Toward a supernodal sparse direct solver over DAG runtimes

HOSCAR 2013, Bordeaux

X. Lacoste

Xavier LACOSTE
HiePACS team – Inria Bordeaux Sud-Ouest

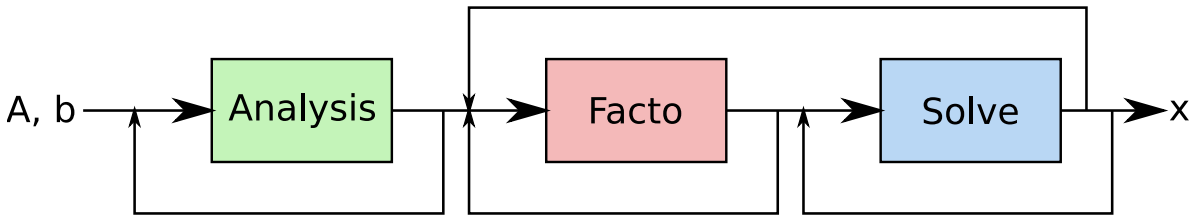
November 27, 2012

Guideline

1
Context and goals

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 Solver Highlights (multicore)

Manumanu (SGI): 20 x 8 Intel Xeon, 2.67GHz, 630 Go RAM

Name	**N**	**NNZ_A**	**Fill ratio**	**OPC**	**Fact**
Audi	9.44×10^5	3.93×10^7	31.28	5.23×10^{12}	float LL^T
10M	1.04×10^7	8.91×10^7	75.66	1.72×10^{14}	complex LDL^T

Audi	8	64	128				160
			128	2x64	4x32	8x16	
Facto (s)	103	21.1	17.8	18.6	13.8	**13.4**	17.2
Mem (Gb)	11.3	12.7	**13.4**	2x7.68	4x4.54	8x2.69	14.5
Solve (s)	1.16	0.31	0.40	0.32	0.21	**0.14**	0.49

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

Direct Solver Highlights (cluster of multicore)

RC3 matrix - complex double precision
N=730700 - NNA=41600758 - Fill-in=50 - 2*6 Westmere
Intel 2.93Ghz - 96Go

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

Goals

- Scalable factorization on emerging architectures (Distributed and manycore (GPU, Xeon Phi...);
- Improve granualarity.

2
Using new emerging architectures

Goals

- New parallel machines with accelerators (GPU and others);
- Achieve scalability on the whole computing units with a sparse direct solver.

Possible solutions

- Multicore: PaStiX already finely tuned with MPI and P-Threads;
- Multiple-GPUs and many-cores, two solutions:
  - Manually handle GPUs $\Rightarrow$ lot of work, heavy maintenance;
  - Use dedicated runtime $\Rightarrow$ May loose the performance obtained on multicore, easy to add new computing devices.

Elected solution, runtime:

- StarPU: RUNTIME – Inria Bordeaux Sud-Ouest;
- PaRSEC: ICL – University of Tennessee, Knoxville.

Panel factorization

- Factorization of the diagonal block (xxTRF);
- TRSM on the extra-diagonal blocks (ie. solves $X \times b_d = b_{i,i>d}$ – where b_d is the diagonal block).

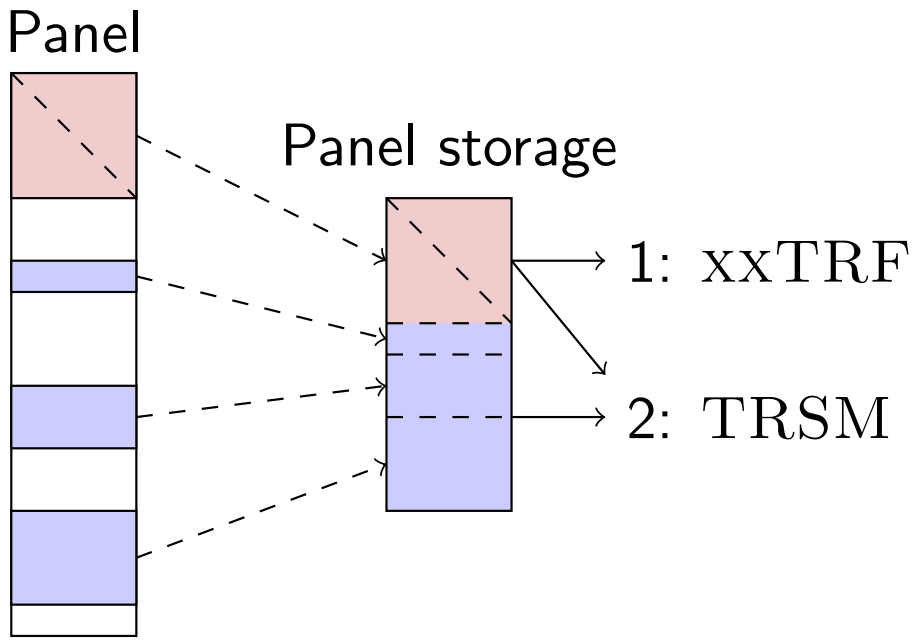


Figure: Panel update

Trailing supernodes update

- One global GEMM in a temporary buffer;
- Scatter addition (many AXPY).

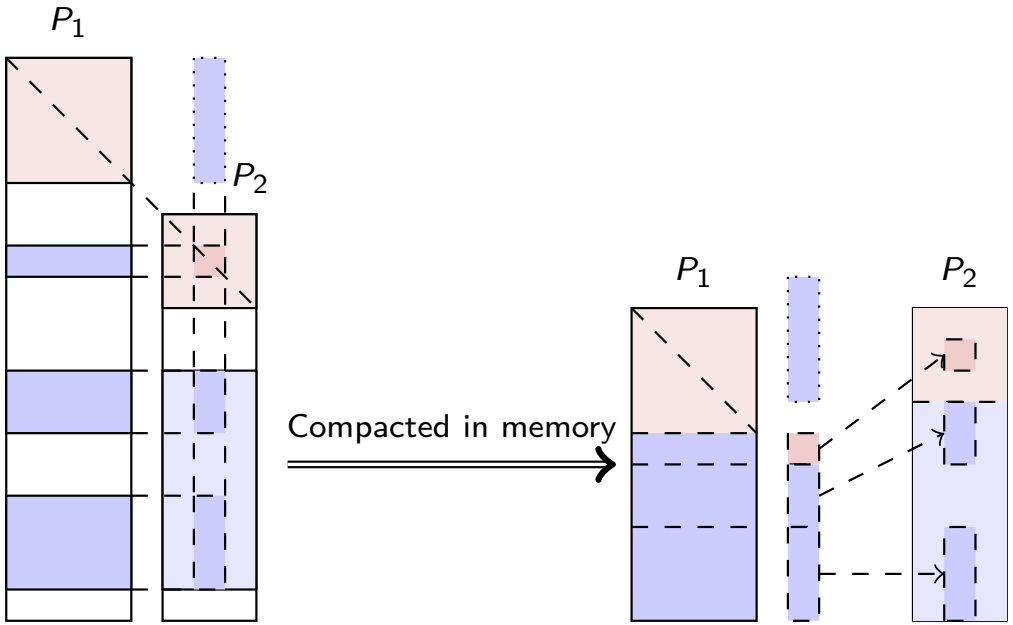


Figure: Panel update

Why a new kernel ?

- A BLAS call $\Rightarrow$ a CUDA startup paid;
- Many AXPY calls $\Rightarrow$ loss of performance.

$\Rightarrow$ need a GPU kernel to compute all the updates from P_1 on P_2 at once.

How ?

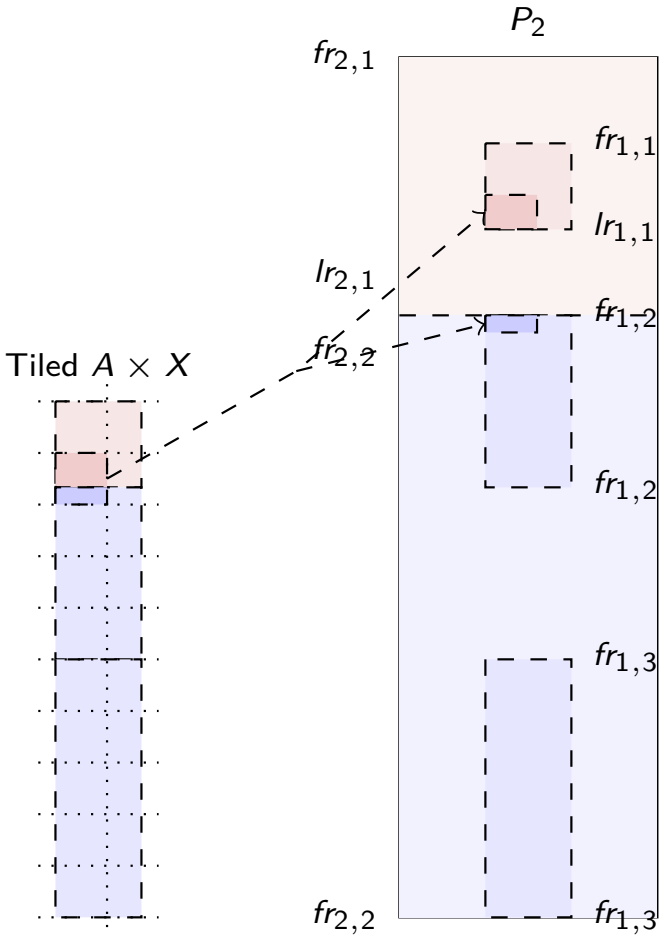
auto-tunning GEMM CUDA kernel

- Auto-tunning done by the framework ASTRA developped by Jakub Kurzak for MAGMA and inspired from ATLAS;
- computes $C \leftarrow \alpha AB + \beta C$, C split into a 2D tiled grid;
- a block of threads computes each tile of the new C;
- each thread computes several entries of the tile in the shared memory and add it from C into the global memory.

Sparse GEMM cuda kernel

- Based on auto-tuning GEMM CUDA kernel;
- Added two arrays giving first and last line of each blocks of P_1 and P_2;
- Computes an offset used when adding to the global memory.

Sparse GEMM on GPU



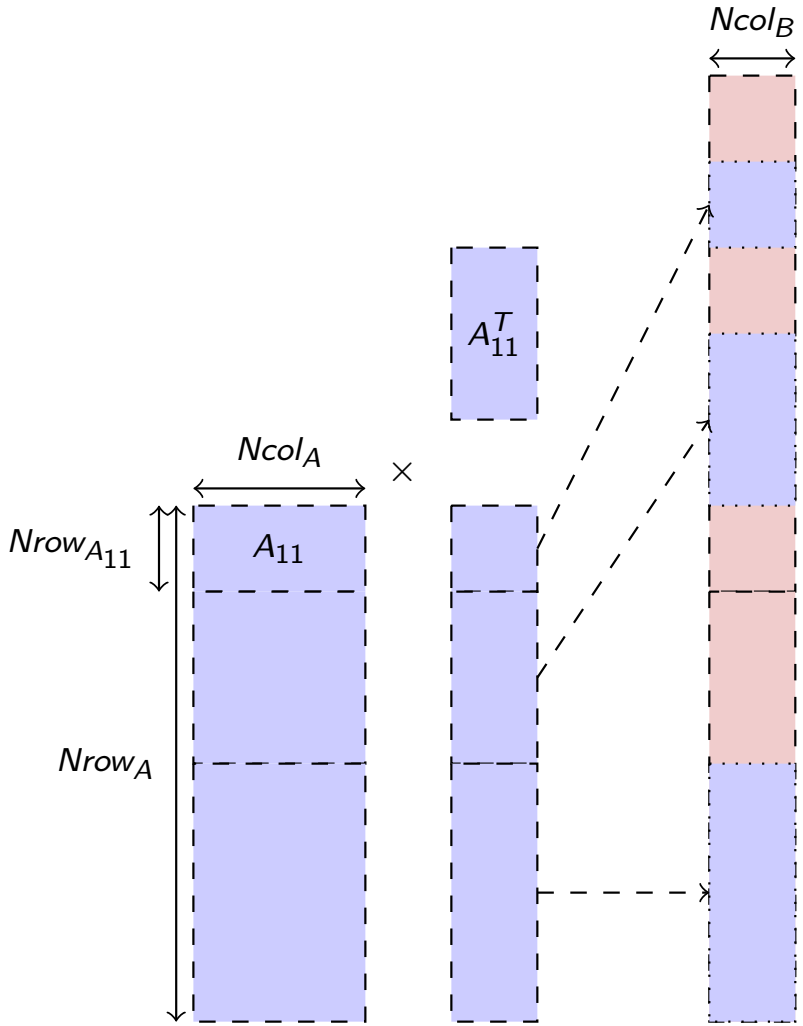
blocknbr = 3;
blocktab = [$fr_{1,1}$, $lr_{1,1}$,
$fr_{1,2}$, $lr_{1,2}$,
$fr_{1,3}$, $lr_{1,3}$];

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

GPU kernel experimentation



Parameters

- $Ncol_A = 100$;
- $Ncol_B = Nrow_{A_{11}} = 100$;
- $Nrow_A$ varies from 100 to 2000;
- Random number and size of blocks in A;
- Random blocks in B matching A;
- Get mean time of 10 runs for a fixed $Nrow_A$ with different blocks distribution.

Figure: GPU kernel experimentation

GPU kernel performance

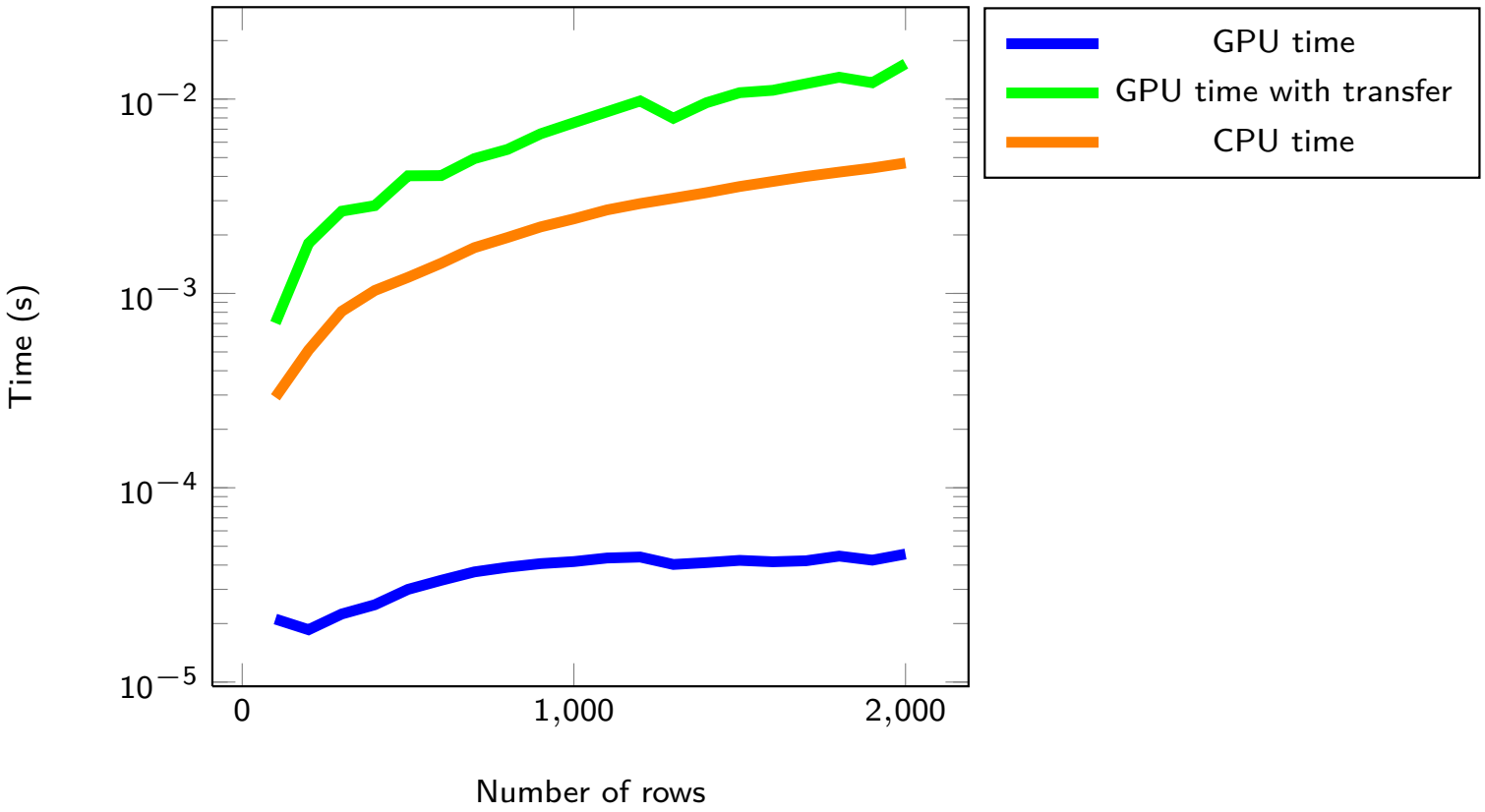


Figure: Sparse kernel timing with 100 columns.

Runtimes

- Task-based programming model;
- Tasks scheduled on computing units (CPUs, GPUs, ...);
- Data transfers management;
- Dynamicaly build models for kernels (StarPU);
- Add new scheduling strategies with plugins;
- Get informations on idle times and load balances.

StarPU Tasks submission

Algorithm 1: StarPU tasks submission

forall the *Supernode* S_1 **do**
  submit_panel (S_1);
  /* update of the panel */
  forall the *extra diagonal block* B_i *of* S_1 **do**
    $S_2 \leftarrow$ supernode_in_front_of (B_i);
    submit_gemm (S_1,S_2);
    /* sparse GEMM $B_{k,k \geq i} \times B_i^T$ substracted from S_2 */
  end
end

PaRSEC's parametrized taskgraph

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

Figure: Panel factorization description in PaRSEC

Giving more information to the runtime

Definition of a new work stealing scheduler

- Use PaStiX static tasks placement;
- steal tasks from other contexts when no more tasks are ready (based on StarPU work stealing policy).

Choose which GEMM will run on GPUs

- static distribution of a portion of the panels onto GPUs following a given criterium:
  - panel size;
  - number of update on the panel;
  - number of flops for the panel update.

Matrices and Machines

Matrices

Name	N	NNZ_A	Fill ratio	OPC	Fact
MHD	4.86×10^5	1.24×10^7	61.20	9.84×10^{12}	Float LU
Audi	9.44×10^5	3.93×10^7	31.28	5.23×10^{12}	Float LL^T
10M	1.04×10^7	8.91×10^7	75.66	1.72×10^{14}	Complex LDL^T

Machines

Machine	Processors	Frequency	GPUs	RAM
Romulus	AMD Opteron 6180 SE (4×12)	2.50 GHz	Tesla T20 ($\times 2$)	256 GiB
Mirage	Westmere Intel Xeon X5650 (2×6)	2.67 GHz	Tesla M2070 ($\times 3$)	36 GiB

CPU only results with Audi on Romulus

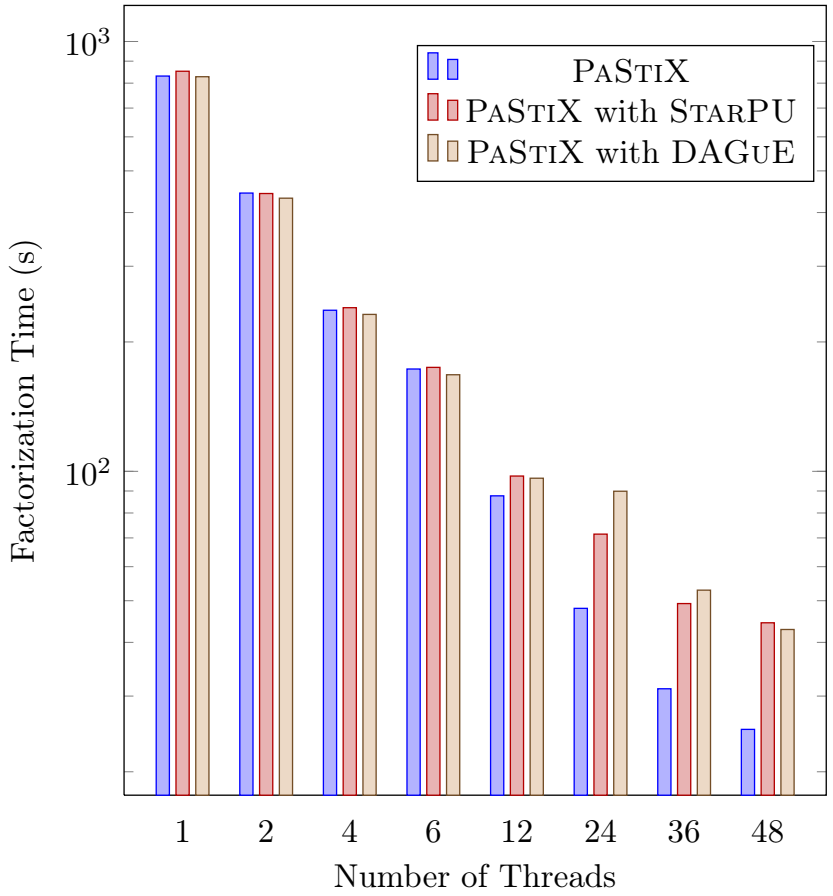


Figure: LL^T decomposition on Audi (double precision)

CPU only results with MHD on Romulus

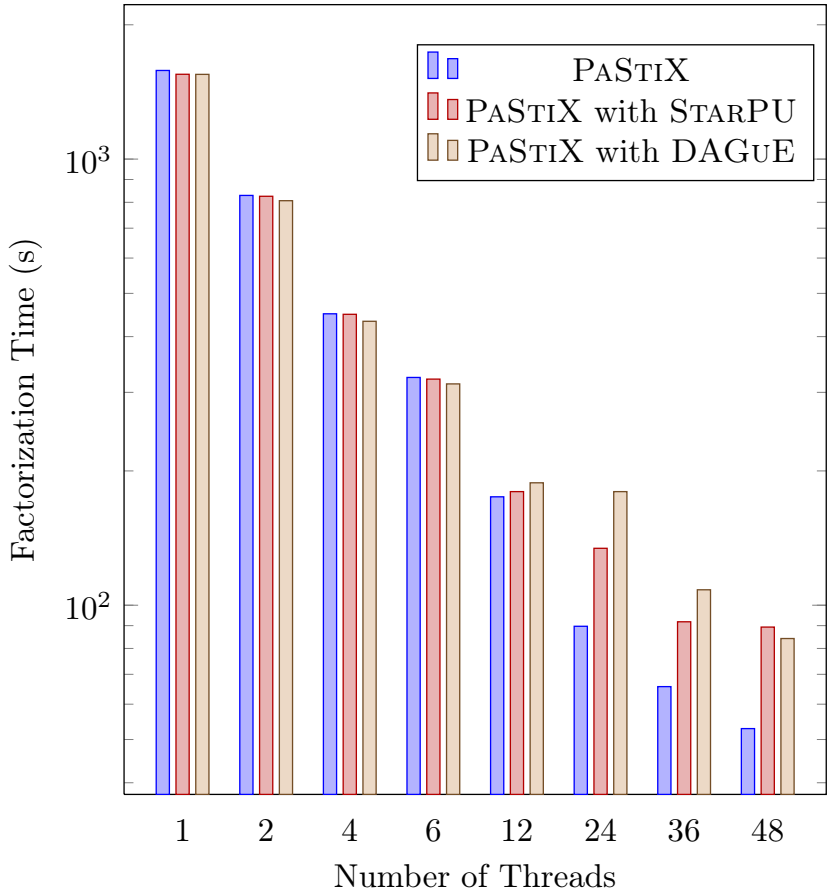


Figure: *LU* decomposition on MHD (double precision)

CPU only results with 10 Millions case on Romulus

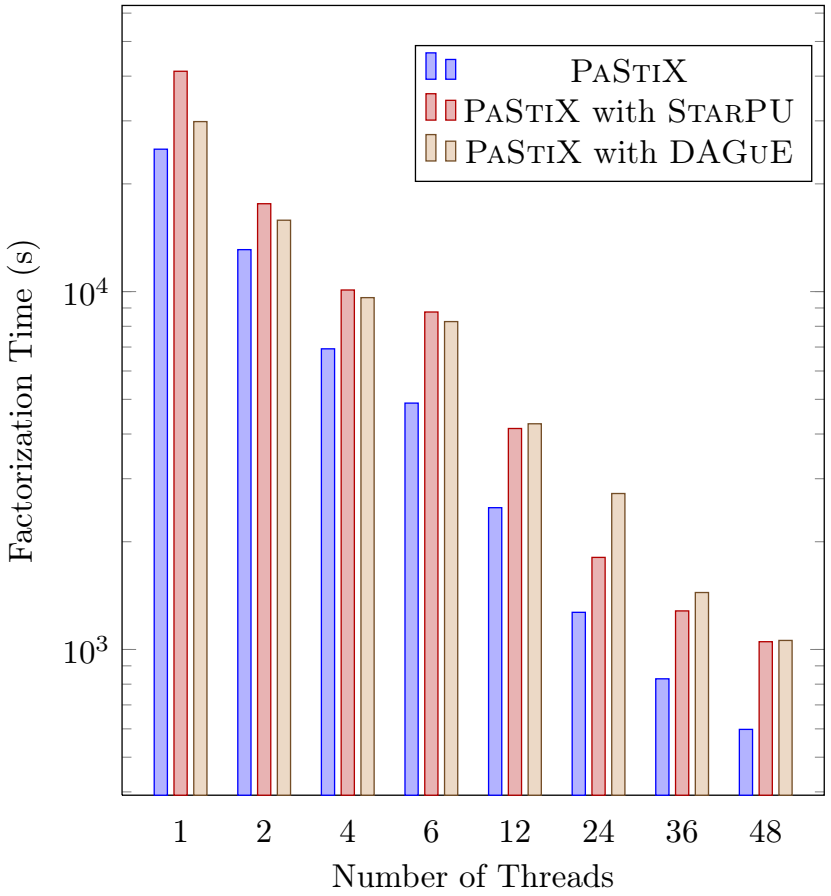


Figure: LDL^T decomposition on 10M (double complex)

GPU study on mirage

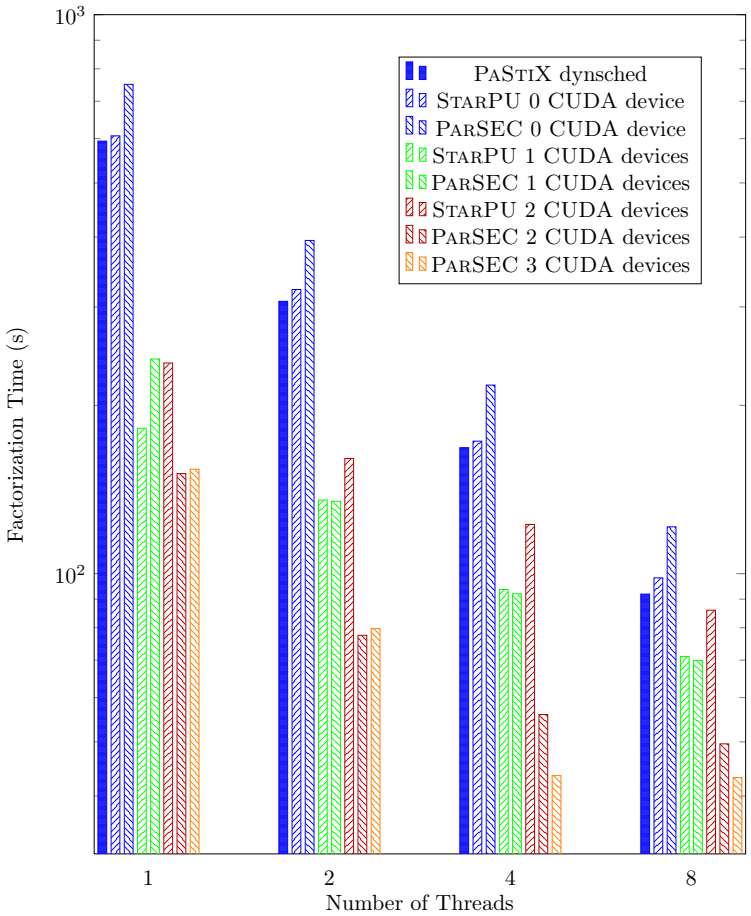


Figure: LL^T decomposition on Audi (double precision)

3
Improvement on granularity

Improvements on granularity

- Graph preprocessing minimal blocking $\Rightarrow$ reduce number of tasks;
- Smarter panel splitting to suppress low flop tasks.

Study on SCOTCH minimal subblock parameter (cmin), on Riri

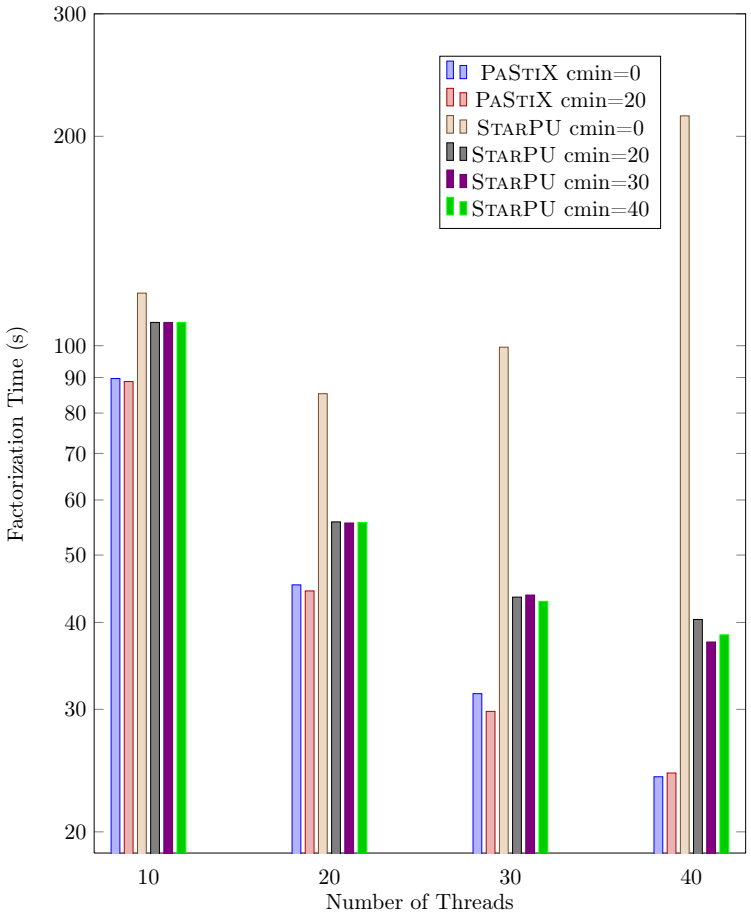


Figure: LL^T decomposition on Audi (double precision)

Panel splitting

Why splitting panels ?

- create more parallelism.

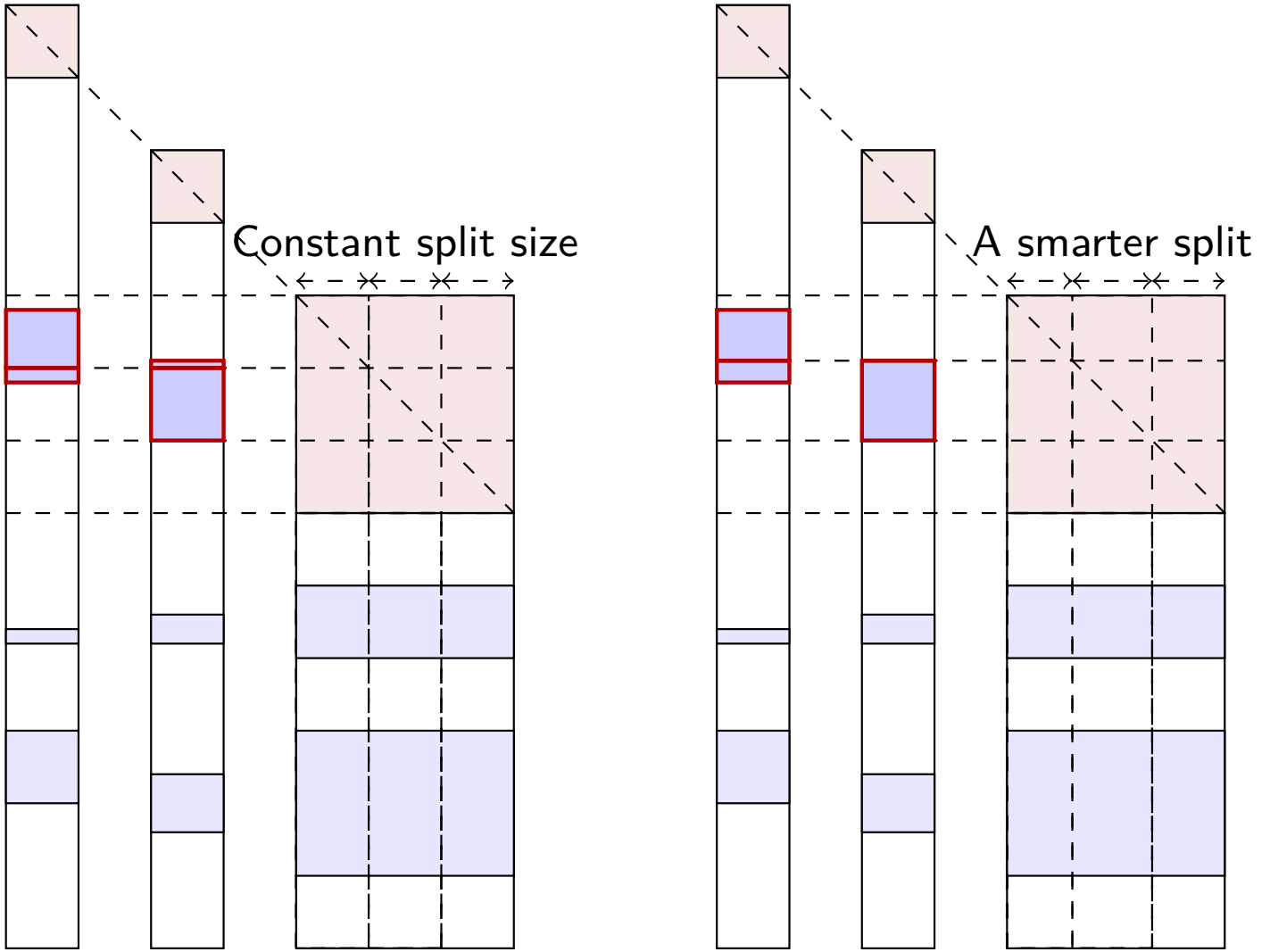
Drawback

- induce facing block splitting that might create many tiny blocks.

Solution

- smarter panel splitting;
- avoid tiny blocks creation which leads to inneficient BLAS.

A smarter split



(a) Classical equal splitting (b) Smarter adapted splitting

A smarter split

- For each panel:
  - Construct a partition of the panel height with the number of facing blocks;
  - Decide to split where the number of splitted blocks is minimal.

Preprocessing option comparaison on Audi, on Mirage

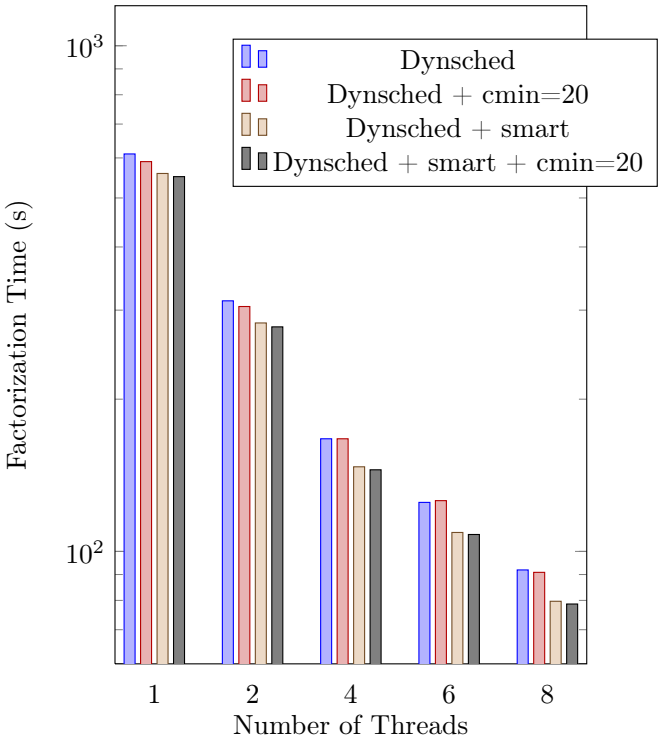


Figure: LL^T decomposition on Audi (double precision)

method	Dynsched		Dynsched + smart	
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

Smart panel splitting

- Factorization time reduction: 6-15%;
- Analyze time augmentation: 16-20%.

cmin 20

- Analyze time reduction: 80%;
- Less tasks may reduce runtime overhead, no effect on PaStiX fatorization time.

4
Conclusion and future works

Conclusion

- Runtimes:
  - Timing and scaling close to original PaStiX;
  - Speedup obtained with one (StarPU) or two (PaRSEC) GPUs;
- Granularity:
  - Using bigger minimal block size reduce number of tasks and improve results with runtimes;
  - Smart splitting improve results in all cases;

Future works

- More locality:
  - StarPU: use contexts to attach tasks to a pool of processing units
  - PaRSEC: virtual processors to organize scheduling by socket;
- Streams: need streams to perform multiple kernel execution on a GPU at a time
- Group tasks to reduce the runtime overhead: gather small tasks in PaStiX or let the runtime decide what is a small task
- Distributed implementation (MPI): mixed Fan-Out (Runtimes de-facto), Fan-In (PaStiX de-facto) implementation of the communications

Around direct solvers in HiePACS

- Two hybrid direct/iterative domain decomposition methods:
  - MaPHyS (Massively Parallel Hybrid Solver)
  - HIPS (Hierarchical Iterative Parallel Solver)
- Interfaces:
  - MURGE: common interface for finite element (PaStiX, HIPS... on going MaPHyS)
  - PETSc interface to PaStiX (new update coming with next PaStiX release)
  - Trilinos interface to PaStiX on the roadmap
  - Python interface via SWIG, will be updated using Cython
- Next generation architectures: Xeon Phi, Kalray, ARM...
- Redesign PaStiX to handle H-matrix approximation (Stanford/Berkeley collaboration)

Thanks !

Xavier LACOSTE

INRIA HiePACS team

HOSCAR - September 03, 2013