SCALABILITY OF PARALLEL SPARSE DIRECT SOLVERS: METHODS, MEMORY AND PERFORMANCE

Soutenance d’Habilitation à Diriger des Recherches

Alfredo Buttari
26 September 2018, Toulouse

CNRS-IRIT, Université de Toulouse
BACKGROUND
This work deals with the solution of systems of linear equations:

\[ Ax = b, \quad \min_x \|Ax - b\|_2, \quad \min \|x\|_2, \quad Ax = b. \]

where \( A \) is potentially large and sparse, i.e., “it has enough zeros that it pays to take advantage of them” (cit. Wilkinson):

- Reduce memory storage
- Reduce computational costs
- Improve parallelism

This operation is often a cornerstone of numerical applications from a wide range of fields such as mechanics, fluid-dynamics, geophysics, circuit simulation, economy.
Sparse linear solvers

The best known classes of solvers for sparse linear systems are

- **Iterative**: start from an initial guess solution, $x_0$ say, and iteratively improve it until the desired solution accuracy is achieved.

- **Direct**: factorize $A$ into a product of matrices that are easy to invert through Gaussian Elimination ($LU$, $LDL^T$ and Cholesky) or through orthogonal transformations ($QR$).
The best known classes of solvers for sparse linear systems are

- **Iterative**: start from an initial guess solution, $x_0$ say, and iteratively improve it until the desired solution accuracy is achieved.

- **Direct**: factorize $A$ into a product of matrices that are easy to invert through Gaussian Elimination ($LU$, $LDL^T$ and Cholesky) or through orthogonal transformations ($QR$).

Direct methods are generally appreciated because they are

- Robust, general purpose and easy to use
- Performance efficient
The best known classes of solvers for sparse linear systems are:

- **Iterative**: start from an initial guess solution, \( x_0 \) say, and iteratively improve it until the desired solution accuracy is achieved.

- **Direct**: factorize \( A \) into a product of matrices that are easy to invert through Gaussian Elimination (\( LU, LDL^T \) and Cholesky) or through orthogonal transformations (\( QR \)).

Direct methods are generally appreciated because they are:

- **Robust, general purpose and easy to use**
- **Performance efficient**

but these advantages come at the price of:

- **heavy memory and CPU consumption (because of fill-in)**
- **limited scalability**
The multifrontal method* relies on the concept of elimination tree which represents the dependencies between unknowns.

- Nodes are associated with dense matrices called frontal matrices or fronts.

*I. S. Duff et al., 1983.*
The multifrontal method* relies on the concept of elimination tree which represents the dependencies between unknowns.

- Nodes are associated with dense matrices called frontal matrices or fronts.
- In the actual factorization the tree is traversed bottom-up and when a node is visited.

---

*I. S. Duff et al., 1983.*
The multifrontal method* relies on the concept of elimination tree which represents the dependencies between unknowns.

- Nodes are associated with dense matrices called frontal matrices or fronts.
- In the actual factorization, the tree is traversed bottom-up and when a node is visited:
  - The frontal matrix is allocated.

---

*I. S. Duff et al., 1983.*
The multifrontal method* relies on the concept of elimination tree which represents the dependencies between unknowns.

- Nodes are associated with dense matrices called frontal matrices or fronts.
- In the actual factorization, the tree is traversed bottom-up and when a node is visited:
  - The frontal matrix is allocated.
  - It is assembled.

---

*I. S. Duff et al., 1983.
The multifrontal method* relies on the concept of elimination tree which represents the dependencies between unknowns.

- Nodes are associated with dense matrices called frontal matrices or fronts.
- In the actual factorization, the tree is traversed bottom-up and when a node is visited:
  - the frontal matrix is allocated
  - it is assembled
  - child contributions are freed

*I. S. Duff et al., 1983.
The multifrontal method* relies on the concept of elimination tree which represents the dependencies between unknowns.

- Nodes are associated with dense matrices called frontal matrices or fronts.
- In the actual factorization, the tree is traversed bottom-up and when a node is visited:
  - The frontal matrix is allocated.
  - It is assembled.
  - Child contributions are freed.
  - The frontal matrix is factorized.

There are subtle differences depending on the type of factorization.

*I. S. Duff et al., 1983.
The Multifrontal method: complexity

Cubic domain of dimension $N$, with Nested Dissection* ordering

- topmost front is of size $O(N^2)$
- front size is divided by 4 at each level
- each node has 8 children

Flops = \[ \sum_{\ell=0}^{\log(N)} 8^\ell \left( \left( \frac{N}{2^\ell} \right)^2 \right)^3 \]

\[ \begin{array}{c}
N^2 \\
(\frac{N}{2})^2 \\
(\frac{N}{2^2})^2 \\
(8) \\
(8) \\
(\frac{N}{2^l})^2 \\
(\frac{N}{2})^2 \\
\end{array} \]

\[ \text{Flops} = O(N^6) \]

---

Cubic domain of dimension $N$, with Nested Dissection* ordering

- topmost front is of size $O(N^2)$
- front size is divided by 4 at each level
- each node has 8 children

$$\text{Flops} = \sum_{\ell=0}^{\log(N)} 8^\ell \left( \left( \frac{N}{2^\ell} \right)^2 \right)^3$$

<table>
<thead>
<tr>
<th></th>
<th>3D</th>
<th>2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flops</td>
<td>$O(N^6)$</td>
<td>$O(N^3)$</td>
</tr>
<tr>
<td>NNZ Factors</td>
<td>$O(N^4)$</td>
<td>$O(N^2 \log(N))$</td>
</tr>
</tbody>
</table>
This work focuses on the scalability of sparse, direct solvers intended as the ability to solve larger and larger problems. This is addressed along three axes:

- **Scalable Performance**: reduce the execution time as the available computing resources increase by means of parallelism.
- **Scalable Memory**: take advantage of parallelism without increasing the memory footprint so that, if more memory is available, bigger problems can be solved.
- **Scalable Complexity**: when possible, rely on mathematical properties of $A$ to achieve asymptotic reductions in the number of operations and memory.
This work focuses on the **scalability** of sparse, direct solvers intended as the ability to **solve larger and larger problems**. This is addressed along three axes

- **Scalable Performance**: reduce the execution time as the available computing resources increase by means of parallelism
This work focuses on the **scalability** of sparse, direct solvers intended as the ability to **solve larger and larger problems**. This is addressed along three axes

- **Scalable Performance**: reduce the execution time as the available computing resources increase by means of parallelism

- **Scalable Memory**: take advantage of parallelism without increasing the memory footprint so that, if more memory is available, bigger problems can be solved
This work focuses on the **scalability** of sparse, direct solvers intended as the ability to **solve larger and larger problems**. This is addressed along three axes:

- **Scalable Performance**: reduce the execution time as the available computing resources increase by means of parallelism.

- **Scalable Memory**: take advantage of parallelism without increasing the memory footprint so that, if more memory is available, bigger problems can be solved.

- **Scalable Complexity**: when possible, rely on mathematical properties of $A$ to achieve asymptotic reductions in the number of operations and memory.
SCALABLE PERFORMANCE

Cubic domain of dimension $N$, with Nested Dissection ordering

$$T = \sum_{\ell=0}^{\log(N)} 8^\ell \left( \left( \frac{N}{2^\ell} \right)^2 \right)^3$$

<table>
<thead>
<tr>
<th></th>
<th>Params</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{seq}$</td>
<td></td>
<td>$N^6$</td>
</tr>
</tbody>
</table>
Parallelism in the Multifrontal method

Cubic domain of dimension $N$, with Nested Dissection ordering

$$\mathcal{T} = \sum_{\ell=0}^{\log(N)} 8^\ell \left( \left( \frac{N}{2^\ell} \right)^2 \right)^\beta$$

<table>
<thead>
<tr>
<th></th>
<th>Params</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{T}_{seq}$</td>
<td>$\beta = 3$</td>
<td>$N^6$</td>
</tr>
<tr>
<td>$\mathcal{T}_{node}$</td>
<td>$\beta = 1$</td>
<td>$N^3$</td>
</tr>
<tr>
<td></td>
<td>$\beta = 2$</td>
<td>$N^4$</td>
</tr>
</tbody>
</table>

- **Node Parallelism**: each front can be factorized in parallel:
  - for Cholesky $\beta = 1$
  - for (Sca)LAPACK like LU and QR $2 \leq \beta \leq 3$
Cubic domain of dimension $N$, with **Nested Dissection** ordering

$$\mathcal{T} = \sum_{\ell=0}^{\log(N)} 8^\ell \alpha \left( \left( \frac{N}{2^\ell} \right)^2 \right)^\beta$$

<table>
<thead>
<tr>
<th></th>
<th>Params</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{T}_{seq}$</td>
<td>$\alpha = 1$, $\beta = 3$</td>
<td>$N^6$</td>
</tr>
<tr>
<td>$\mathcal{T}_{node}$</td>
<td>$\alpha = 1$, $\beta = 1$</td>
<td>$N^3$</td>
</tr>
<tr>
<td></td>
<td>$\alpha = 1$, $\beta = 2$</td>
<td>$N^4$</td>
</tr>
<tr>
<td>$\mathcal{T}_{tree+node}$</td>
<td>$\alpha = 0$, $\beta = 1$</td>
<td>$N^2$</td>
</tr>
<tr>
<td></td>
<td>$\alpha = 0$, $\beta = 2$</td>
<td>$N^4$</td>
</tr>
</tbody>
</table>

- **Node Parallelism**: each front can be factorized in parallel:
  - for Cholesky $\beta = 1$
  - for (Sca)LAPACK like LU and QR $2 \leq \beta \leq 3$

- **Tree Parallelism**: multiple branches can be traversed concurrently: $\alpha = 0$
In the (Sca)LAPACK panelwise factorization, the panel reduction is dominated by communications and thus hardly parallelizable → execution time lower bound is $O(m^{\beta=2})$

Tiled algorithms can be used instead where the matrix is decomposed into (square or moderately rectangular) blocks whose elimination and corresponding updates can be pipelined or done concurrently


With such algorithms the lower bound for the parallel execution time is $O(m^{\beta=1})$*

* Bouwmeester et al., 2011.
Workload is expressed as a Directed Acyclic Graph of tasks

- Inherently expresses *concurrency* and *data flow*
- *Asynchronous* execution
- Allows for *dynamic* scheduling policies
The classical approach is based on a mixture of technologies (e.g., MPI+OpenMP+CUDA) which
- requires a big programming effort
- is difficult to maintain and update
- is prone to (performance) portability issues
The classical approach is based on a mixture of technologies (e.g., MPI+OpenMP+CUDA) which
- requires a big programming effort
- is difficult to maintain and update
- is prone to (performance) portability issues

runtimes provide an abstraction layer that hides the architecture details:
- portable programming interface
- handling of data
- support for multiple architectures
- customizable scheduling policies
- ...
Elimination tree is translated into a DAG of tasks which provides a consistent expression of tree and node parallelism

*Augonnet et al., 2011.*
Improved node and tree parallelism bring great benefit to small size and strongly overdetermined problems.

Speedups are uniform for all tested matrices.

Performance ranges from 46% to 66% of the peak.

• Very high computing power (O(1) Tflop/s)
• Very high memory bandwidth (O(100) GB/s)
• Very convenient Gflops/s/Watt ratio (O(10))

In principle, as simple as providing the runtime with GPU for single tasks.

Because we want to use all the available resources, we have to deal with heterogeneity

• in the system: PUs and memories with different speeds and capabilities
• in the workload: operations with different properties (e.g., PU or memory-bound)
• **Granularity**: GPUs require coarser grained tasks to achieve full speed:

  ➔ use a hierarchical and dynamic partitioning scheme to create tasks that are suited for both CPUs and GPUs. This is achieved by inserting partitioning and unpartitioning tasks in the workflow

---

• **Granularity**: GPUs require coarser grained tasks to achieve full speed:
  ➔ use a hierarchical and dynamic partitioning scheme to create tasks that are suited for both CPUs and GPUs. This is achieved by inserting partitioning and unpartitioning tasks in the workflow.

• **Scheduling**: account for different computing capabilities and different tasks characteristics while maximizing concurrency:
  ➔ use and improve the HeteroPrio* scheduling policy which maximizes the matching between tasks and PUs.

---

*Agullo, Bramas, et al., 2014*
• **Granularity**: GPUs require coarser grained tasks to achieve full speed:
  ➔ use a hierarchical and dynamic partitioning scheme to create tasks that are suited for both CPUs and GPUs. This is achieved by inserting partitioning and unpartitioning tasks in the workflow.

• **Scheduling**: account for different computing capabilities and different tasks characteristics while maximizing concurrency:
  ➔ use and improve the HeteroPrio* scheduling policy which maximizes the matching between tasks and PUs.

• **Communications**: deal with the cost of host-to-device data transfers:
  ➔ anticipate the assignment of tasks in order to prefetch the data.

---


---

*Agullo, Bramas, et al., 2014*
Thanks to IDRIS, Plafrim, CALMIP and GENCI for providing access to the resources.

In sequential the memory consumption varies greatly because

- fronts are allocated dynamically during the factorization
- each front is composed of a permanent (the factors) and temporary (the contribution block) memory
In sequential the memory consumption varies greatly because
- fronts are allocated dynamically during the factorization
- each front is composed of a permanent (the factors) and temporary (the contribution block) memory
In sequential the memory consumption varies greatly because
- fronts are allocated dynamically during the factorization
- each front is composed of a permanent (the factors) and temporary (the contribution block) memory

Parallelism increases the memory consumption
- **Node parallelism**: it requires no extra memory but it has limited scalability because it involves communications.
- **Tree parallelism**: embarrassingly parallel computations but requires extra memory because more fronts are allocated simultaneously.

**Objective**: Achieve the parallel factorization without exceeding a prescribed memory constraint $S_{\text{max}} = S_{\text{seq}}$.

**Methodology**: use scheduling/mapping methods that use tree parallelism as much as possible but temporarily/locally reduce it when the memory budget is exceeded. That means trading parallelism for memory.
- **Node parallelism**: it requires no extra memory but it has limited scalability because it involves communications.

- **Tree parallelism**: embarrassingly parallel computations but requires *extra memory* because more fronts are allocated simultaneously.

**Objective**: Achieve the parallel factorization without exceeding a prescribed memory constraint $S_{max} = \alpha S_{seq}, \alpha \geq 1$.
• **Node parallelism**: it requires no extra memory but it has limited scalability because it involves communications

• **Tree parallelism**: embarrassingly parallel computations but requires *extra memory* because more fronts are allocated simultaneously

**Objective**: Achieve the parallel factorization without exceeding a prescribed memory constraint $S_{max} = \alpha S_{seq}$, $\alpha \geq 1$

**Methodology**: use scheduling/mapping methods that use tree parallelism as much as possible but temporarily/locally reduce it when the memory budget is exceeded. That means *trading parallelism for memory*

---

*Éyraud-Dubois et al., 2015*
In a shared memory setting

- tree parallelism is temporarily turned off **dynamically** when the memory limit is reached
- extra dependencies added to the DAG to avoid memory deadlocks
In a shared memory setting

- tree parallelism is temporarily turned off *dynamically* when the memory limit is reached
- *extra dependencies* added to the DAG to avoid memory deadlocks
In a shared memory setting
- tree parallelism is temporarily turned off **dynamically** when the memory limit is reached
- extra dependencies added to the DAG to avoid memory deadlocks

In a distributed memory setting*
- the memory budget has to be taken into account in the nodes-to-processes **mapping**
- extra dependencies added to prevent the concurrent processing of sibling subtrees if this violates the memory constraint

*Agullo, 2008
In a **shared memory** setting
- tree parallelism is temporarily turned off *dynamically* when the memory limit is reached
- *extra dependencies* added to the DAG to avoid memory deadlocks

In a **distributed memory** setting*
- the memory budget has to be taken into account in the nodes-to-processes *mapping*
- *extra dependencies* added to prevent the concurrent processing of sibling subtrees if this violates the memory constraint

---

*Agullo, 2008*
- Tighter memory bound → less concurrency → slower execution
- In practice the execution time is increased only for very small matrices or very narrow/unbalanced elimination trees

- Performance varies smoothly with the memory constraint
- Variant with aggregation allows for better performance with the same memory budget

SCALABLE COMPLEXITY


Take a dense matrix $B$ of size $b \times b$ and its SVD $B = USV^T$.
Take a dense matrix $B$ of size $b \times b$ and its SVD $B = U S V^T$

$$B = U_1 S_1 V_1^T + U_2 S_2 V_2^T \quad \text{with} \quad S_1(k,k) = \sigma_k > \varepsilon, \quad S_2(1,1) = \sigma_{k+1} \leq \varepsilon$$
Take a dense matrix $B$ of size $b \times b$ and its SVD $B = USV^T$

$$B = U_1 S_1 V_1^T + U_2 S_2 V_2^T$$

with $S_1(k, k) = \sigma_k > \varepsilon$, $S_2(1, 1) = \sigma_{k+1} \leq \varepsilon$

If $\tilde{B} = U_1 S_1 V_1^T$ then

$$\|B - \tilde{B}\|_2 = \|U_2 S_2 V_2^T\|_2 = \sigma_{k+1} \leq \varepsilon$$

If the singular values of $B$ decay very fast (e.g. exponentially) then $k \ll b$ even for very small $\varepsilon$ (e.g. $10^{-14}$) → memory and CPU consumption can be reduced considerably with a controlled loss of accuracy ($\leq \varepsilon$) if $\tilde{B}$ is used instead of $B$
Our matrices are not low-rank but in some applications they exhibit low-rank blocks.

A block $B|_s$ represents the interaction between two subdomains $\sigma$ and $\tau$. If they have a small diameter and are far away their interaction is weak $\rightarrow$ rank is low.

The block-admissibility condition formalizes this intuition:

$$\sigma \times \tau \text{ is admissible } \iff \max (\text{diam}(\sigma), \text{diam}(\tau)) \leq \eta \dist(\sigma, \tau)$$
**H and BLR matrices**

- Nearly linear complexity*
- Complex, hierarchical structure

*Börm et al., 2003*
\( \mathcal{H} \) and BLR matrices

- Nearly linear complexity*
- Complex, hierarchical structure

BLR is a compromise between complexity and performance:
- Small blocks → can fit on single shared-memory node
- No global order between blocks → flexible data distribution
- Easy to handle numerical pivoting

*Börm et al., 2003
Use the BLR format on fronts:

- Blocking in an **algebraic** setting
  - Extract the front subgraph
  - Patch it to improve regularity and better capture the geometry
  - Run a k-way partitioning method

Matrices are assembled in full-rank blocks are compressed after panel reduction (with pivoting) before the trailing submatrix update. The use of low-rank approximations reduces operations, factors storage and communications.

Use the BLR format on fronts:

- Blocking in an algebraic setting
  - Extract the front subgraph
  - Patch it to improve regularity and better capture the geometry
  - Run a k-way partitioning method

- Matrices are assembled in full-rank
Use the BLR format on fronts:

- Blocking in an *algebraic* setting
  - Extract the front subgraph
  - Patch it to improve regularity and better capture the geometry
  - Run a k-way partitioning method

- Matrices are assembled in full-rank
- blocks are compressed after panel reduction (with *pivoting*) before the trailing submatrix update
Use the BLR format on fronts:

- Blocking in an algebraic setting
  - Extract the front subgraph
  - Patch it to improve regularity and better capture the geometry
  - Run a k-way partitioning method

- Matrices are assembled in full-rank

- Blocks are compressed after panel reduction (with pivoting) before the trailing submatrix update

The use of low-rank approximations reduces operations, factors storage and communications

3D seismic modeling problem 10Hz: $n = 17.2M$, $nz = 446M$

BLR $\varepsilon = 10^{-3} \quad \rightarrow \quad$ flops reduction $\times \approx 12.5$

Lower time reduction wrt flops:
- lower efficiency of computations due to smaller granularity
- relatively higher cost of communications
- potentially higher load imbalance

\( \mathcal{H} \)-matrix assumptions that do not hold for BLR

- The number of blocks of a given size in a row/column is constant
- The rank of blocks is upper bounded by a value \( r < b \)
**BLR: complexity**

**H**-matrix assumptions that do not hold for BLR

- The number of blocks of a given size in a row/column is constant
- The rank of blocks is upper bounded by a value $r < b$

---

**Theorem**

For any matrix, it is possible to build a BLR blocking such that

- The number of non-admissible blocks in a row/column is bounded by a constant $q$
- The rank of admissible blocks is $r = O(r_{\max}^H) < b$

Such a blocking is called BLR-admissible

---

Complexity of the factorization for a *dense* matrix of size $m$ with blocks or rank $r$ and size $b = O(\sqrt{rm})$

<table>
<thead>
<tr>
<th></th>
<th>operations (OPC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR</td>
<td>$O(m^3)$</td>
</tr>
<tr>
<td>BLR</td>
<td>$O(m^{2.5}\sqrt{r})$</td>
</tr>
</tbody>
</table>
Complexity of the factorization for a **dense** matrix of size $m$ with blocks or rank $r$ and size $b = O(\sqrt{rm})$

<table>
<thead>
<tr>
<th></th>
<th>operations (OPC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR</td>
<td>$O(m^3)$</td>
</tr>
<tr>
<td>BLR</td>
<td>$O(m^{2.5} \sqrt{r})$</td>
</tr>
</tbody>
</table>

Complexity of the factorization for a **sparse** matrix of size $n = N^3$

<table>
<thead>
<tr>
<th></th>
<th>operations (OPC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>BLR</td>
<td>$O(n^{1.67} \sqrt{r})$</td>
</tr>
</tbody>
</table>
Complexity of the factorization for a **dense** matrix of size $m$ with blocks or rank $r$ and size $b = O(\sqrt{rm})$

<table>
<thead>
<tr>
<th></th>
<th>operations (OPC)</th>
<th>factor size (NNZ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR</td>
<td>$O(m^3)$</td>
<td>$O(m^2)$</td>
</tr>
<tr>
<td>BLR</td>
<td>$O(m^{2.5}\sqrt{r})$</td>
<td>$O(m^{1.5}\sqrt{r})$</td>
</tr>
</tbody>
</table>

Complexity of the factorization for a **sparse** matrix of size $n = N^3$

<table>
<thead>
<tr>
<th></th>
<th>operations (OPC)</th>
<th>factor size (NNZ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR</td>
<td>$O(n^2)$</td>
<td>$O(n^{1.33})$</td>
</tr>
<tr>
<td>BLR</td>
<td>$O(n^{1.67}\sqrt{r})$</td>
<td>$O(n \max (\sqrt{r}, \log n))$</td>
</tr>
</tbody>
</table>

Experimental results are in very good agreement with theory.
<table>
<thead>
<tr>
<th>Variant name</th>
<th>$C$</th>
<th>Matrix 10Hz ($\varepsilon = 10^{-3}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full-Rank</td>
<td>$n^2$</td>
<td>2703</td>
</tr>
<tr>
<td>FSCU</td>
<td>$n^{1.67} \sqrt{r}$</td>
<td>222</td>
</tr>
</tbody>
</table>
**UFSC** improves **performance** thanks to a left-looking approach which reduces memory transfers

<table>
<thead>
<tr>
<th>Variant name</th>
<th>( C )</th>
<th>Matrix 10Hz ( (\varepsilon = 10^{-3}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full-Rank</td>
<td>( n^2 )</td>
<td>2703</td>
</tr>
<tr>
<td>FSCU</td>
<td>( n^{1.67} \sqrt{r} )</td>
<td>222</td>
</tr>
<tr>
<td>UFSC</td>
<td>( = )</td>
<td>( = )</td>
</tr>
</tbody>
</table>
LUA improves performance because it accumulates multiple low-rank updates and applies them at once increasing the granularity of operations

<table>
<thead>
<tr>
<th>Variant name</th>
<th>$C$</th>
<th>Matrix 10Hz ($\varepsilon = 10^{-3}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TFlops</td>
<td>Time (sec.)</td>
</tr>
<tr>
<td>Full-Rank</td>
<td>$n^2$</td>
<td>2703</td>
</tr>
<tr>
<td>FSCU</td>
<td>$n^{1.67}\sqrt{r}$</td>
<td>222</td>
</tr>
<tr>
<td>UFSC</td>
<td>$=\sqrt{r}$</td>
<td>$=\sqrt{r}$</td>
</tr>
<tr>
<td>UFSC+LUA</td>
<td>$=\sqrt{r}$</td>
<td>$=\sqrt{r}$</td>
</tr>
</tbody>
</table>

**LUAR** reduces complexity because recompresses accumulated updates on the fly

<table>
<thead>
<tr>
<th>Variant name</th>
<th>$C$</th>
<th>Matrix 10Hz ($\varepsilon = 10^{-3}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full-Rank</td>
<td>$n^2$</td>
<td>TFlops: 2703</td>
</tr>
<tr>
<td></td>
<td>$n^{1.67}\sqrt{r}$</td>
<td>Time (sec.): 5649</td>
</tr>
<tr>
<td>FSCU</td>
<td>$n^{1.67}\sqrt{r}$</td>
<td>TFlops: 222</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time (sec.): 1497</td>
</tr>
<tr>
<td>UFSC</td>
<td>=</td>
<td>TFlops: =</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time (sec.): 1334</td>
</tr>
<tr>
<td>UFSC+LUA</td>
<td>=</td>
<td>TFlops: =</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time (sec.): 1245</td>
</tr>
<tr>
<td>UFSC+LUAR</td>
<td>$n^{1.55}r^{0.66}$</td>
<td>TFlops: 146</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time (sec.): 1196</td>
</tr>
</tbody>
</table>

**UFCS** reduces complexity because solve operations are also done in low-rank (needs restricted pivoting)

<table>
<thead>
<tr>
<th>Variant name</th>
<th>$C$</th>
<th>Matrix 10Hz ($\varepsilon = 10^{-3}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>TFlops</td>
</tr>
<tr>
<td>Full-Rank</td>
<td>$n^2$</td>
<td>2703</td>
</tr>
<tr>
<td>FSCU</td>
<td>$n^{1.67} \sqrt{r}$</td>
<td>222</td>
</tr>
<tr>
<td>UFSC</td>
<td>$n^{1.67} \sqrt{r}$</td>
<td>222</td>
</tr>
<tr>
<td>UFSC+LUAR</td>
<td>$n^{1.33} r$</td>
<td>100</td>
</tr>
</tbody>
</table>

BLR variants: Experimental results

Performance results on a 24-cores system

- Up to 18.8 times faster than the full-rank solver
- Up to 4.3 times faster than the basic BLR variant
FUTURE WORK
Applications:
- increasingly large scale
- novel numerical features from Multi-physics/scale modeling, High Order schemes etc.

Solvers:
- increasingly complex and difficult to implement in an efficient and portable way
- irregular and unpredictable because of pivoting or low-rank compression rates are unknown beforehand

Supercomputers:
- Heterogeneous: different processing units, memory and interconnects with different speeds and capabilities
- Irregular: units run at different frequencies, dynamic thermal throttling, different hard/soft configurations
Take *unpredictability* into account in the generation of workload/DAG

Hierarchical tasks: macro-tasks which, upon execution, generate other tasks
Take **unpredictability** into account in the generation of workload/DAG

**Hierarchical tasks**: macro-tasks which, upon execution, generate other tasks
Scalable performance

Take **unpredictability** into account in the generation of workload/DAG

**Hierarchical tasks**: macro-tasks which, upon execution, generate other tasks

- Enables dynamic setting of **granularity** of tasks, algorithm and **parameters** and data/work **distribution**
Take **unpredictability** into account in the generation of workload/DAG

**Hierarchical tasks**: macro-tasks which, upon execution, generate other tasks

- enables dynamic setting of **granularity** of tasks, algorithm and **parameters** and data/work distribution
- improves scalability of the runtime

The StarPU team is implementing this into the **Bubbles** feature
CPU speed grows at a much faster pace than memory and network → Communication Avoiding (CA) algorithms achieve better scalability by reducing communications and synchronizations

→ CA pivoting strategies for $LU$, $LDL^T$ and QR factorizations

→ 2.5D or 3D* algorithms that trade memory and flops for communications
  - take the extra memory into account in memory-aware execution
  - choose dynamically the best option using hierarchical tasks

*Solomonik et al., 2011; Sao et al., 2018.
How to improve complexity of BLR?

BLR matrix

$\mathcal{H}$-matrix
How to improve complexity of BLR?

- Simple way to **finely control** the desired complexity
- Block size is small enough to fit in a single node
- Number of blocks per row/column is large enough to have a balanced distribution
\[ C_{\text{dense}} = m^{\frac{\ell+3}{\ell+1}} r^{\frac{2\ell}{\ell+1}} \]

\[ C_{\text{sparse}} = \sum_{\ell=0}^{\log(N)} 8^\ell \left( \left( \frac{N}{2^\ell} \right)^2 \right)^3 \]

<table>
<thead>
<tr>
<th></th>
<th>BLR</th>
<th>BLR-2</th>
<th>BLR-3</th>
<th>BLR-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense</td>
<td>( m^2 )</td>
<td>( m^{1.67} )</td>
<td>( m^{1.5} )</td>
<td>( m^{1.4} )</td>
</tr>
<tr>
<td>Sparse (3D)</td>
<td>( n^{1.33} )</td>
<td>( n^{1.11} )</td>
<td>( n\log(n) )</td>
<td>( n )</td>
</tr>
</tbody>
</table>

- Only four levels are enough to achieve linear complexity in the sparse factorization as good as \( H \)-matrix in sparse solvers but much simpler.
- most of the potential gain achieved in 2/3 levels

CONCLUSIONS
Sparse direct methods are regarded as extremely robust and reliable tools but also heavy on flops and memory.

We have demonstrated that
Sparse direct methods are regarded as extremely robust and reliable tools but also heavy on flops and memory.

We have demonstrated that

- It is possible to improve their performance scalability by making use of:
  - algorithms that improve the amount of available concurrency
  - parallel programming models that avoid unnecessary synchronizations and allow for dynamic execution patterns
  - parallel programming tools (runtimes) that ease the development and improve the performance and portability on modern, heterogeneous machines

It is possible to improve the memory scalability at the price of a controlled performance loss.

It is possible to reduce the complexity using low-rank techniques that allow for considerable gains in terms of flops and storage depending on the desired solution accuracy.
Sparse direct methods are regarded as extremely robust and reliable tools but also heavy on flops and memory.

We have demonstrated that

- It is possible to improve their performance scalability by making use of
  - algorithms that improve the amount of available concurrency
Sparse direct methods are regarded as extremely robust and reliable tools but also heavy on flops and memory.

We have demonstrated that

- It is possible to improve their performance scalability by making use of
  - algorithms that improve the amount of available concurrency
  - parallel programming models that avoid unnecessary synchronizations and allow for dynamic execution patterns
- It is possible to reduce the complexity using low-rank techniques that allow for considerable gains in terms of flops and storage depending on the desired solution accuracy.
Sparse direct methods are regarded as extremely robust and reliable tools but also heavy on flops and memory.

We have demonstrated that:

- It is possible to improve their performance scalability by making use of:
  - algorithms that improve the amount of available concurrency
  - parallel programming models that avoid unnecessary synchronizations and allow for dynamic execution patterns
  - parallel programming tools (runtimes) that ease the development and improve the performance and portability on modern, heterogeneous machines

- It is possible to reduce the complexity using low-rank techniques that allow for considerable gains in terms of flops and storage depending on the desired solution accuracy.
Sparse direct methods are regarded as extremely robust and reliable tools but also heavy on flops and memory.

We have demonstrated that

- It is possible to improve their performance scalability by making use of
  - algorithms that improve the amount of available concurrency
  - parallel programming models that avoid unnecessary synchronizations and allow for dynamic execution patterns
  - parallel programming tools (runtimes) that ease the development and improve the performance and portability on modern, heterogeneous machines

- It is possible to improve the memory scalability at the price of a controlled performance loss
Sparse direct methods are regarded as extremely **robust** and **reliable** tools but also **heavy on flops and memory**

We have demonstrated that

- It is possible to improve their **performance scalability** by making use of
  - algorithms that improve the amount of available **concurrency**
  - parallel programming models that avoid unnecessary synchronizations and allow for dynamic execution patterns
  - parallel programming tools (runtimes) that ease the development and improve the performance and portability on modern, heterogeneous machines

- It is possible to improve the **memory scalability** at the price of a controlled performance loss

- It is possible to reduce the **complexity** using **low-rank** techniques that allow for considerable gains in terms of flops and storage depending on the desired solution accuracy
Thanks! Questions?
PUBLISHED WORK


