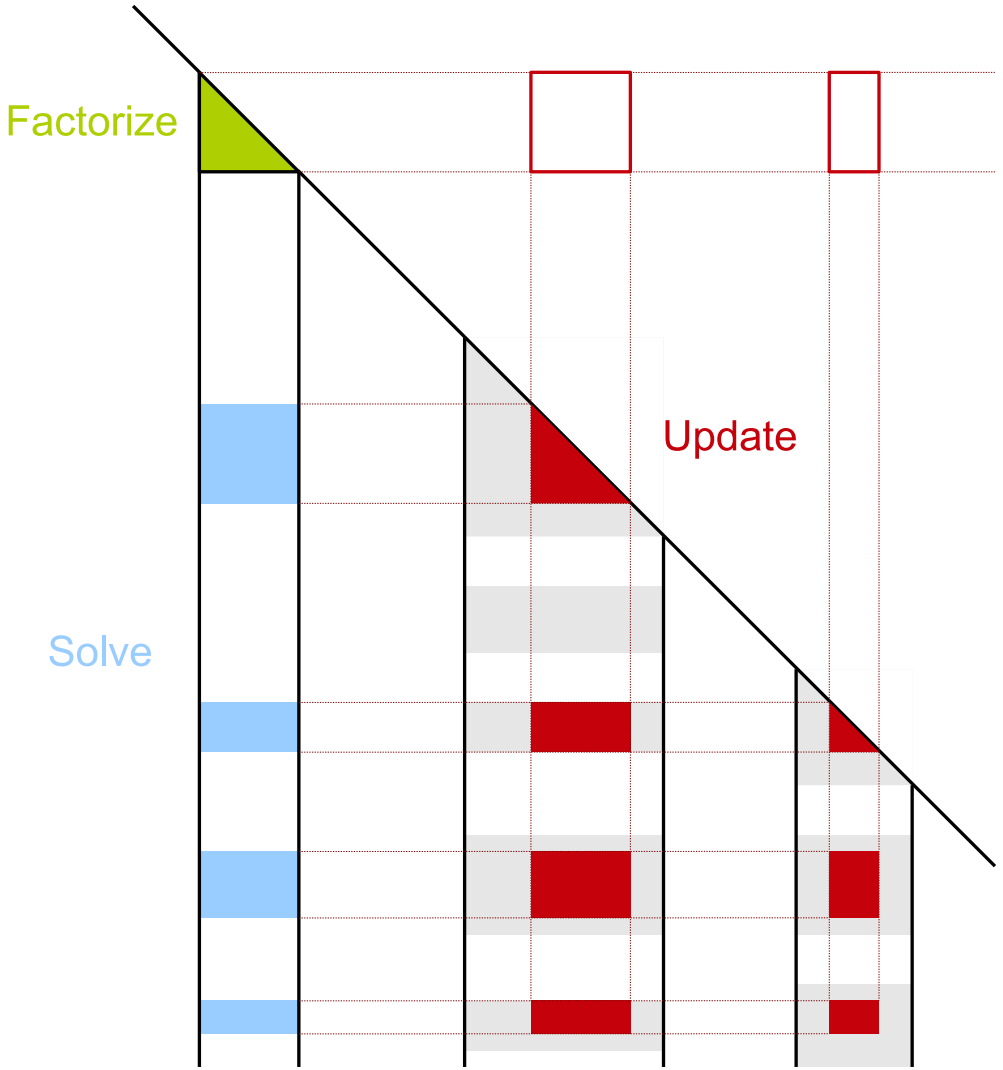
PaStiX Features

- LLt, LDLt, LU : supernodal implementation (BLAS3)
- Static pivoting + Refinement: CG/GMRES/BiCGstab
- 1D/2D block distribution
- Simple/Double precision + Float/Complex operations
- **MPI/Threads (Cluster/Multicore/SMP/NUMA)**
- **Multiple GPUs using DAG runtimes**
- Support external ordering library (PT-Scotch/METIS)
- Multiple RHS (direct factorization)
- Incomplete factorization with ILU(k) preconditionner
- Schur computation
- Out-of Core implementation (shared memory only)

Current works

- with Astrid Casadei (PhD student) : memory optimization to build a Schur complement in PaStiX tight coupling between sparse direct and iterative solvers (HIPS)
- with Xavier Lacoste (PhD student) and the MUMPS team : GPU optimizations for sparse factorizations with StarPU
- with Mathieu Faverge (Assistant Professor) : **redesigned** PaStiX static/dynamic scheduling with PaRSEC (George Bosilca's team) in order to get a generic framework for multicore/MPI/GPU/Out-of-Core + **compression**

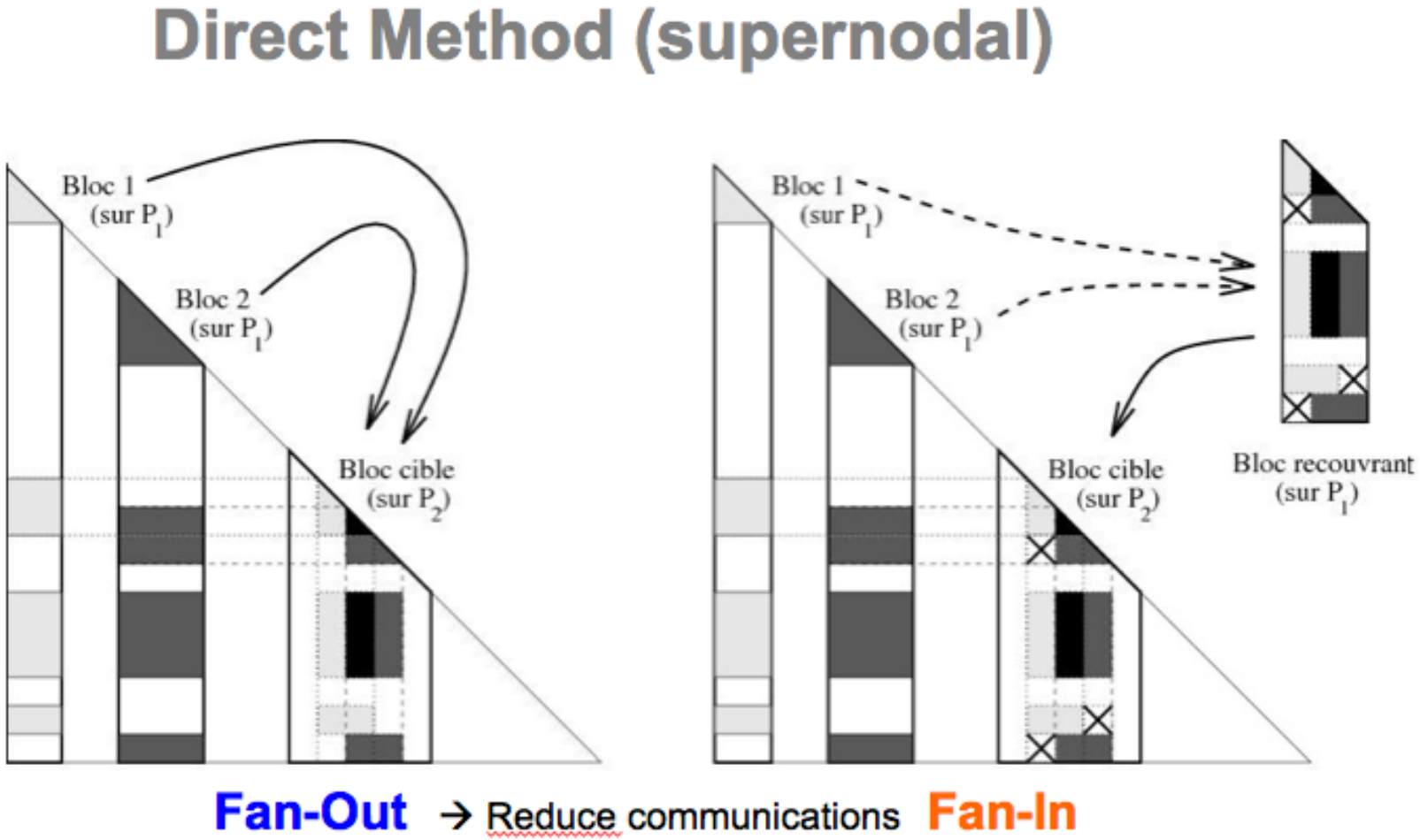
Supernodal factorization tasks



Static scheduling within PaStiX

forall the *Supernode* S_1 *attributed to* t **do**
  `wait` (S_1);
  `factorize` (S_1);
  forall the *off diagonal blocks* B_i *of* S_1 **do**
    $S_2 \leftarrow$ `supernode_in_front_of` (B_i);
    `lock` (S_2);
    `update` (S_1, S_2);
    `unlock` (S_2);
    if *All updates applied on* S_2 **then**
      `release` (S_2)
    end
  end
end

Communication graph for supernodal factorization



Direct Solver Highlights (multicore)

SGI 160-cores

Name	**N**	**NNZ_A**	**Fill ratio**	**Fact**
Audi	9.44×10^5	3.93×10^7	31.28	float LL^T
10M	1.04×10^7	8.91×10^7	75.66	complex LDL^T

10M	10	20	40	80	160
Facto (s)	3020	1750	654	356	260
Mem (Gb)	122	124	127	133	146
Solve (s)	24.6	13.5	3.87	2.90	2.89

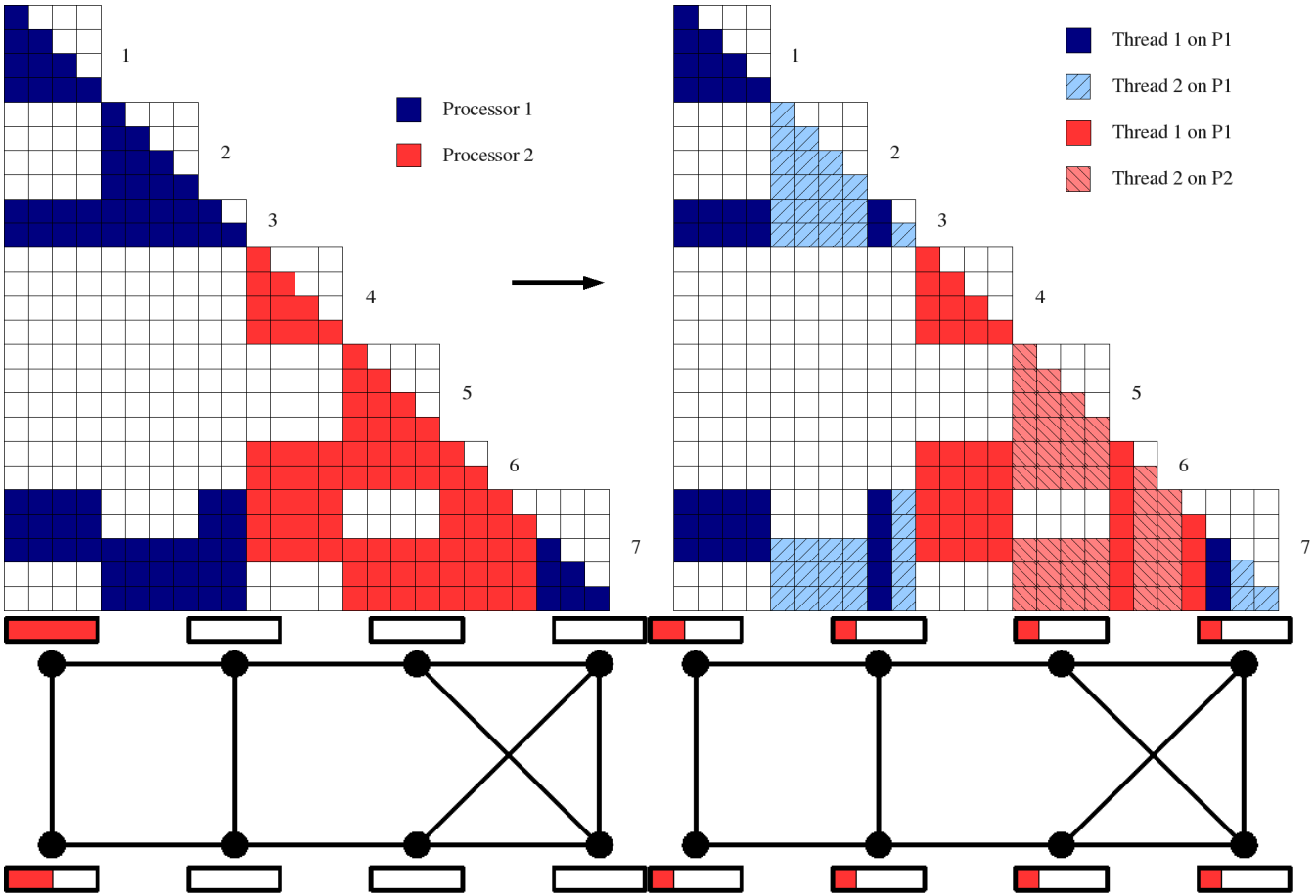
Audi	128	2x64	4x32	8x16
Facto (s)	17.8	18.6	13.8	**13.4**
Mem (Gb)	**13.4**	2x7.68	4x4.54	8x2.69
Solve (s)	0.40	0.32	0.21	**0.14**

Direct Solver Highlights (cluster of multicore)

RC3 matrix - complex double precision
N=730700 - NNA=41600758 - Fill-in=50

Facto	1 MPI	2 MPI	4 MPI	8 MPI
1 thread	6820	3520	1900	1890
6 threads	1020	639	337	287
12 threads	525	360	155	121
Mem Gb	1 MPI	2 MPI	4 MPI	8 MPI
1 thread	34	19,2	12,5	9,22
6 threads	34,3	19,5	12,8	9,66
12 threads	34,6	19,7	13	9,14
Solve	1 MPI	2 MPI	4 MPI	8 MPI
1 thread	6,97	3,75	1,93	1,03
6 threads	2,5	1,43	0,78	0,54
12 threads	1,33	0,93	0,66	0,59

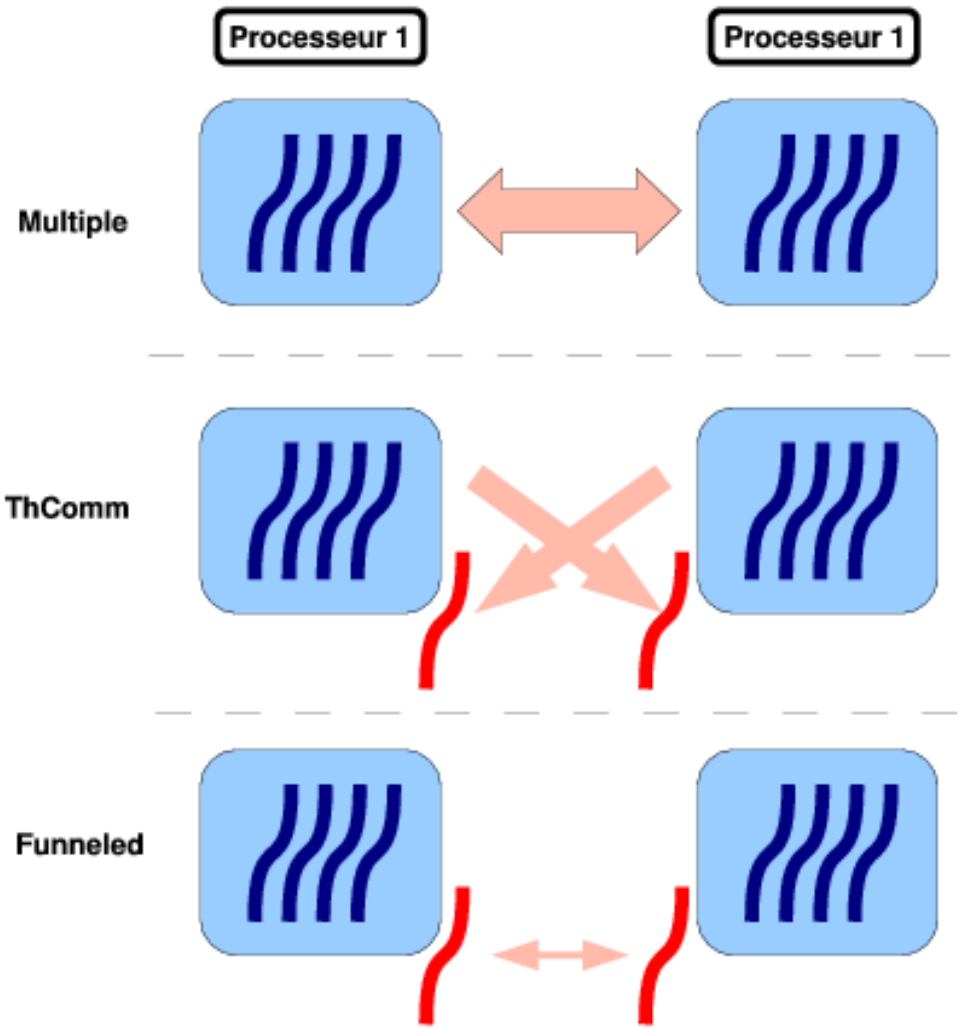
NUMA-Aware Allocation (upto 20% efficiency)



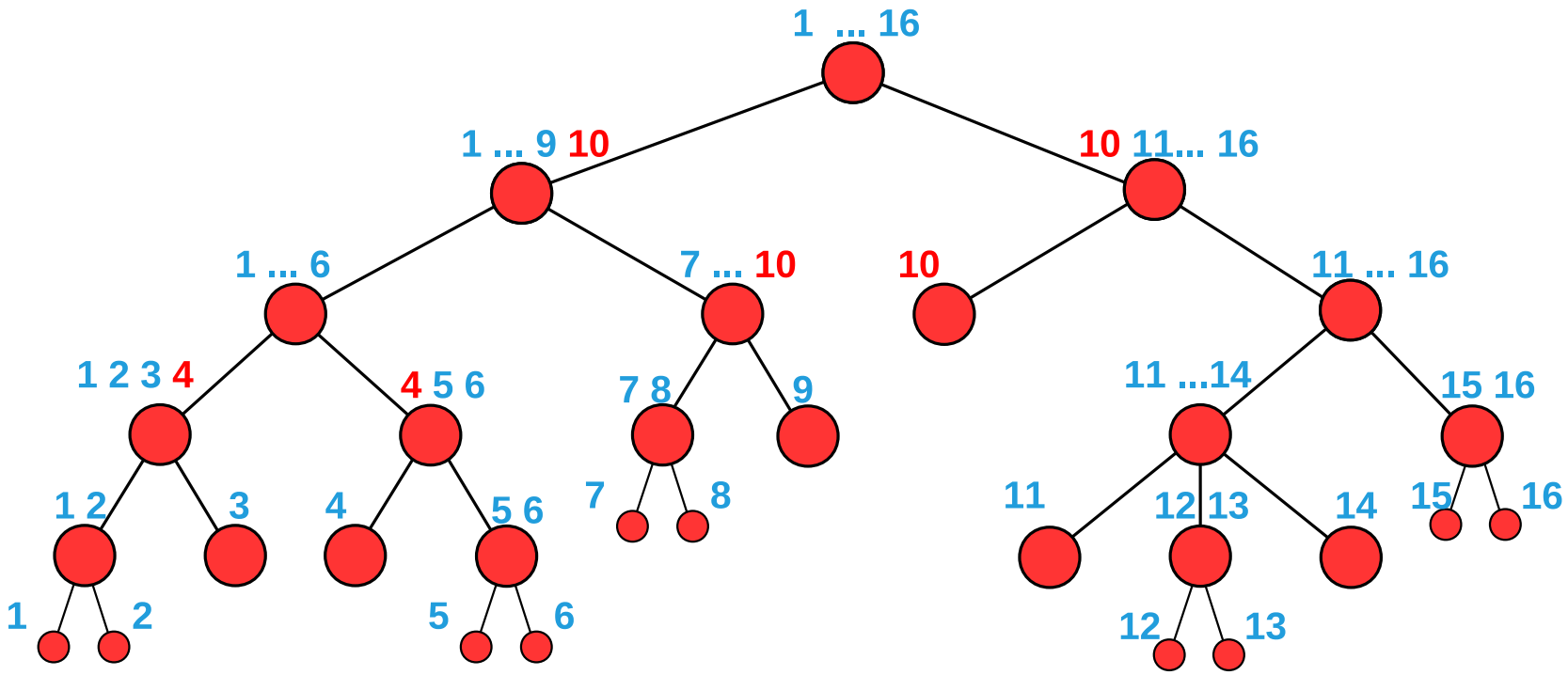
Thread support inside MPI librairies

- `MPI_THREAD_SINGLE`
  - Only one thread will execute.
- `MPI_THREAD_FUNNELED`
  - The process may be multi-threaded, but only the main thread will make MPI calls
    (all MPI calls are funneled to the main thread).
- `MPI_THREAD_SERIALIZED`
  - The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads
    (all MPI calls are serialized).
- `MPI_THREAD_MULTIPLE`
  - Multiple threads may call MPI, with no restrictions.

Communication schemes (upto 10% efficiency)

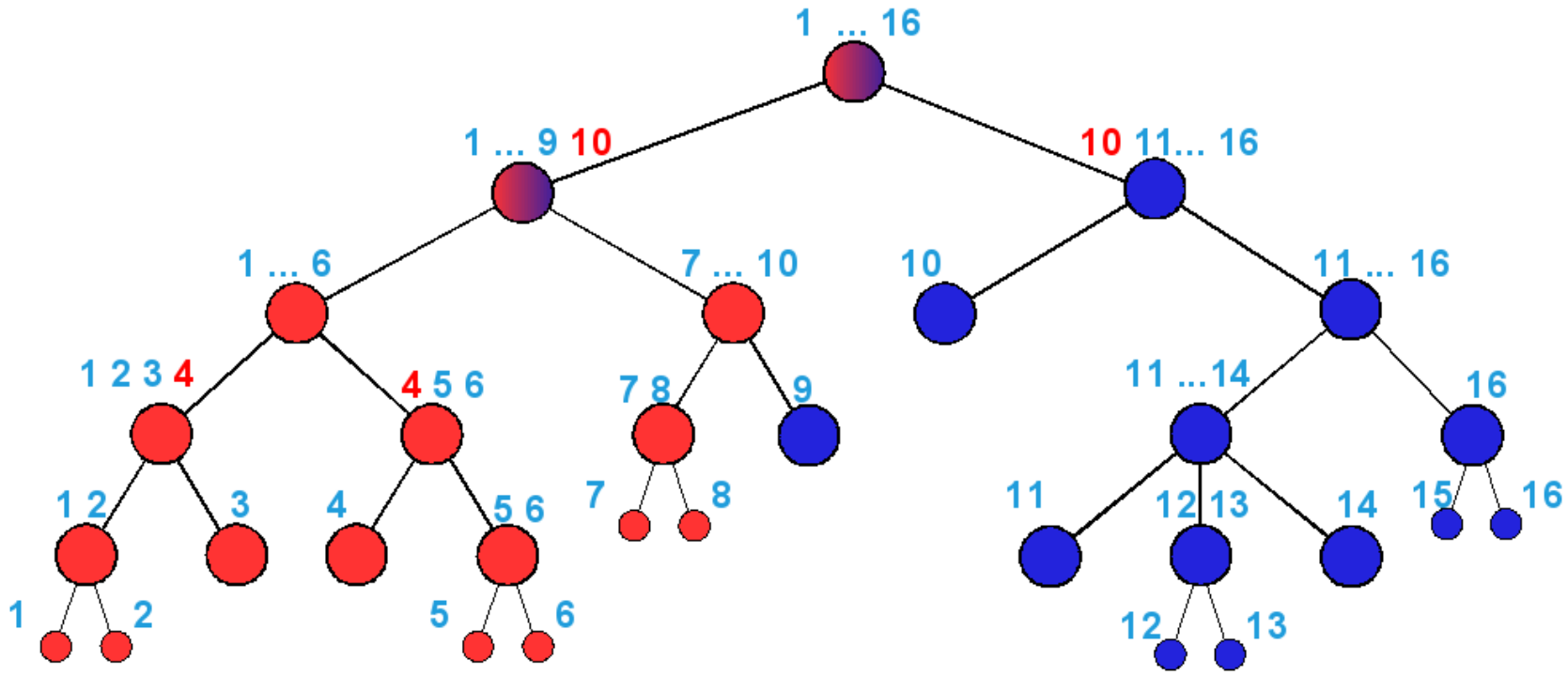


Proportional Mapping [A. Pothen, C. Sun 93]



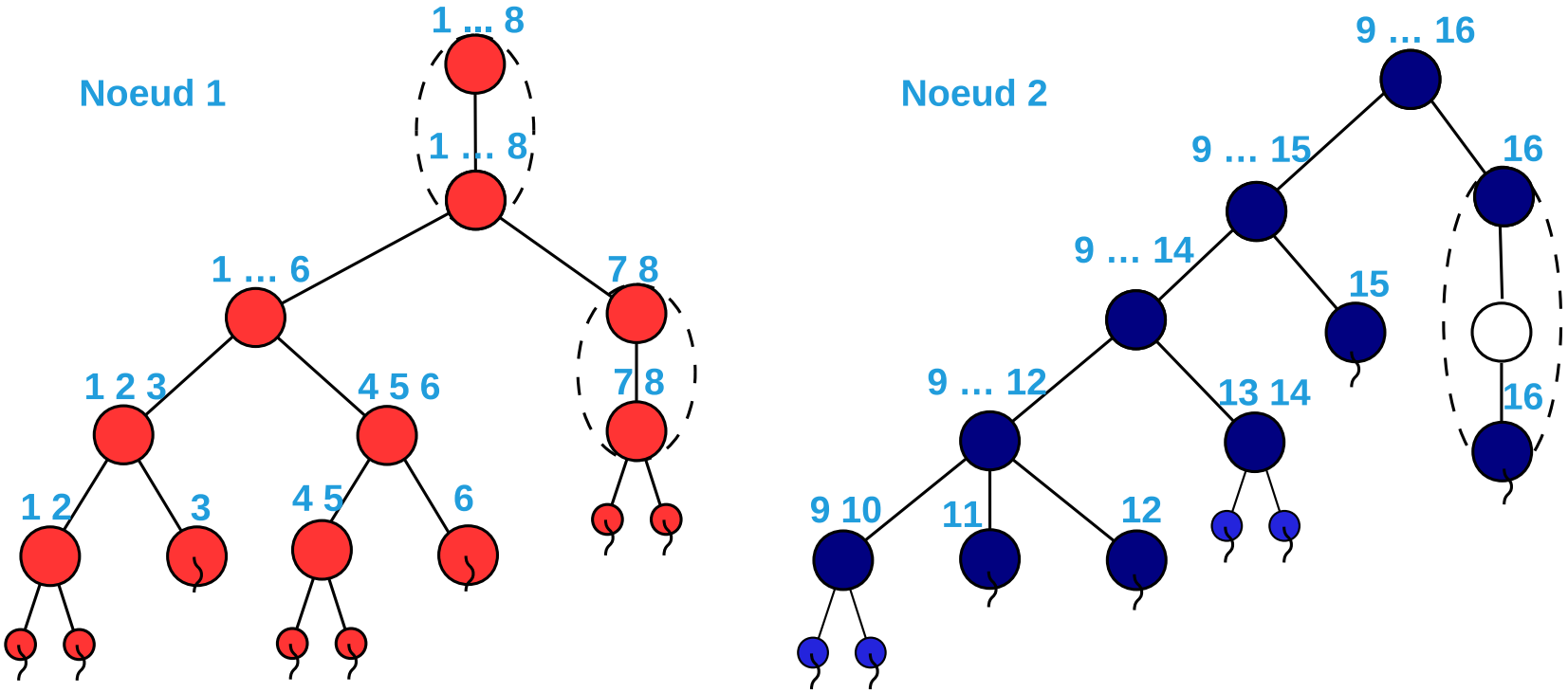
- Top-down strategy to build candidate processor groups for each block (ensure good locality of communications)
- Down-top mapping induced by a logical simulation of computations of the block solver (models for factorization time, communication and aggregation)

Dynamic Scheduling : New Mapping



- Need to map data on MPI process
- Two steps :
  - **A first proportional mapping step to map data**
  - A second step to build a file structure for the work stealing algorithm

Dynamic Scheduling : New Mapping



- Need to map data on MPI process
- Two steps :
  - A first proportional mapping step to map data
  - **A second step to build a file structure for the work stealing algorithm**

Study on a large test case: 10M

Properties

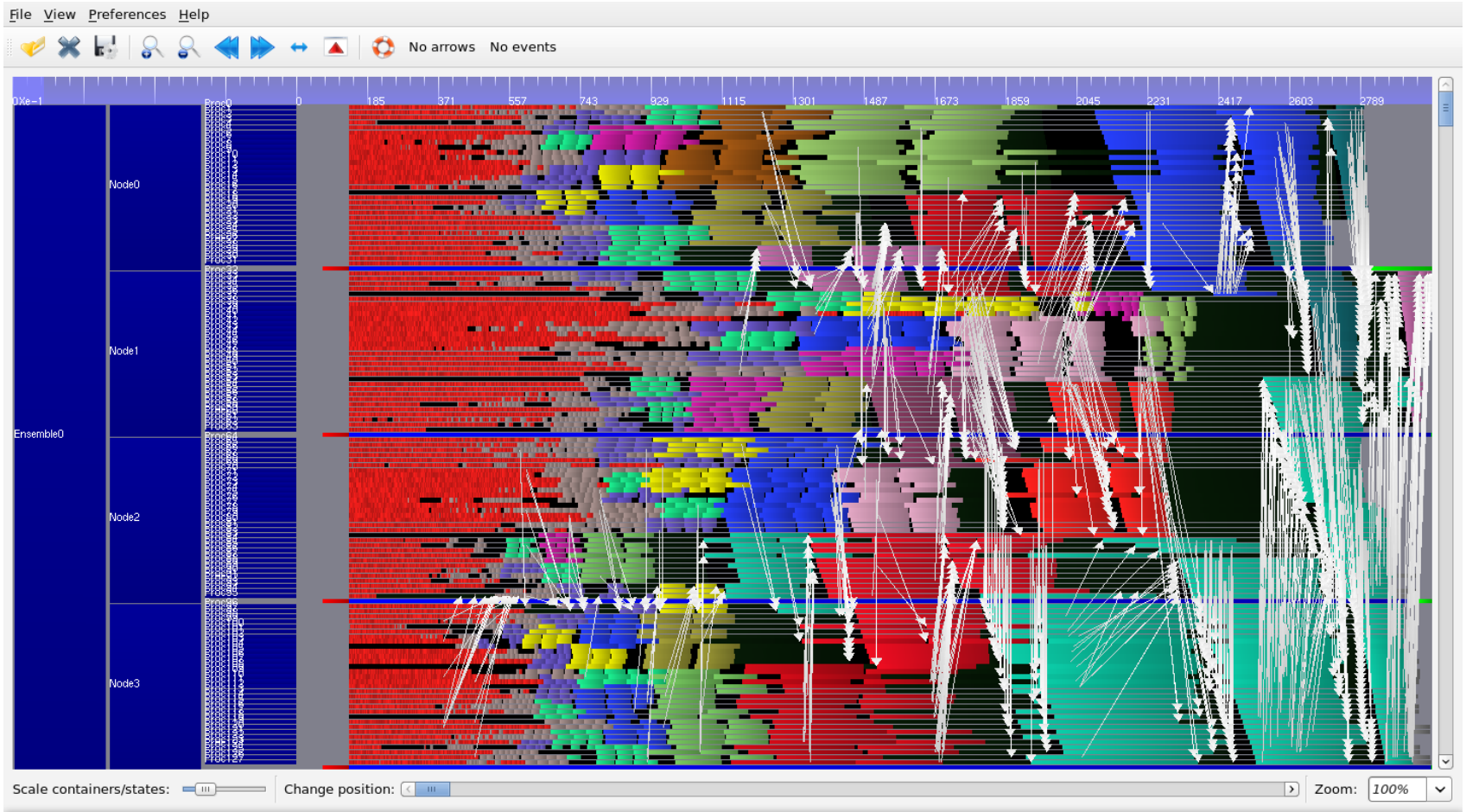
N	10 423 737
NNZ_A	89 072 871
NNZ_L	6 724 303 039
OPC	4.41834e+13

	4x32	8x32
Static Scheduler	289	195
Dynamic Scheduler	240	184

Table: Factorization time in seconds

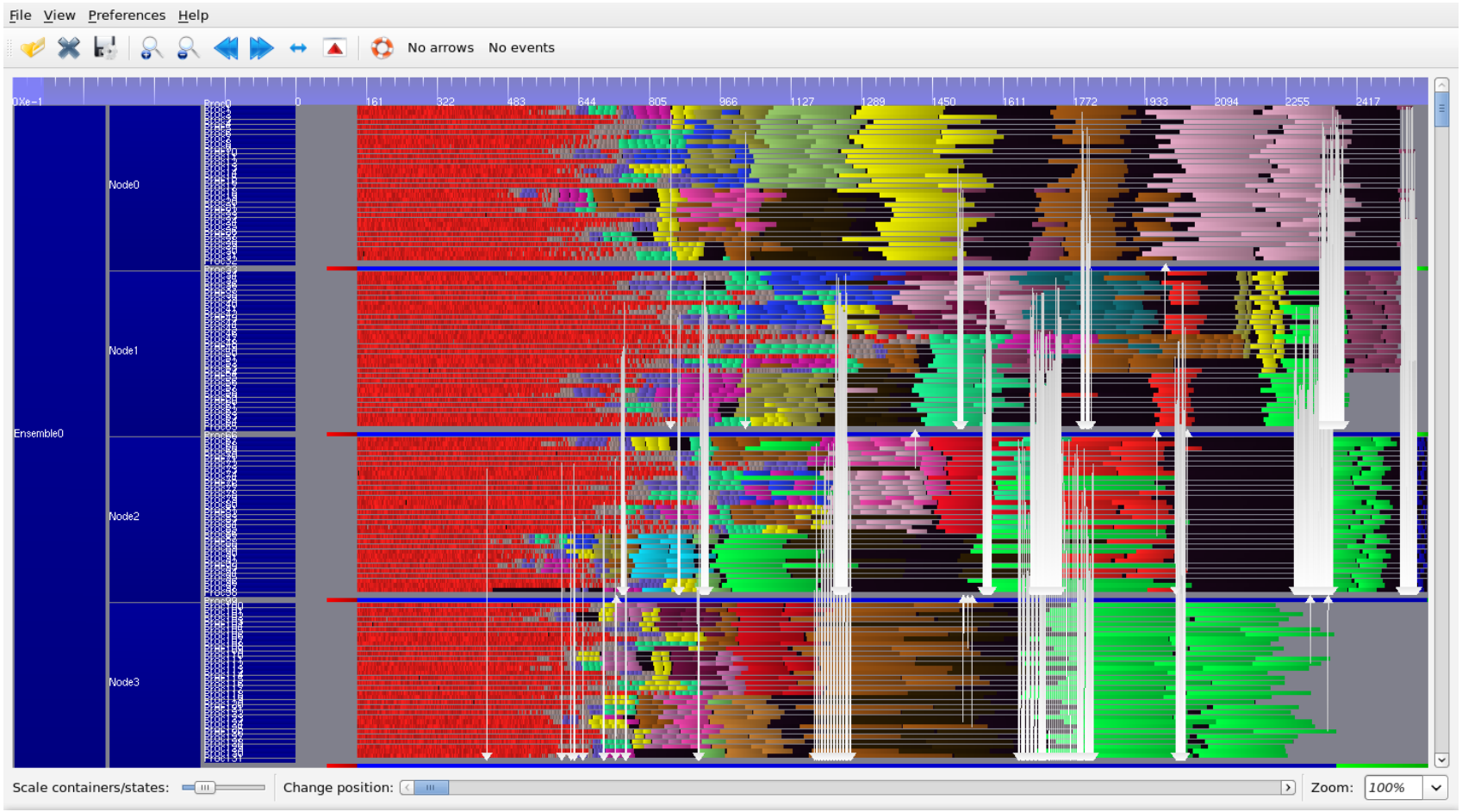
- Electromagnetism problem in double complex from CEA CESTA
- Cluster Vargas from IDRIS with 32 power6 per node

Static Scheduling Gantt Diagram



- *10Million* test case on IDRIS IBM Power6 with 4 MPI process of 32 threads (color is level in the tree)

Dynamic Scheduling Gantt Diagram



- Reduces time by 10-15%

Block ILU(k): supernode amalgamation algorithm

Derive a block incomplete LU factorization from the supernodal parallel direct solver

- Based on existing package PaStiX
- Level-3 BLAS incomplete factorization implementation
- Fill-in strategy based on level-fill among block structures identified thanks to the quotient graph
- **Amalgamation strategy to enlarge block size**

Highlights

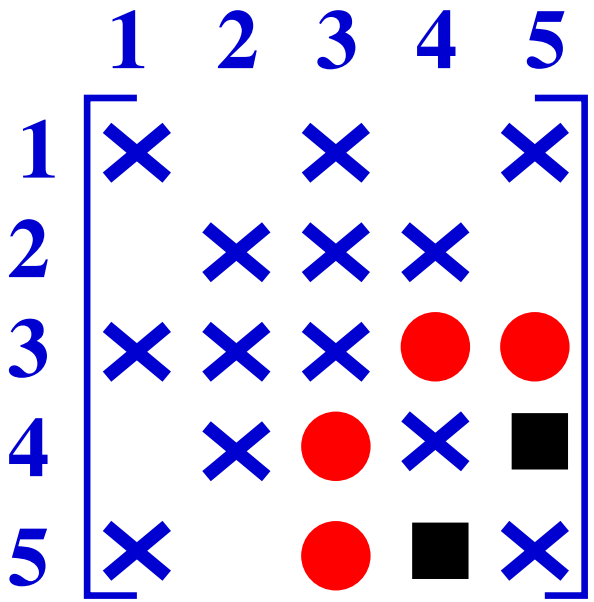
- Handles efficiently high level-of-fill
- Solving time faster than with scalar ILU(k)
- Scalable parallel implementation

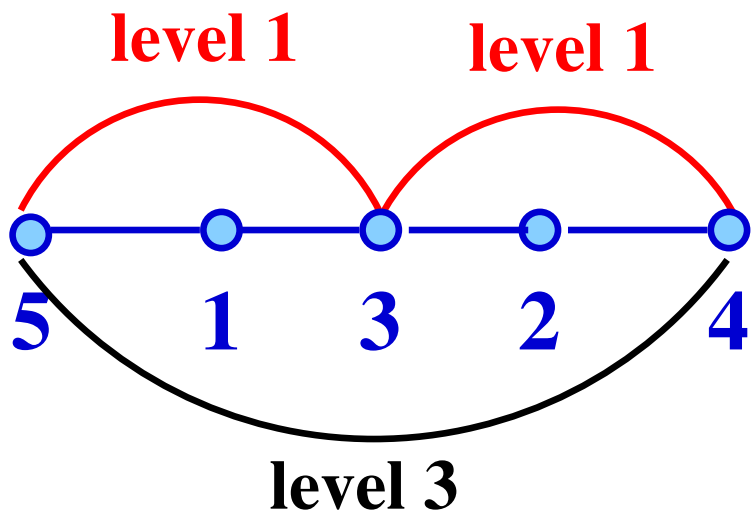
Fill-in theorem

Theorem

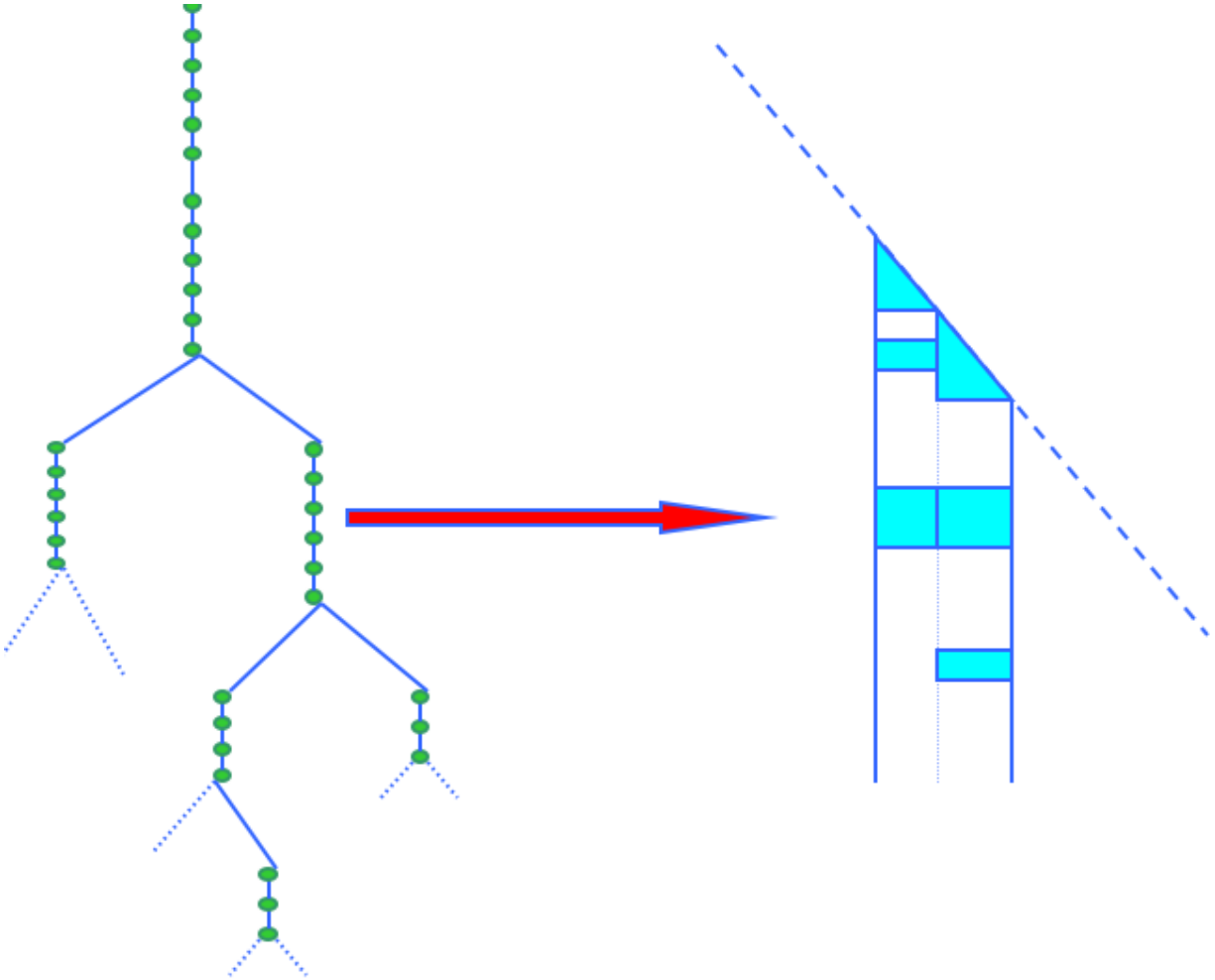
Any $A_{ij} = 0$ will become a non-null entry L_{ij} or $U_{ij} \neq 0$ in $A = iLU(k)$ if and only if it exists a shortest path of lengh $k+1$ in $G_A(V, E)$ from vertex i to vertex j that only goes through vertices with a lower number than i and j.

Factorization ILU(k)



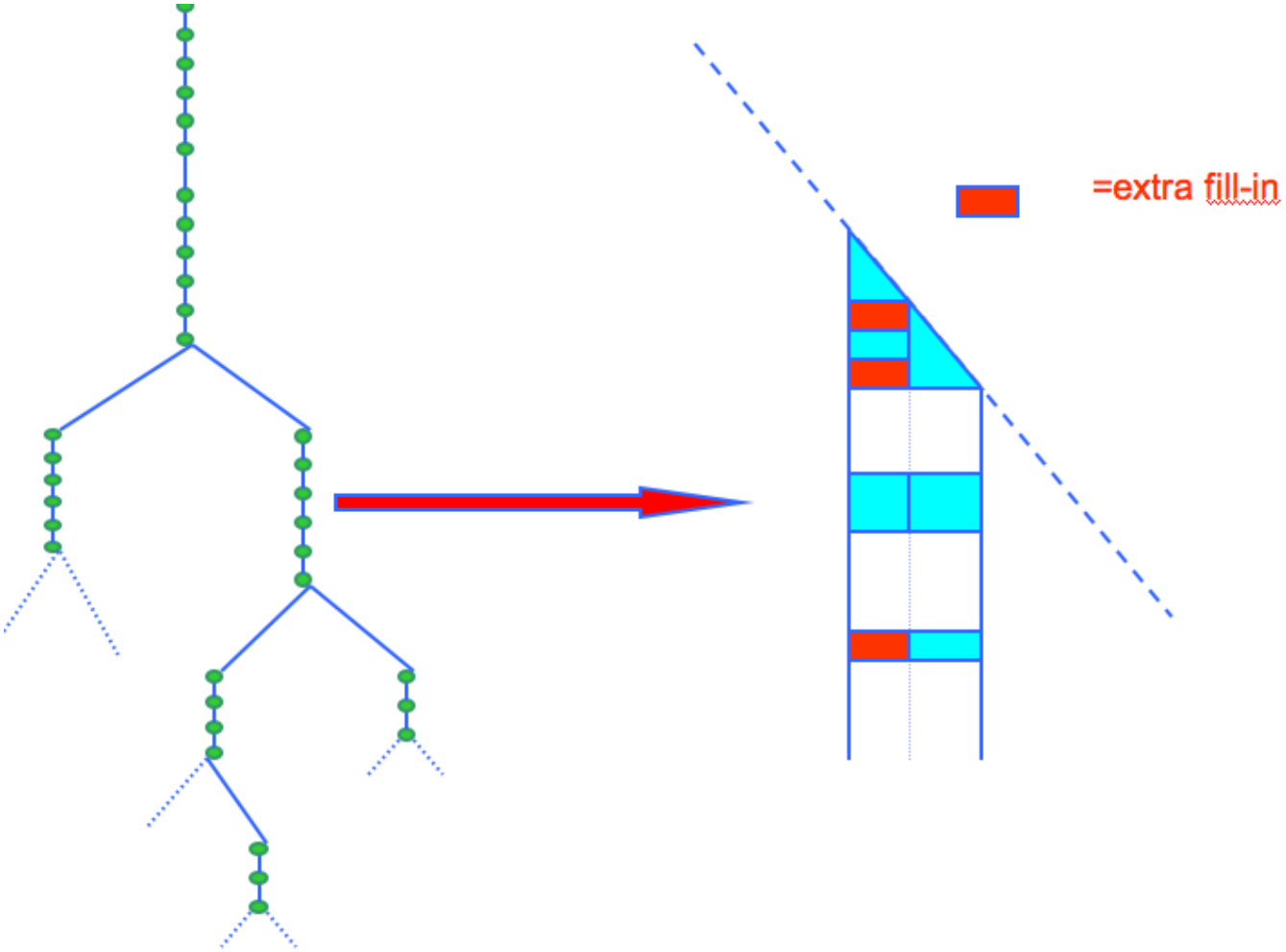


BILU(k): the amalgamation strategy



Find the exact supernode partition

BILU(k): the amalgamation strategy



Merge the couple of supernodes that add the less extra fill-in

3
Hybrid methods

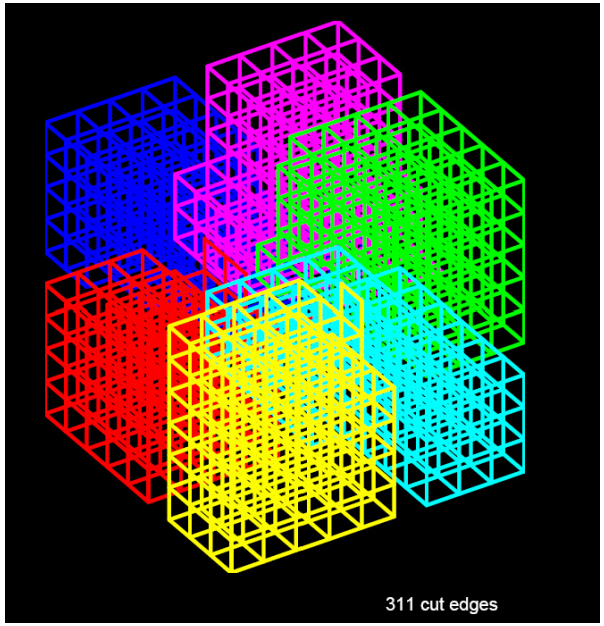
Hybrid Linear Solvers

Develop robust scalable parallel hybrid direct/iterative linear solvers

- Exploit the efficiency and robustness of the sparse direct solvers
- Develop robust parallel preconditioners for iterative solvers
- Take advantage of scalable implementation of iterative solvers

Domain Decomposition (DD)

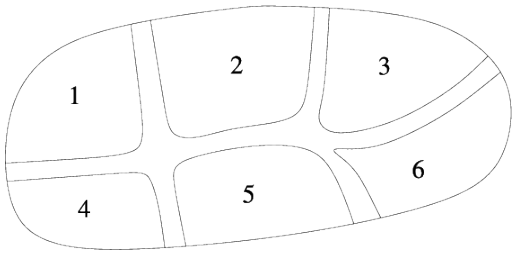
- Natural approach for PDE's
- Extend to general sparse matrices
- Partition the problem into subdomains
- Use a direct solver on the subdomains
- Robust preconditioned iterative solver



HIPS : hybrid direct-iterative solver

Based on a **domain decomposition** : interface one node-wide (no overlap in DD lingo)

$$\begin{pmatrix} A_B & F \\ E & A_C \end{pmatrix}$$

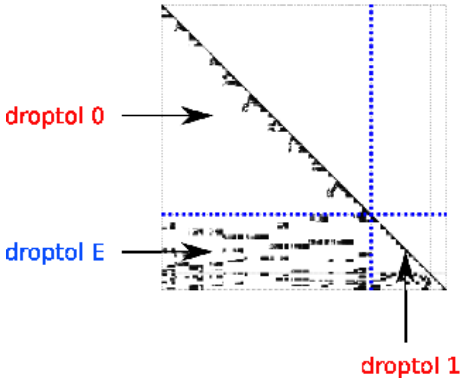


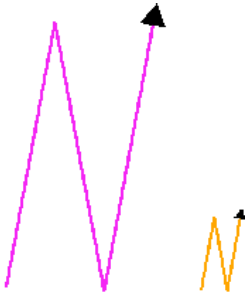
B : Interior nodes of subdomains (direct factorization).

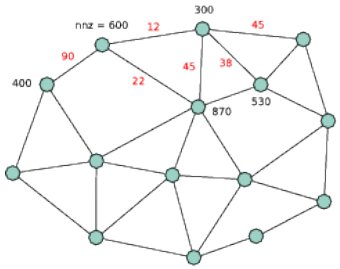
C : Interface nodes.

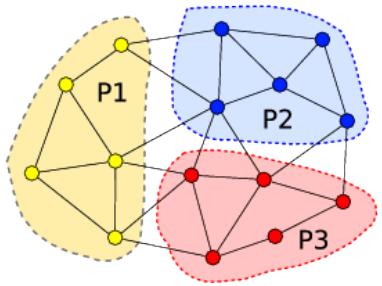
Special decomposition and ordering of the subset C :
Goal : Building a global Schur complement preconditioner (ILU) from the local domain matrices only.

HIPS: preconditioners









Main features

- Iterative or "hybrid" direct/iterative method are implemented.
- Mix direct supernodal (BLAS-3) and sparse ILUT factorization in a seamless manner.
- Memory/load balancing : distribute the domains on the processors (domains $>$ processors).

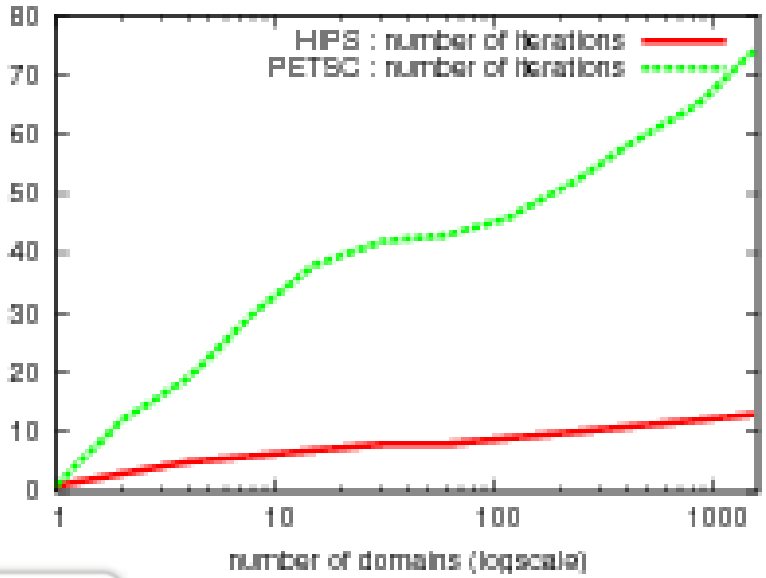
HIPS vs Additive Schwarz (from PETSc)

Experimental conditions

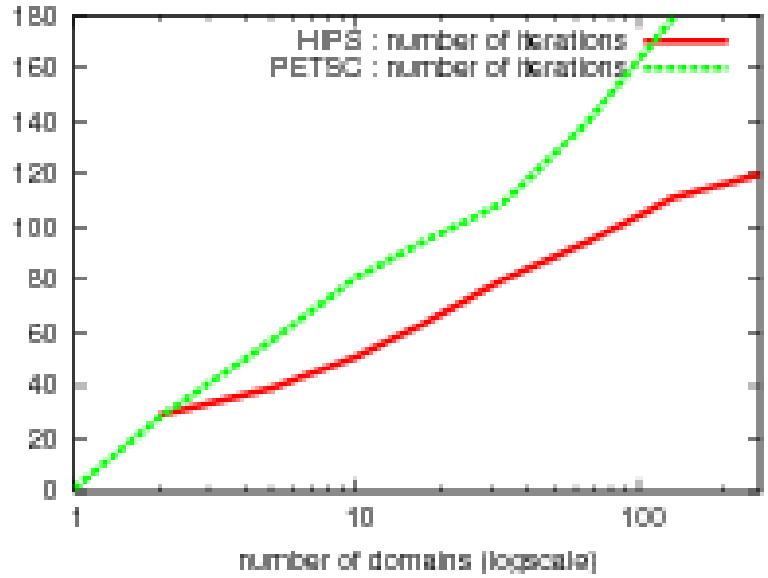
These curves compare HIPS (Hybrid) with Additive Schwarz from PETSc.

Parameters were tuned to compare the result with a very similar fill-in

Haltere

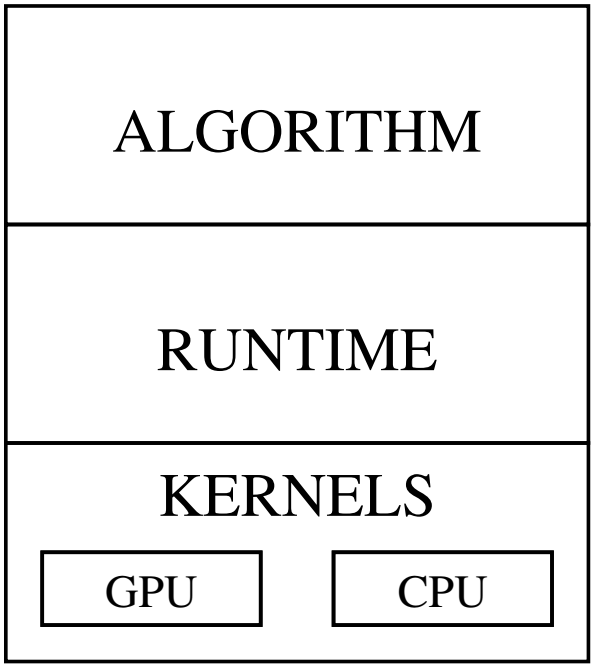


MHD



4
Towards sparse factorization on manycore

Multiple layer approach



Governing ideas: Enable advanced numerical algorithms to be executed on a scalable unified runtime system for exploiting the full potential of future exascale machines.

Basics:

- Graph of tasks
- Out-of-order scheduling
- Fine granularity

DAG schedulers considered

StarPU

- RunTime Team – Inria Bordeaux Sud-Ouest
- C. Augonnet, R. Namyst, S. Thibault.
- Dynamic Task Discovery
- Computes cost models on the fly
- Multiple kernels on the accelerators
- Heterogeneous First-Time strategy

PaRSEC (formerly DAGuE)

- ICL – University of Tennessee, Knoxville
- G. Bosilca, A. Bouteiller, A. Danalys, T. Herault
- Parameterized Task Graph
- Only the most compute intensive kernel on accelerators
- Simple scheduling strategy based on computing capabilities
- GPU multi-stream enabled

StarPU loop to submit tasks (DTD)

forall the *Supernode* S_1 **do**
  `submit_factorize` (S_1);
  forall the *off diagonal blocks* B_i *of* S_1 **do**
    $S_2 \leftarrow$ `supernode_in_front_of` (B_i);
    `submit_update` (S_1, S_2);
  end
end

PaRSEC's representation (PTG)

```
panel(j) [high_priority = on]
/* execution space */
c = 0 .. cblknbr-1
/* Extra parameters */
firstblock = diagonal_block_of( c )
lastblock = last_block_of( c )
lastbrow = last_brow_of( c )
/* Locality */
:A(c)
RW A ← leaf ? A(c) : C update(lastbrow)
       → A update(firstblock+1..lastblock)
       → A(c)
```

```
update(j)
/* execution space */
b = 0 .. bloknbr-1
/* Extra parameters */
c = get_cblk_of( b )
fc = get_facing_cblk_of( b )
...
/* Locality */
:A(fc)
READ A ← A panel(c)
RW C ← previous ? C update(prev) : A(fc)
       → next ? C update(next) : A panel(fc)
```

Experiments

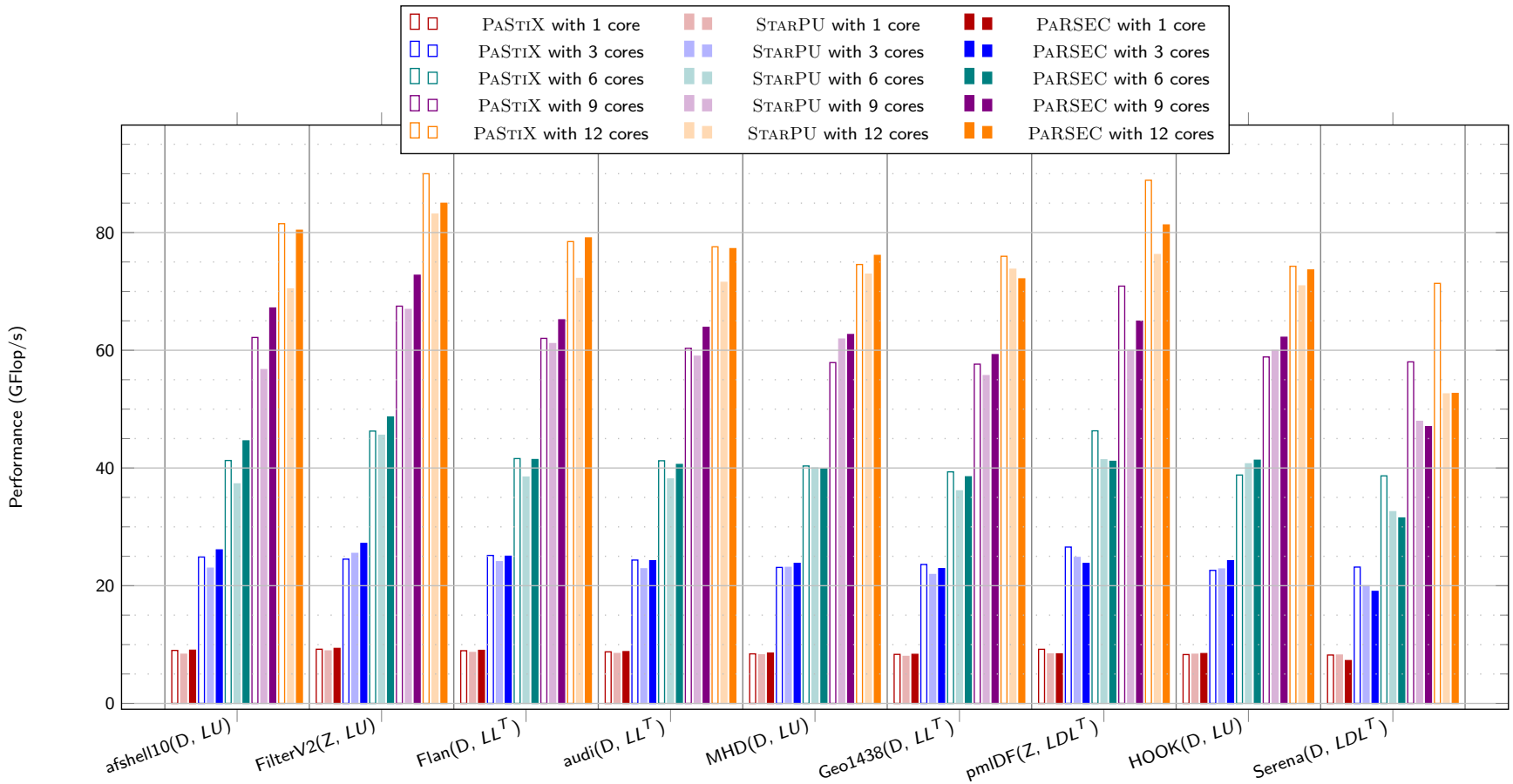
Matrix	Prec	Method	Size	nnz_A	nnz_L	TFlop/s
Afshell10	D	LU	1.5e+6	27e+6	610e+6	0.12
FilterV2	Z	LU	0.6e+6	12e+6	536e+6	3.6
Flan	D	LL^T	1.6e+6	59e+6	1712e+6	5.3
Audi	D	LL^T	0.9e+6	39e+6	1325e+6	6.5
MHD	D	LU	0.5e+6	24e+6	1133e+6	6.6
Geo1438	D	LL^T	1.4e+6	32e+6	2768e+6	23
Pmldf	Z	LDL^T	1.0e+6	8e+6	1105e+6	28
Hook	D	LU	1.5e+6	31e+6	4168e+6	35
Serena	D	LDL^T	1.4e+6	32e+6	3365e+6	47

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

Machine

- Two hexa-cores Westmere Xeon X5650 (2.67 GHz)
- 32 GB of memory

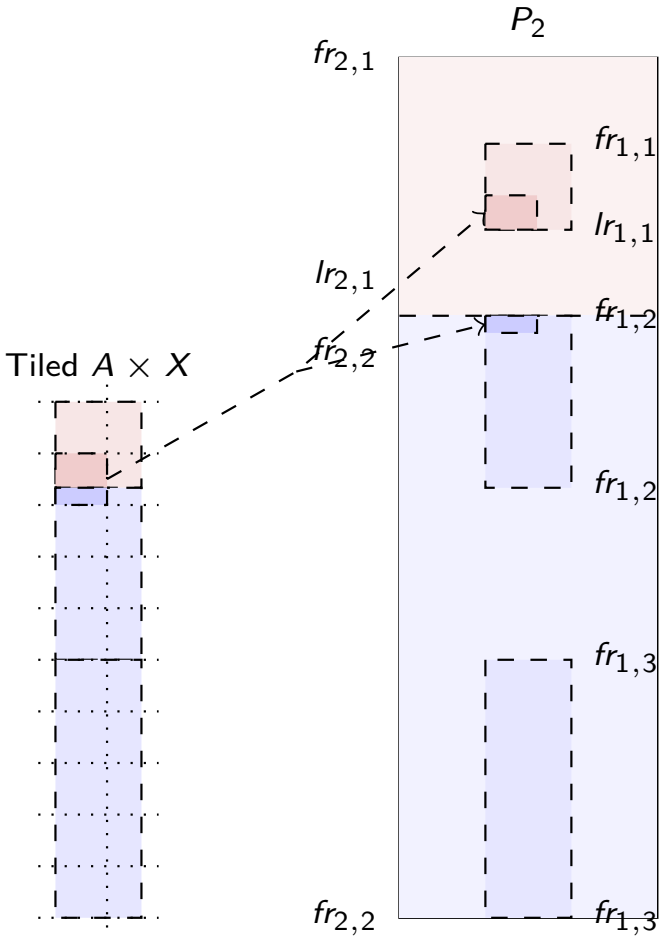
CPU Scaling study



What can be offloaded to the GPU?

- Panel factorization:
  - Call MAGMA kernel?
  - Diagonal block size < 120
  - Panel is done on CPU
  - $\rightarrow$ No GPU kernel for factorization
- Panel update:
  - GEMM variant
  - Highly efficient GEMM source code available
  - $\rightarrow$ existing GEMM can easily be adapted to our problem
  - Extension of ASTRA kernel (J. Kurzak, ICL)

Sparse GEMM on GPU



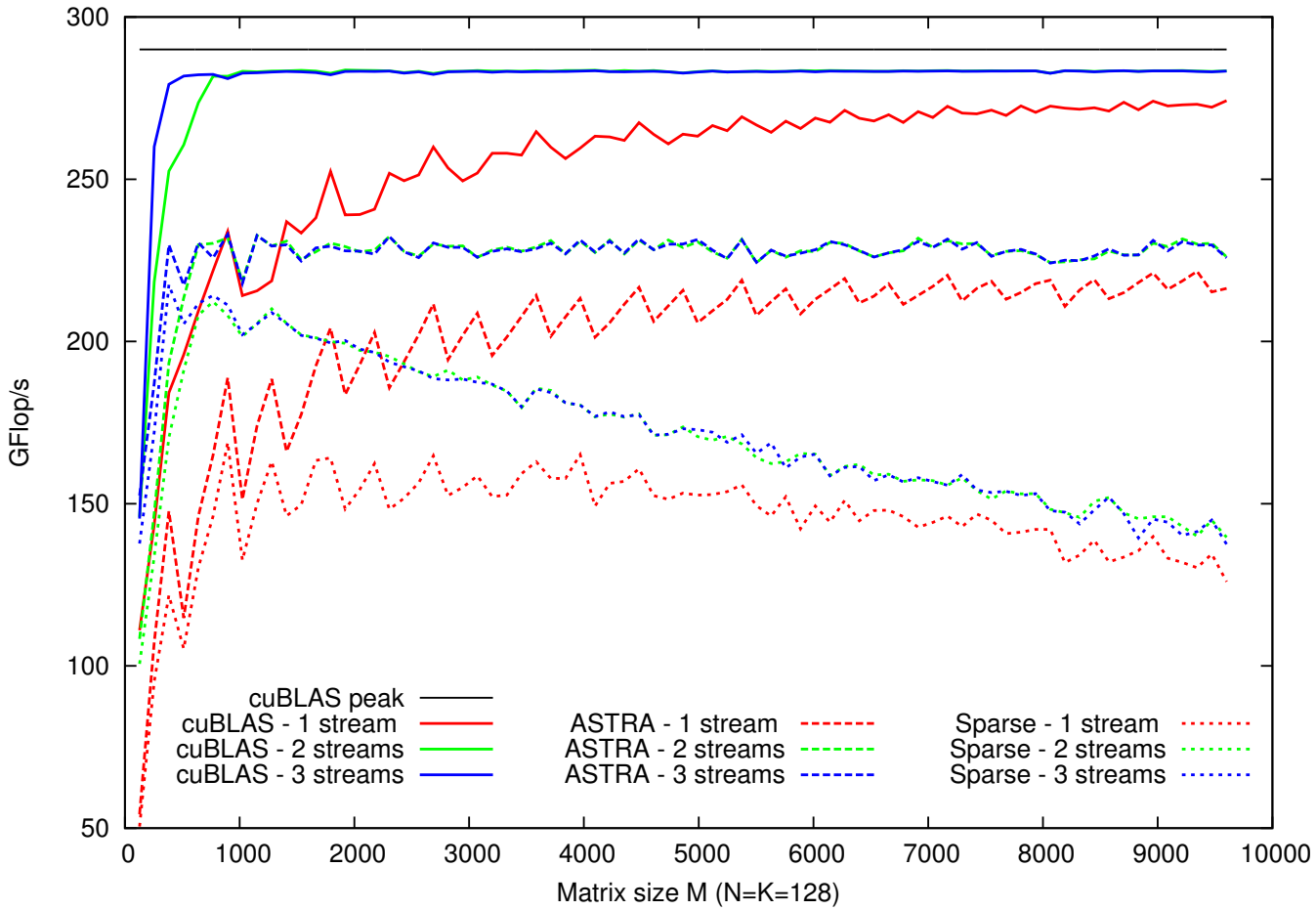
blocknbr = 3;
blocktab = [$fr_{1,1}$, $lr_{1,1}$,
$fr_{1,2}$, $lr_{1,2}$,
$fr_{1,3}$, $lr_{1,3}$];

fblocknbr = 2;
fblocktab = [$fr_{2,1}$, $lr_{2,1}$,
$fr_{2,2}$, $lr_{2,2}$];

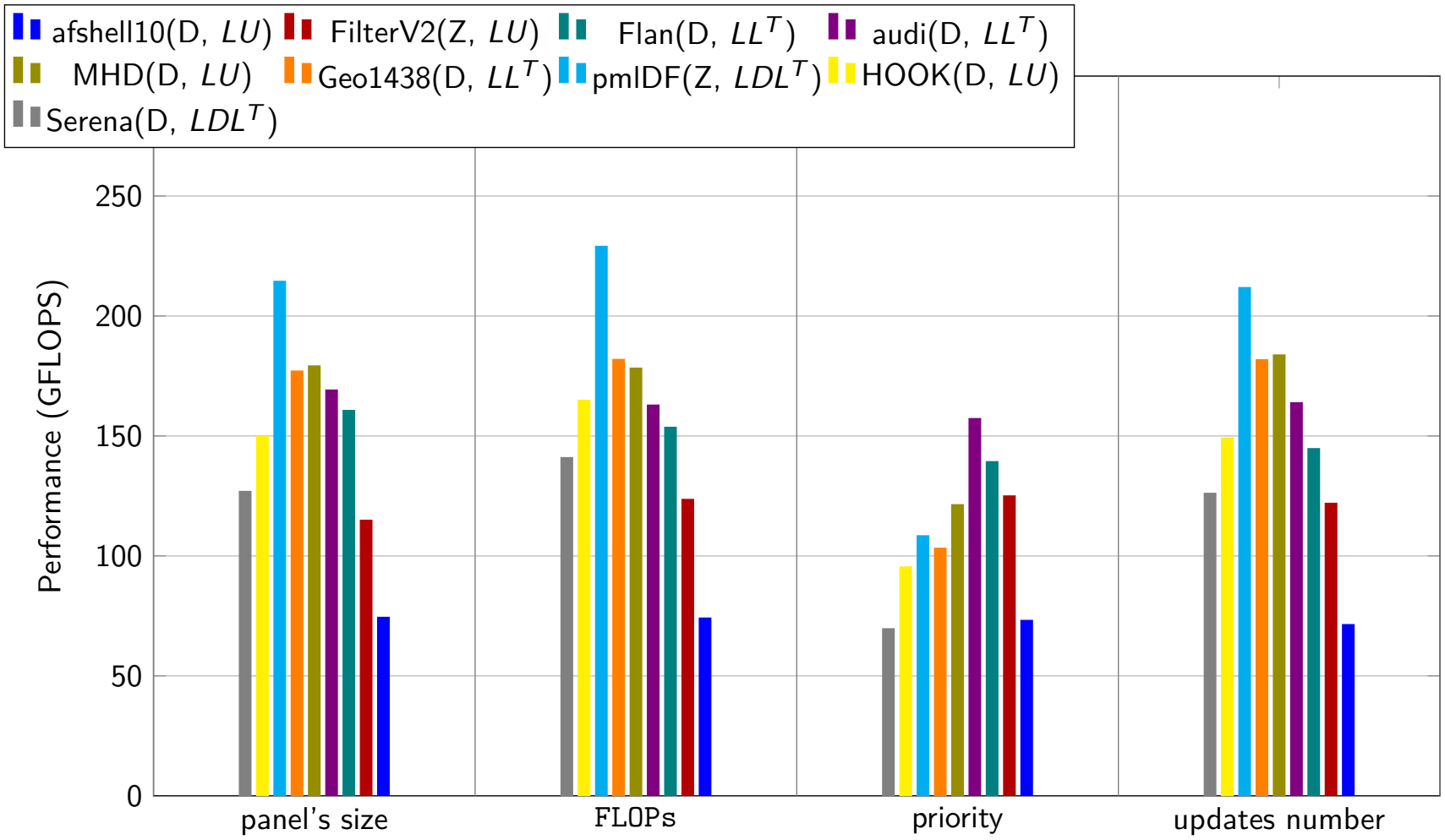
```
sparse_gemm_cuda( char TRANSA, char TRANSB, int m, int n,
                  cuDoubleComplex alpha,
                  const cuDoubleComplex *d_A, int lda,
                  const cuDoubleComplex *d_B, int ldb,
                  cuDoubleComplex beta,
                  cuDoubleComplex *d_C, int ldc,
                  int blocknbr, const int *blocktab,
                  int fblocknbr, const int *fblocktab,
                  CUstream stream );
```

Figure: Panel update on GPU

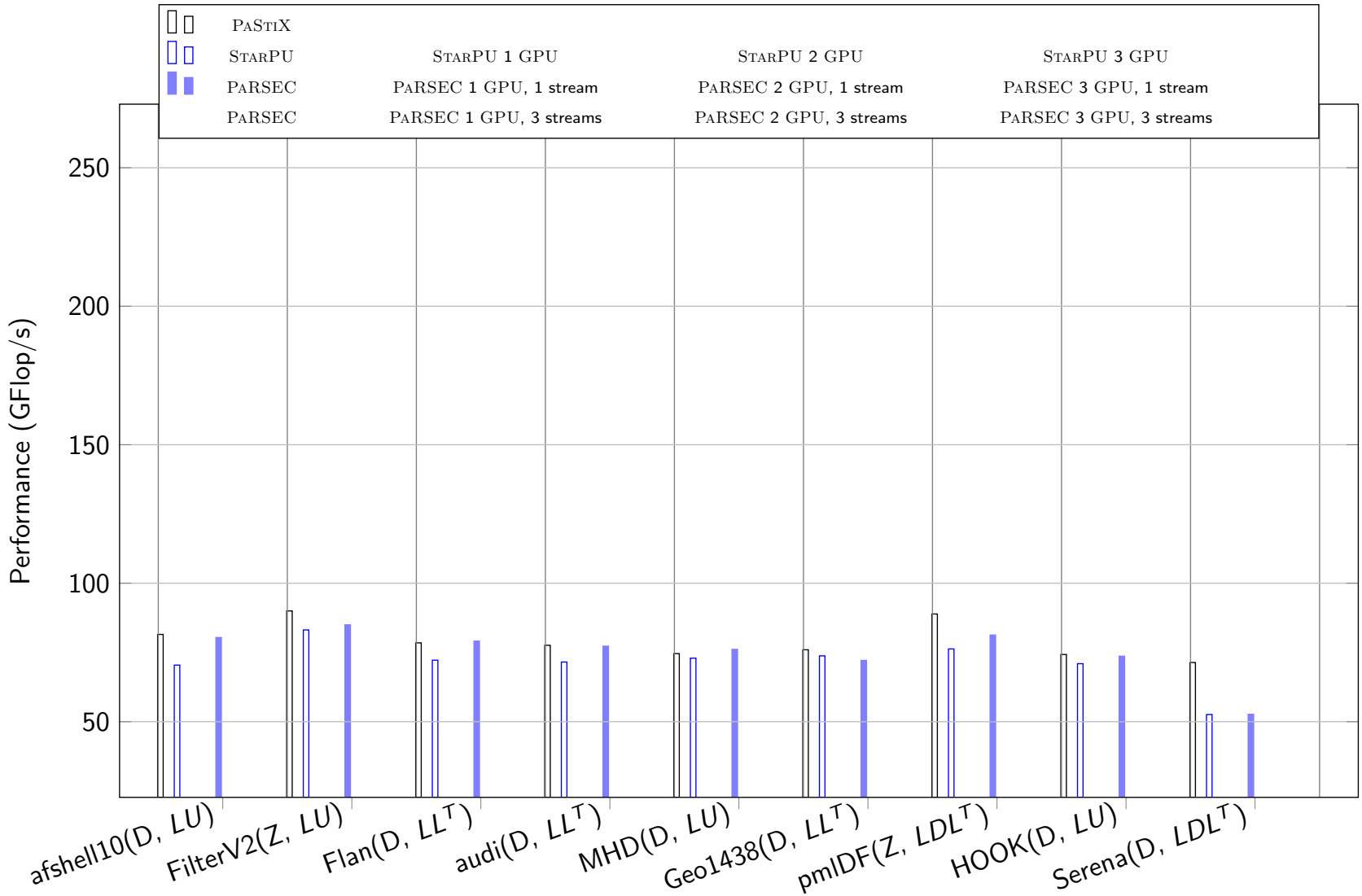
Multi-stream performance comparison (Tesla M2070)



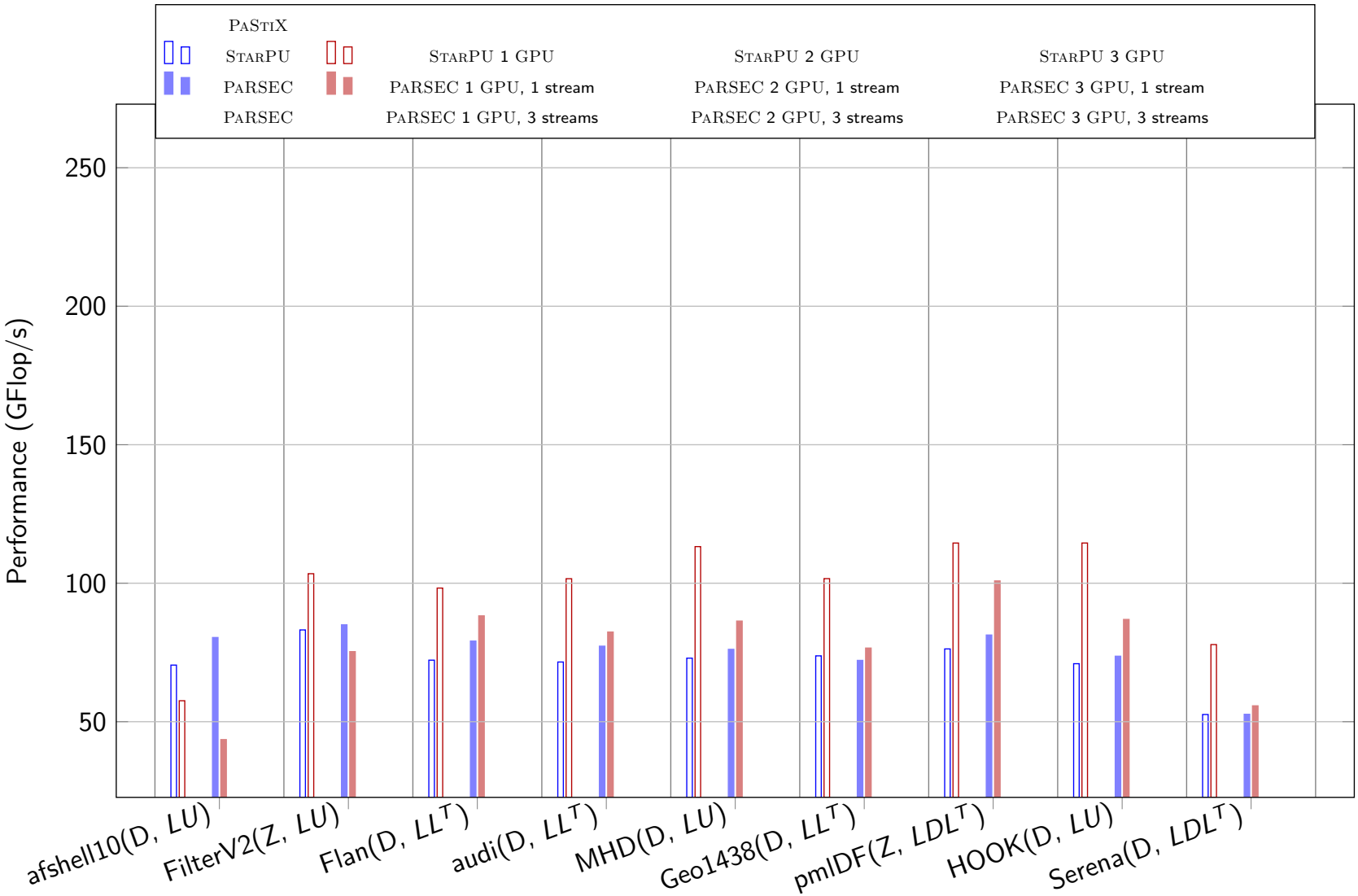
Data mapping over GPU (PaRSEC, 3 Tesla M2070)



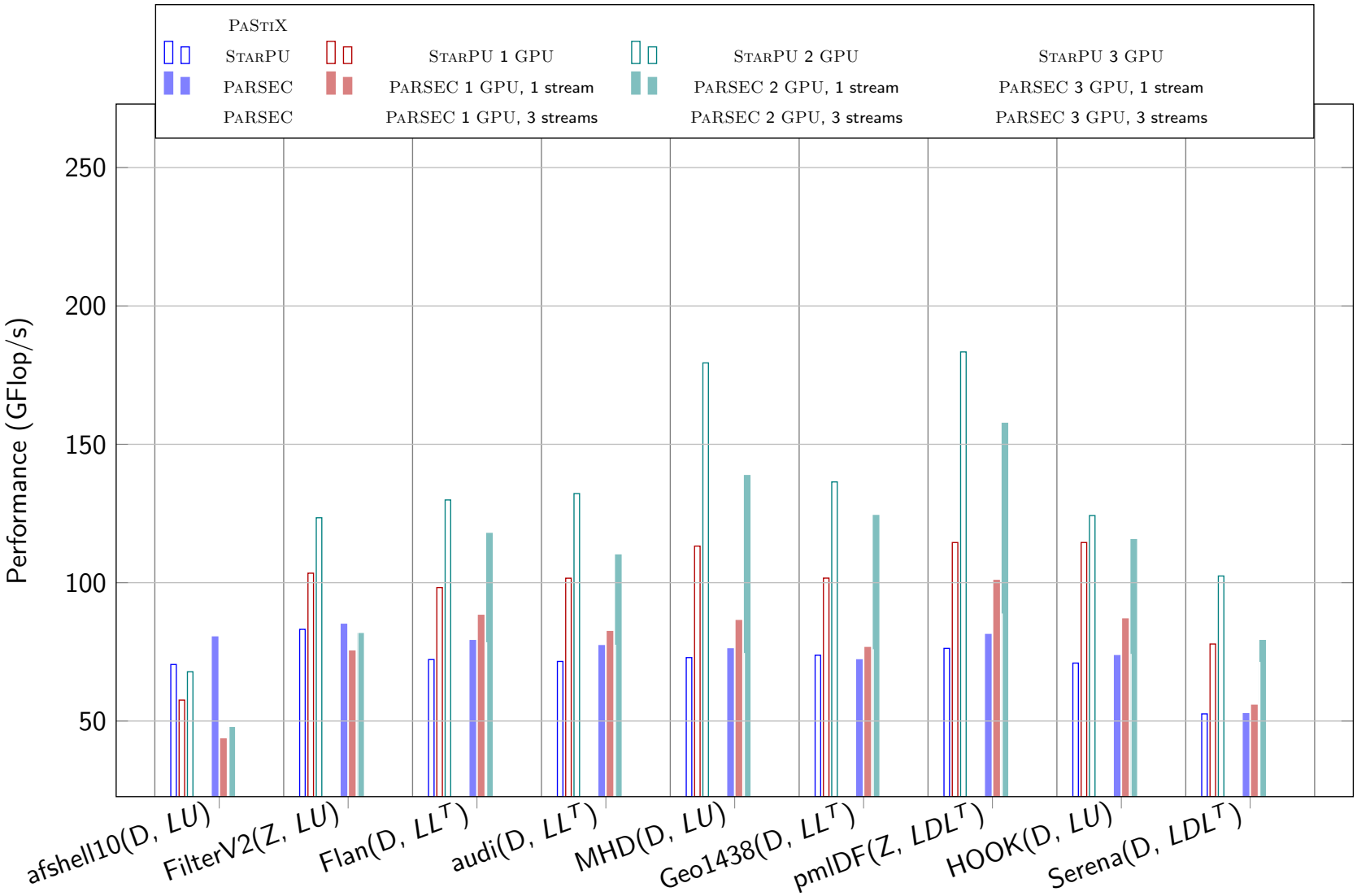
GPU scaling study (No GPU - 12 CPUs)



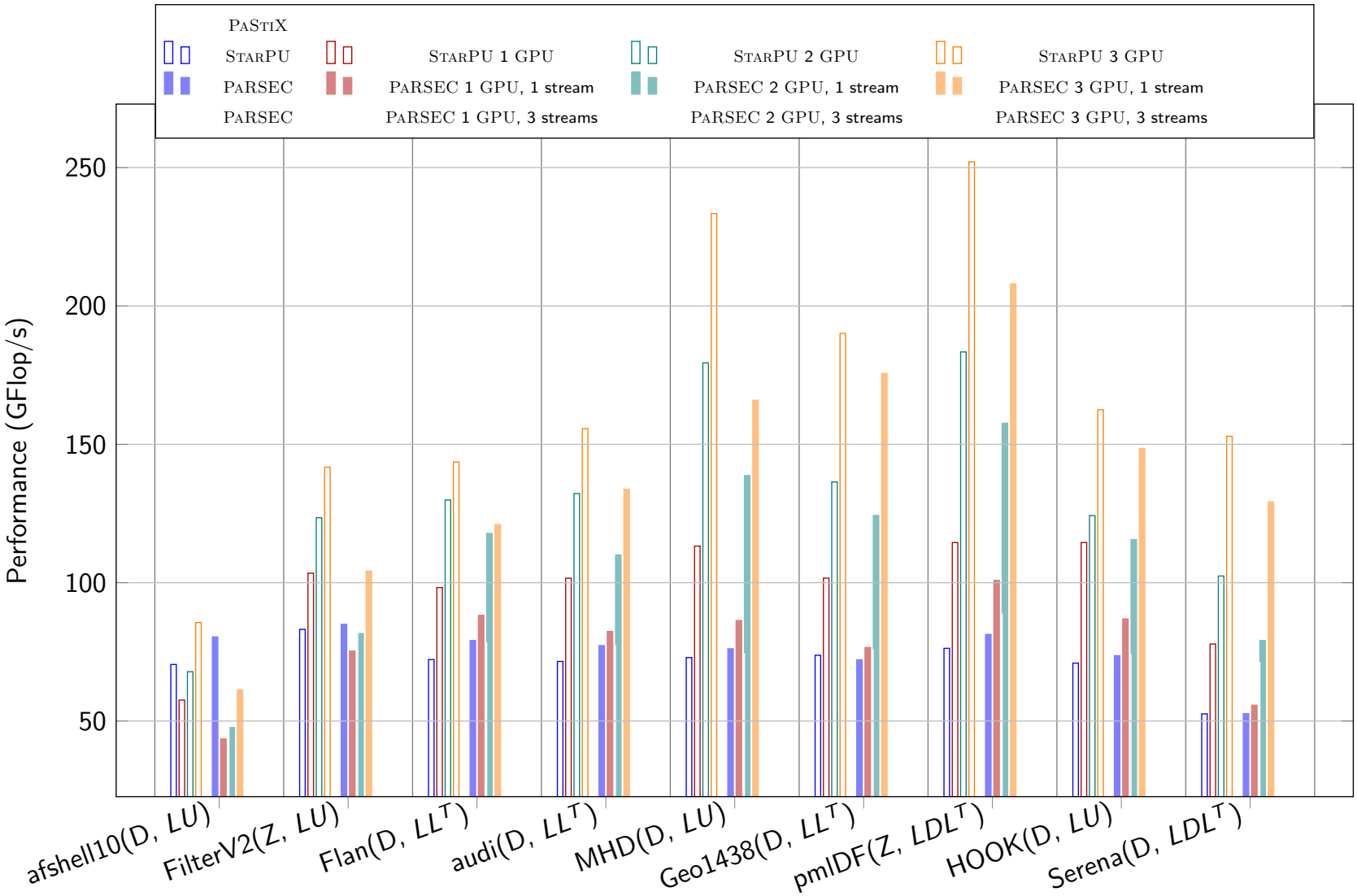
GPU scaling study (1 GPU - 12 CPUs)



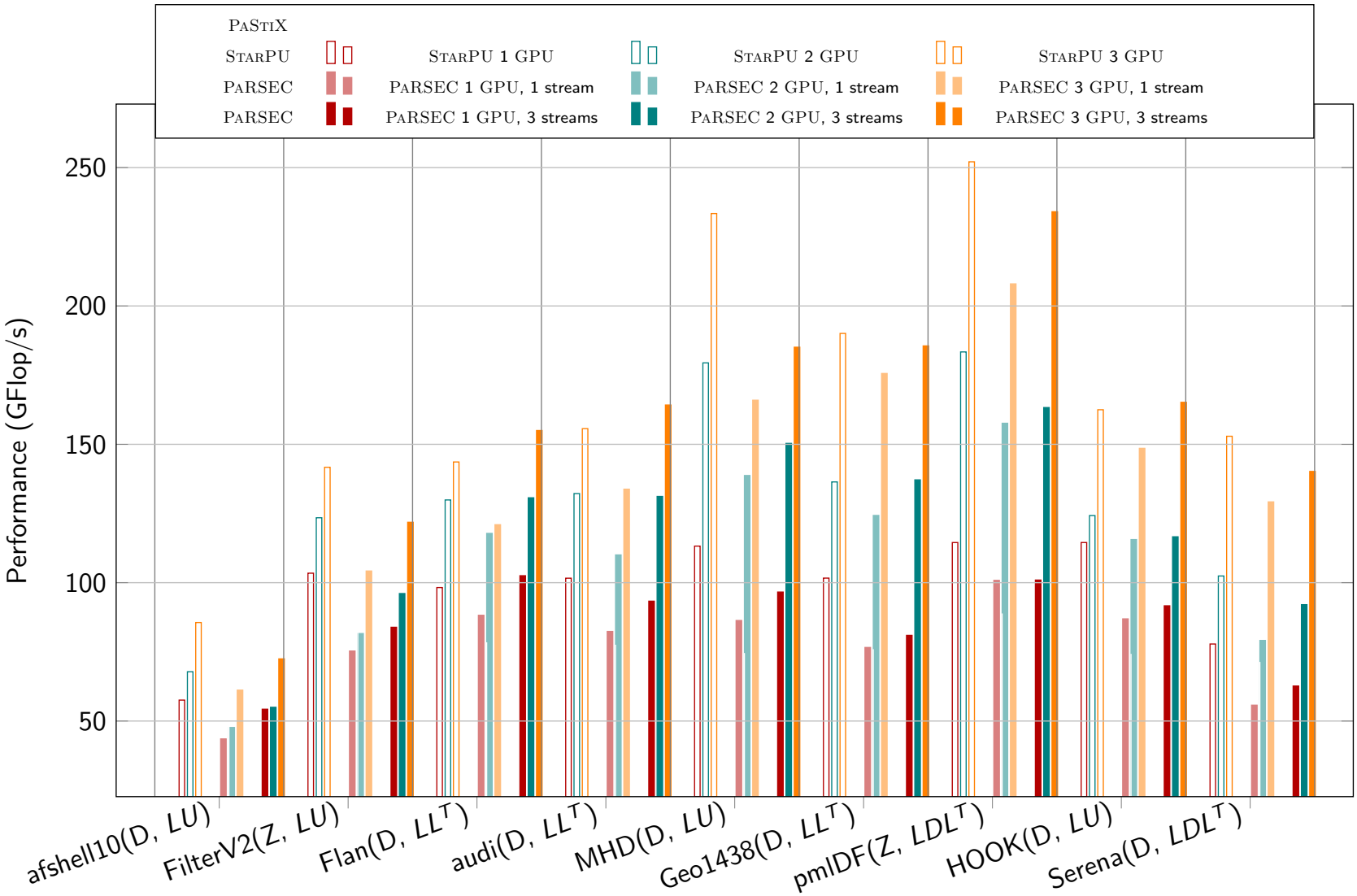
GPU scaling stud (2 GPUs - 12 CPUs)



GPU scaling study (3 GPUs - 12 CPUs)



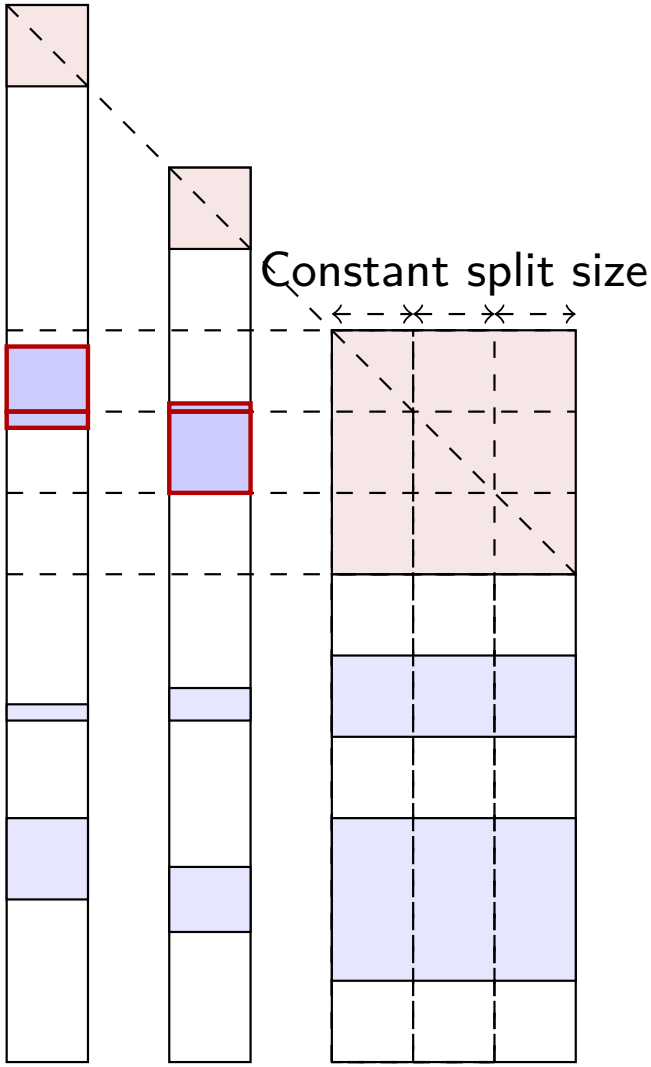
GPU scaling study (Multi-streams - 12 CPUs)



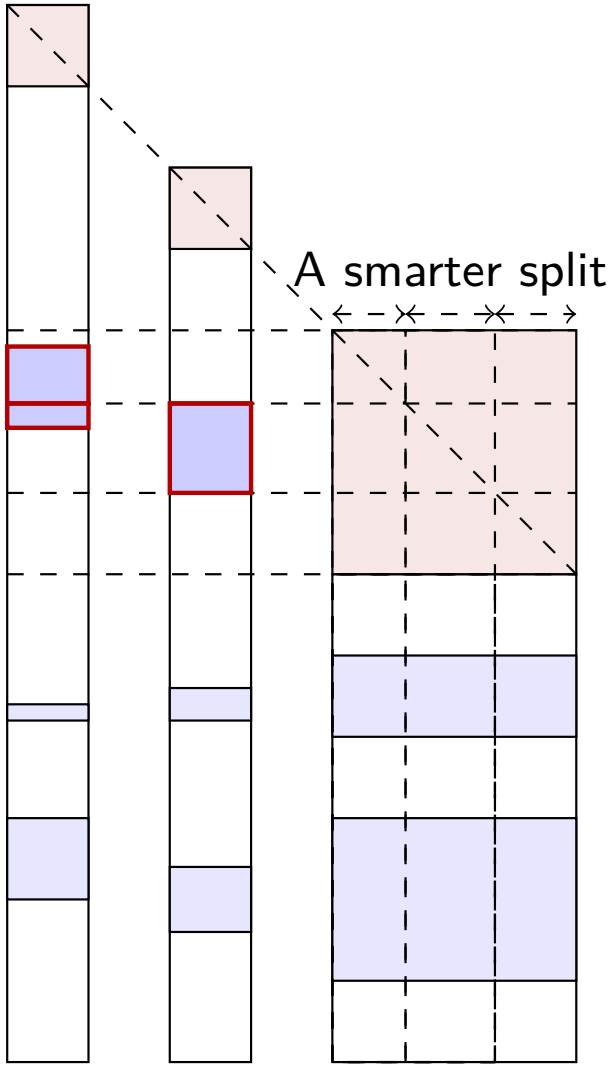
Improvements on granularity

1. Try to improve BLAS efficiency by larger blocking
   - Study impact of SCOTCH minimal subblock parameter (cmin)
2. Create more parallelism while avoiding low flops tasks
   - Improve supernode splitting algorithm

Idea of smarter splitting algorithm



(a) Regular splitting (Actual)



(b) Adapted splitting (New)

Preliminary results

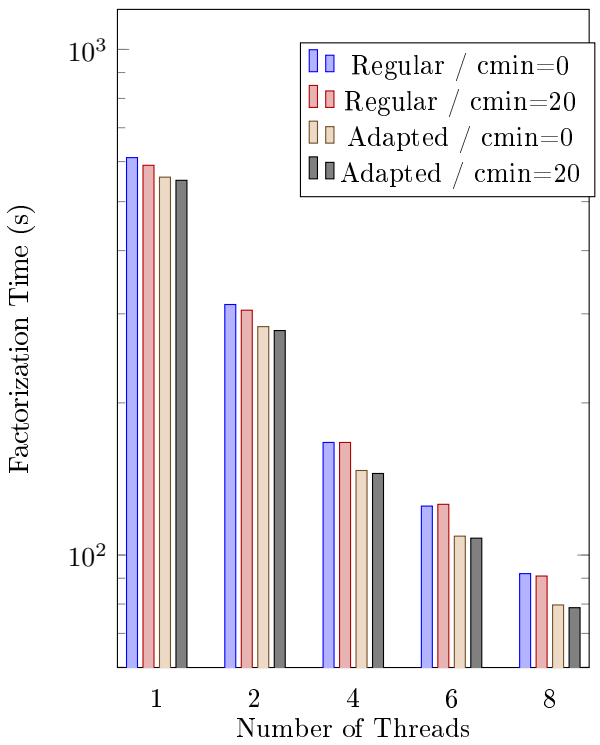


Figure: Audi, LL^T, D

Splitting alg.	Regular		Adapted	
SCOTCH cmin	0	20	0	20
Analyze time	1.95 s	0.35 s	2.56 s	0.42 s
Number of panels	118814	10082	118220	9491
Number of blocks	2283029	338493	2213497	280722
Created by splitting	65147	48284	18072	13081
Avg. panel size	7.94262	93.602	7.98253	99.4305
Avg. block height	10.1546	29.2206	9.08452	24.5355
Memory usage	10.1 Go	10.7 Go	10.5 Go	11.1 Go

Adapted splitting algorithm

- 6-15% improvement on factorization time
- 16-20% augmentation on analyze time

SCOTCH cmin

- 80% faster on analyze time
- No impact on factorization time
- 6% increase on memory consumption

5
Low-rank compression - $\mathcal{H}$-matrix

$\mathcal{H}$-PaStiX ?

Many works on hierarchical matrices

- Eric Darve : Hierarchical matrices classifications (*Building O(N) Linear Solvers Using Nested Dissection*)
- Sherry Li : Multifrontal solver + HSS (*Towards an Optimal-Order Approximate Sparse Factorization Exploiting Data-Sparseness in Separators*)
- David Bindel : CHOLMOD + Low Rank (*An Efficient Solver for Sparse Linear Systems Based on Rank-Structured Cholesky Factorization*)
- Jean-Yves L'Excellent : MUMPS + BLR

FastLA associate team between INRIA/Berkeley/Stanford

Workpackages

1. Check the potential compression ratio on top level blocks
2. Develop a prototype with:
   - low-rank compression on the larger supernodes
   - compression tree built at each update
3. Study coupling between nested dissection and compression tree ordering

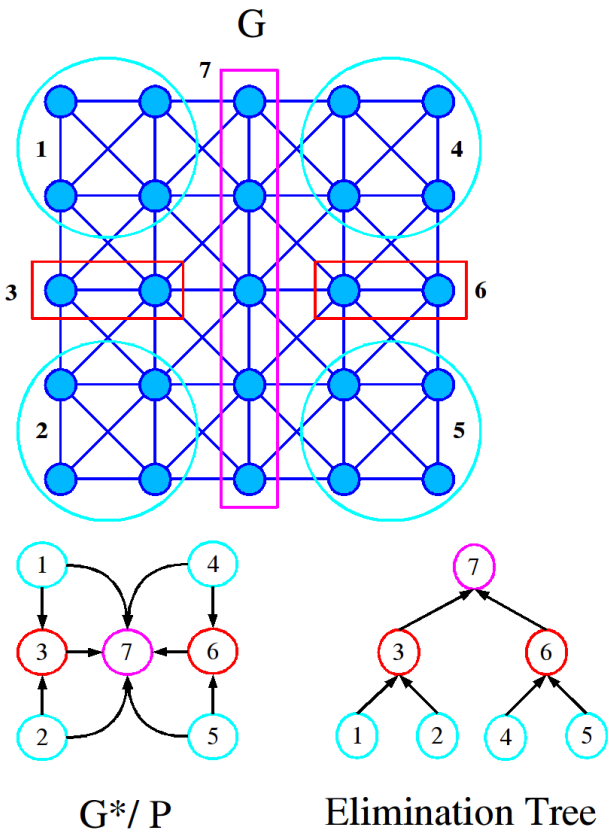
Which algorithm to find low-rank approximation ?

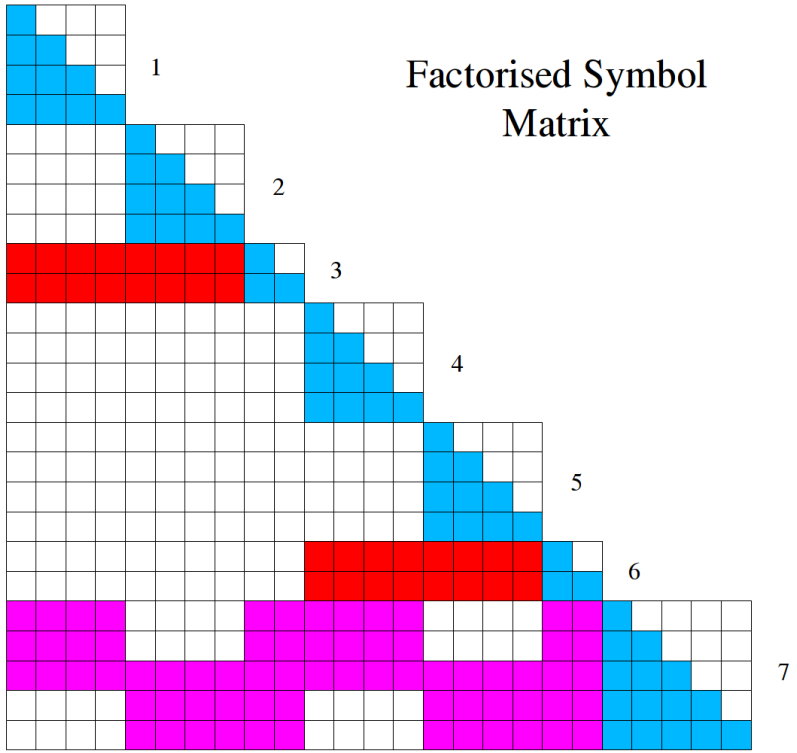
SVD, RR-LU, RR-QR, ACA, CUR, Random ...

Which family of hierarchical matrix ?

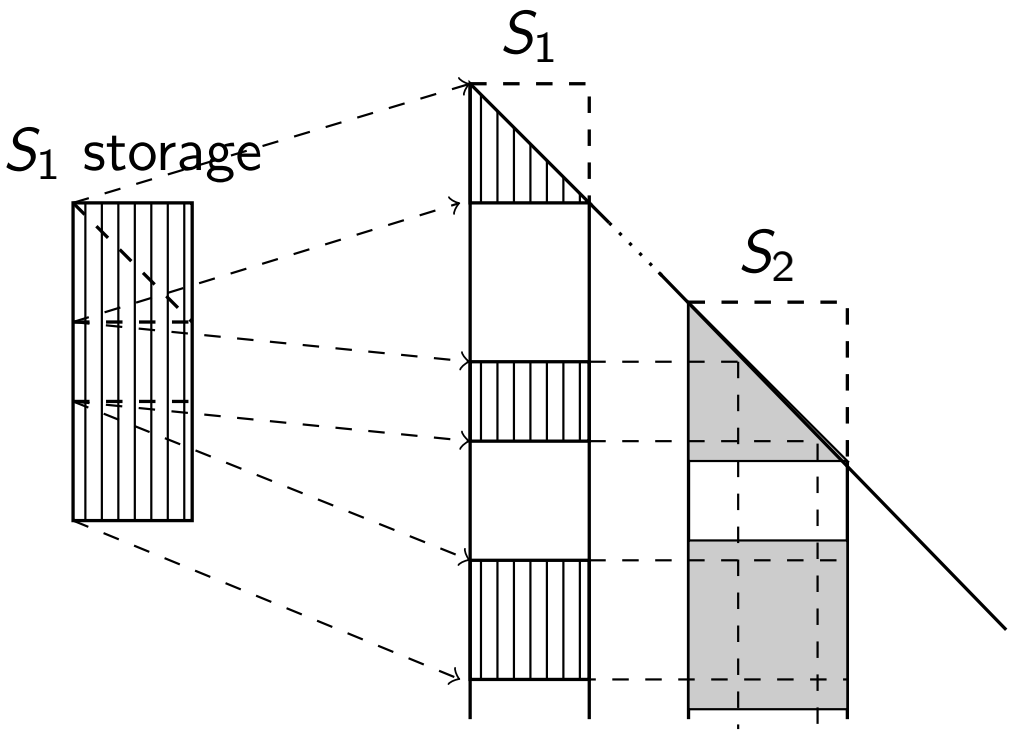
$\mathcal{H}$, $\mathcal{H}^2$, HODLR ...

Target larger supernodes for compression (top tree)

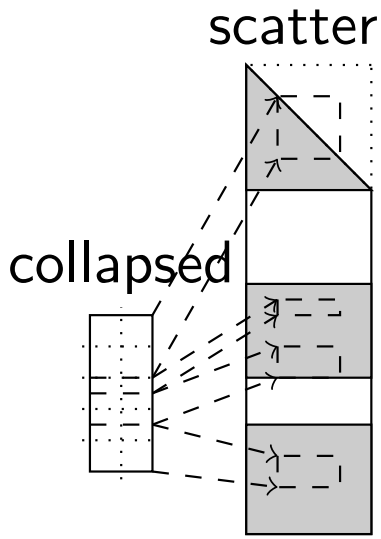




Sparse structure of supernodes

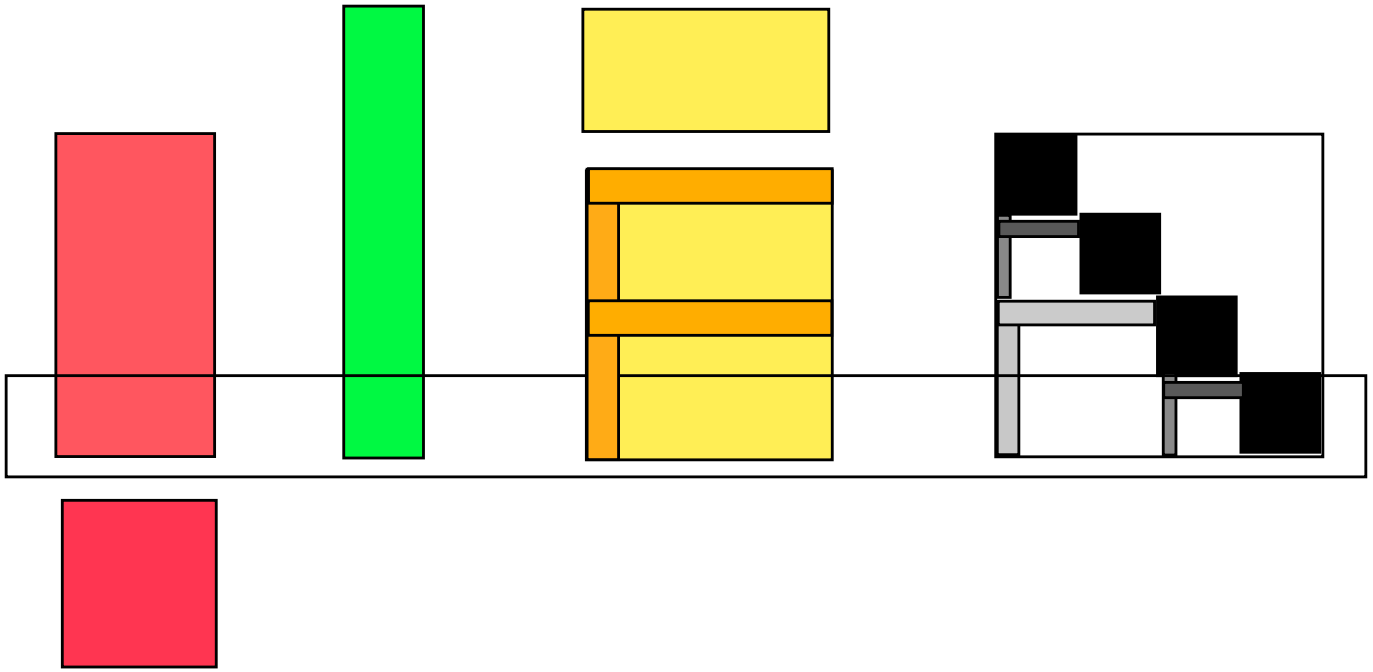


(a) Compressed storage



(b) Scatter collapsed blocks

Forming compressed updates



6 Conclusion

MURGE: a common API to the sparse linear solvers of HiePACS

`http://murge.gforge.inria.fr`

Features

- Through one interface, access to many solver strategies
- Enter a graph/matrix in a centralized or distributed way
- Simple formats : coordinate, CSR or CSC
- Very easy to implement an assembly step

General structure of the code

```
MURGE_Initialize(idnbr, ierror)
MURGE_SetDefaultOptions(id, MURGE_ITERATIVE)  /* Choose general strategy */
MURGE_SetOptionInt(id, MURGE_DOF, 3)  /* Set degrees of freedom */
..
MURGE_Graph_XX(id..) /* Enter the graph : several possibilities */
DO
 MURGE_SetOptionReal(id, MURGE_DROPTOL1, 0.001) /* Threshold for ILUT */
 MURGE_SetOptionReal(id, MURGE_PREC, 1e-7)  /* Precision of solution */
 ...
 /** Enter new coefficient for the matrix **/
  MURGE_AssemblyXX(id..) /* Enter the matrix coefficients */
 DO
   MURGE_SetRHS(id, rhs) /* Set the RHS */
   MURGE_GetSol(id, x)   /* Get the solution */
 END
  MURGE_MatrixReset(id) /* Reset matrix coefficients */
END
MURGE_Clean(id)  /* Clean-up for system "id" */
MURGE_Finalize() /* Clean-up all remaining structure */
```

BACCHUS/HiePACS softwares

Graph/Mesh partitioner and ordering :

http://scotch.gforge.inria.fr

Sparse linear system solvers :

http://pastix.gforge.inria.fr

http://hips.gforge.inria.fr

Thank You

Pierre Ramet

HiePACS

http://www.labri.fr/ ramet