Toward a supernodal sparse direct solver over DAG runtimes

HOSCAR 2013, Bordeaux

X. Lacoste
Guideline

Context and goals
   About PaStiX

Using new emerging architectures
   Kernels
      Panel factorization
      Trailing supernodes update (CPU version)
      Sparse GEMM on GPU
   Runtime
   Results

Improvement on granularity
   Smarter panel splitting
   Results

Conclusion and future works
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 $A' = P_nPD_rAD_cQP^T$)

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

<table>
<thead>
<tr>
<th>Name</th>
<th>N</th>
<th>NNZ&lt;sub&gt;A&lt;/sub&gt;</th>
<th>Fill ratio</th>
<th>OPC</th>
<th>Fact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Audi</td>
<td>9.44×10&lt;sup&gt;5&lt;/sup&gt;</td>
<td>3.93×10&lt;sup&gt;7&lt;/sup&gt;</td>
<td>31.28</td>
<td>5.23×10&lt;sup&gt;12&lt;/sup&gt;</td>
<td>float $LL^T$</td>
</tr>
<tr>
<td>10M</td>
<td>1.04×10&lt;sup&gt;7&lt;/sup&gt;</td>
<td>8.91×10&lt;sup&gt;7&lt;/sup&gt;</td>
<td>75.66</td>
<td>1.72×10&lt;sup&gt;14&lt;/sup&gt;</td>
<td>complex $LDL^T$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Audi</th>
<th>8</th>
<th>64</th>
<th>2x64</th>
<th>4x32</th>
<th>8x16</th>
<th>160</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facto (s)</td>
<td>103</td>
<td>21.1</td>
<td>17.8</td>
<td>18.6</td>
<td>13.8</td>
<td><strong>13.4</strong></td>
</tr>
<tr>
<td>Mem (Gb)</td>
<td>11.3</td>
<td>12.7</td>
<td><strong>13.4</strong></td>
<td>2x7.68</td>
<td>4x4.54</td>
<td>8x2.69</td>
</tr>
<tr>
<td>Solve (s)</td>
<td>1.16</td>
<td>0.31</td>
<td>0.40</td>
<td>0.32</td>
<td>0.21</td>
<td><strong>0.14</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10M</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
<th>160</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facto (s)</td>
<td>3020</td>
<td>1750</td>
<td>654</td>
<td>356</td>
<td>260</td>
</tr>
<tr>
<td>Mem (Gb)</td>
<td>122</td>
<td>124</td>
<td>127</td>
<td>133</td>
<td>146</td>
</tr>
<tr>
<td>Solve (s)</td>
<td>24.6</td>
<td>13.5</td>
<td>3.87</td>
<td>2.90</td>
<td>2.89</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Facto</th>
<th>1 MPI</th>
<th>2 MPI</th>
<th>4 MPI</th>
<th>8 MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 thread</td>
<td>6820</td>
<td>3520</td>
<td>1900</td>
<td>1890</td>
</tr>
<tr>
<td>6 threads</td>
<td>1020</td>
<td>639</td>
<td>337</td>
<td>287</td>
</tr>
<tr>
<td>12 threads</td>
<td><strong>525</strong></td>
<td>360</td>
<td><strong>155</strong></td>
<td>121</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mem Gb</th>
<th>1 MPI</th>
<th>2 MPI</th>
<th>4 MPI</th>
<th>8 MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 thread</td>
<td>34</td>
<td>19,2</td>
<td>12,5</td>
<td>9,22</td>
</tr>
<tr>
<td>6 threads</td>
<td>34,3</td>
<td>19,5</td>
<td>12,8</td>
<td>9,66</td>
</tr>
<tr>
<td>12 threads</td>
<td>34,6</td>
<td>19,7</td>
<td>13</td>
<td>9,14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solve</th>
<th>1 MPI</th>
<th>2 MPI</th>
<th>4 MPI</th>
<th>8 MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 thread</td>
<td>6,97</td>
<td>3,75</td>
<td>1,93</td>
<td>1,03</td>
</tr>
<tr>
<td>6 threads</td>
<td>2,5</td>
<td>1,43</td>
<td><strong>0,78</strong></td>
<td><strong>0,54</strong></td>
</tr>
<tr>
<td>12 threads</td>
<td><strong>1,33</strong></td>
<td>0,93</td>
<td>0,66</td>
<td>0,59</td>
</tr>
</tbody>
</table>
Goals

- Scalable factorization on emerging architectures (Distributed and manycore (GPU, Xeon Phi...));
- Improve granulararity.
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 ⇒ lot of work, heavy maintenance;
  ▶ Use dedicated runtime ⇒ 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 ⇒ a CUDA startup paid;
- Many AXPY calls ⇒ loss of performance.

⇒ need a GPU kernel to compute all the updates from $P_1$ on $P_2$ at once.
How?

auto-tunning GEMM CUDA kernel

- Auto-tuning done by the framework ASTRA developed 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

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
Using new emerging architectures Kernels

**GPU kernel performance**

![Graph showing GPU kernel performance](image)

**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**

```plaintext
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 $B_{k,k \geq i} \times B_i^T$ subtracted from $S_2$ */
    end
end
```
**PARSEC’s parametrized taskgraph**

```plaintext
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

<table>
<thead>
<tr>
<th>Name</th>
<th>N</th>
<th>NNZ&lt;sub&gt;A&lt;/sub&gt;</th>
<th>Fill ratio</th>
<th>OPC</th>
<th>Fact</th>
</tr>
</thead>
<tbody>
<tr>
<td>MHD</td>
<td>$4.86 \times 10^5$</td>
<td>$1.24 \times 10^7$</td>
<td>61.20</td>
<td>$9.84 \times 10^{12}$</td>
<td>Float LU</td>
</tr>
<tr>
<td>Audi</td>
<td>$9.44 \times 10^5$</td>
<td>$3.93 \times 10^7$</td>
<td>31.28</td>
<td>$5.23 \times 10^{12}$</td>
<td>Float $LL^T$</td>
</tr>
<tr>
<td>10M</td>
<td>$1.04 \times 10^7$</td>
<td>$8.91 \times 10^7$</td>
<td>75.66</td>
<td>$1.72 \times 10^{14}$</td>
<td>Complex $LDL^T$</td>
</tr>
</tbody>
</table>

Machines

<table>
<thead>
<tr>
<th>Machine</th>
<th>Processors</th>
<th>Frequency</th>
<th>GPUs</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Romulus</td>
<td>AMD Opteron 6180 SE (4 × 12)</td>
<td>2.50 GHz</td>
<td>Tesla T20 (×2)</td>
<td>256 GiB</td>
</tr>
<tr>
<td>Mirage</td>
<td>Westmere Intel Xeon X5650 (2 × 6)</td>
<td>2.67 GHz</td>
<td>Tesla M2070 (×3)</td>
<td>36 GiB</td>
</tr>
</tbody>
</table>
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 inefficient 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 comparison on Audi, on Mirage

![Graph showing factorization times for different methods and number of threads.]

<table>
<thead>
<tr>
<th>method</th>
<th>Dynsched</th>
<th>Dynsched + cmin=20</th>
<th>Dynsched + smart</th>
<th>Dynsched + smart + cmin=20</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmin</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>analyze time</td>
<td>1.95 s</td>
<td>0.35 s</td>
<td>2.56 s</td>
<td>0.42 s</td>
</tr>
<tr>
<td>number of panels</td>
<td>118814</td>
<td>10082</td>
<td>118220</td>
<td>9491</td>
</tr>
<tr>
<td>number of blocks</td>
<td>2283029</td>
<td>338493</td>
<td>2213497</td>
<td>280722</td>
</tr>
<tr>
<td>created by splitting</td>
<td>65147</td>
<td>48284</td>
<td>18072</td>
<td>13081</td>
</tr>
<tr>
<td>Avg. panel size</td>
<td>7.94262</td>
<td>93.602</td>
<td>7.98253</td>
<td>99.4305</td>
</tr>
<tr>
<td>Avg. block height</td>
<td>10.1546</td>
<td>29.2206</td>
<td>9.08452</td>
<td>24.5355</td>
</tr>
<tr>
<td>Memory usage</td>
<td>10.1 Go</td>
<td>10.7 Go</td>
<td>10.5 Go</td>
<td>11.1 Go</td>
</tr>
</tbody>
</table>

**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 factorization time.

**Figure**: $LL^T$ decomposition on Audi (double precision)
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;
Conclusion and future works

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