Evolutions of MUMPS towards multicore architectures

M.W. Sid-Lakhdar, Ecole Normale Supérieure de Lyon

MUMPS Users Group Meeting, May 29-30, 2013
Context: the multifrontal method (Duff & Reid ’83)

Storage is composed of two parts:
- Factors
- Active memory

At each frontal matrix:
- compute factors: $L_{11}, L_{21}, U_{11}, U_{12}$
- build a Schur complement
Context and objectives

- **Context:**
  - MUMPS: MUltifrontal Massively Parallel Solver
  - Node and Tree parallelism at process level (MPI)
  - Node parallelism at thread level (threaded BLAS + OpenMP)

- **Questions/Objectives:**
  - Is it possible to do better on multicore systems by introducing tree parallelism at thread level?
  - Can we improve parallel factorization kernels in multithreaded and distributed-memory environments?
1. **Tree parallelism at thread level** at bottom of the tree (inside one MPI process)
2. **Node parallelism at thread level** at middle of the tree: Multithreaded dense factorization kernels
3. **Node parallelism at process level** at top of the tree: Distributed and hybrid dense factorization kernels
Introduction

1. Tree parallelism at thread level

2. Multithreaded dense factorization kernels

3. Distributed and hybrid dense factorization kernels

Conclusion
Introduction

1. Tree parallelism at thread level

2. Multithreaded dense factorization kernels

3. Distributed and hybrid dense factorization kernels

Conclusion
$\mathcal{L}_{mpi} \leftarrow \text{root(s) of the assembly tree}$

Estimate cost of all subtrees

repeat

Find node $N$ in $\mathcal{L}_{mpi}$ with the heaviest subtree

$\mathcal{L}_{mpi} \leftarrow \mathcal{L}_{mpi} \cup \{\text{children of } N\} \setminus \{N\}$

Tentative mapping of $\mathcal{L}_{mpi}$ subtrees onto the processors

until Acceptance criterion is reached
Tree parallelism: Geist-Ng algorithm

\[ \mathcal{L}_{mpi} \leftarrow \text{root(s) of the assembly tree} \]

Estimate cost of all subtrees

repeat

Find node \( \mathcal{N} \) in \( \mathcal{L}_{mpi} \) with the heaviest subtree

\[ \mathcal{L}_{mpi} \leftarrow \mathcal{L}_{mpi} \cup \{ \text{children of } \mathcal{N} \} \setminus \{ \mathcal{N} \} \]

Tentative mapping of \( \mathcal{L}_{mpi} \) subtrees onto the processors

until Acceptance criterion is reached
\( \mathcal{L}_{mpi} \leftarrow \text{root(s) of the assembly tree} \)
Estimate cost of all subtrees

repeat
  Find node \( \mathcal{N} \) in \( \mathcal{L}_{mpi} \) with the heaviest subtree
  \( \mathcal{L}_{mpi} \leftarrow \mathcal{L}_{mpi} \cup \{ \text{children of } \mathcal{N} \} \setminus \{ \mathcal{N} \} \)
  Tentative mapping of \( \mathcal{L}_{mpi} \) subtrees onto the processors
until Acceptance criterion is reached
\[ \mathcal{L}_{mpi} \leftarrow \text{root(s) of the assembly tree} \]

Estimate \textit{cost} of all subtrees

\begin{itemize}
  \item \textbf{repeat}
  \begin{itemize}
    \item Find node \( \mathcal{N} \) in \( \mathcal{L}_{mpi} \) with the heaviest subtree
    \[ \mathcal{L}_{mpi} \leftarrow \mathcal{L}_{mpi} \cup \{ \text{children of } \mathcal{N} \} \setminus \{ \mathcal{N} \} \]
    \item Tentative mapping of \( \mathcal{L}_{mpi} \) subtrees onto the processors
  \end{itemize}
\end{itemize}

\begin{itemize}
  \item \textbf{until} \textit{Acceptance criterion} is reached
\end{itemize}
\[ \mathcal{L}_{mpi} \leftarrow \text{root(s) of the assembly tree} \]

Estimate cost of all subtrees

**repeat**

Find node \( \mathcal{N} \) in \( \mathcal{L}_{mpi} \) with the heaviest subtree

\[ \mathcal{L}_{mpi} \leftarrow \mathcal{L}_{mpi} \cup \{ \text{children of } \mathcal{N} \} \setminus \{ \mathcal{N} \} \]

Tentative mapping of \( \mathcal{L}_{mpi} \) subtrees onto the processors

**until** Acceptance criterion is reached
\[ \mathcal{L}_{mpi} \leftarrow \text{root(s) of the assembly tree} \]

Estimate cost of all subtrees

repeat

Find node \( N \) in \( \mathcal{L}_{mpi} \) with the heaviest subtree

\[ \mathcal{L}_{mpi} \leftarrow \mathcal{L}_{mpi} \cup \{ \text{children of } N \} \setminus \{N\} \]

Tentative mapping of \( \mathcal{L}_{mpi} \) subtrees onto the processors

until Acceptance criterion is reached
Tree parallelism: Geist-Ng algorithm

\[ L_{mpi} \leftarrow \text{root(s) of the assembly tree} \]

Estimate cost of all subtrees

repeat
  Find node \( \mathcal{N} \) in \( L_{mpi} \) with the heaviest subtree
  \[ L_{mpi} \leftarrow L_{mpi} \cup \{ \text{children of } \mathcal{N} \} \setminus \{ \mathcal{N} \} \]
  Tentative mapping of \( L_{mpi} \) subtrees onto the processors
until Acceptance criterion is reached
\( L_{mpi} \leftarrow \text{root(s) of the assembly tree} \)

Estimate cost of all subtrees

repeat
  Find node \( N \) in \( L_{mpi} \) with the heaviest subtree

  \( L_{mpi} \leftarrow L_{mpi} \cup \{ \text{children of } N \} \setminus \{N\} \)

  Tentative mapping of \( L_{mpi} \) subtrees onto the processors

until Acceptance criterion is reached
$\mathcal{L}_{mpi} \leftarrow \text{root(s) of the assembly tree}$

Estimate cost of all subtrees

repeat

Find node $\mathcal{N}$ in $\mathcal{L}_{mpi}$ with the heaviest subtree

$\mathcal{L}_{mpi} \leftarrow \mathcal{L}_{mpi} \cup \{ \text{children of } \mathcal{N} \} \setminus \{ \mathcal{N} \}$

Tentative mapping of $\mathcal{L}_{mpi}$ subtrees onto the processors

until Acceptance criterion is reached
\( \mathcal{L}_{mpi} \leftarrow \text{root(s) of the assembly tree} \)

Estimate cost of all subtrees

repeat
  Find node \( \mathcal{N} \) in \( \mathcal{L}_{mpi} \) with the heaviest subtree
  \( \mathcal{L}_{mpi} \leftarrow \mathcal{L}_{mpi} \cup \{ \text{children of } \mathcal{N} \} \setminus \{ \mathcal{N} \} \)
  Tentative mapping of \( \mathcal{L}_{mpi} \) subtrees onto the processors
until Acceptance criterion is reached
\[ \mathcal{L}_{mpi} \leftarrow \text{root(s) of the assembly tree} \]
Estimate cost of all subtrees

repeat
  Find node \( N \) in \( \mathcal{L}_{mpi} \) with the heaviest subtree
  \[ \mathcal{L}_{mpi} \leftarrow \mathcal{L}_{mpi} \cup \{ \text{children of } N \} \setminus \{ N \} \]
  Tentative mapping of \( \mathcal{L}_{mpi} \) subtrees onto the processors
until Acceptance criterion is reached
Shared-memory tree parallelism: \texttt{AlgTime} algorithm

- Study serial and threaded dense-matrix factorization kernels to predict performance on dense matrices
- \texttt{AlgTime}: simulate and implement $\mathcal{L}_{th}$ algorithms for computation on sparse matrices
  - \textbf{Under} $\mathcal{L}_{th}$: tree parallelism only (sequential BLAS)
  - \textbf{Above} $\mathcal{L}_{th}$: node parallelism only (threaded BLAS)

\texttt{AlgTime}: find the minimum of the blue curve
Memory management in multifrontal method

Standard Algorithm (without \textit{AlgTime})

\begin{itemize}
\item Factors
\item Active front
\item Contribution blocks
\end{itemize}

Shared workspace
Memory management in multifrontal method

Standard Algorithm (without ALGTIME)
Memory management in multifrontal method

Standard Algorithm (without ALGTIME)
Memory management in multifrontal method

Standard Algorithm (without \texttt{AlgTime})

![Diagram showing memory management in multifrontal method]

- **Factors**
- **Active front**
- **Contribution blocks**

- **Shared workspace**

- **Private workspace 0**
- **Private workspace 1**
Memory management in multifrontal method

Standard Algorithm (without ALGTime)
Memory management in multifrontal method

Standard Algorithm (without \textit{ALGTIME})
Standard Algorithm (without ALGTIME)
Memory management in multifrontal method

Standard Algorithm (without ALGTIME)

Factors
Active front
Contribution blocks

Shared workspace
Memory management in multifrontal method

Standard Algorithm (without \texttt{ALGTIME})
Memory management in multifrontal method

Standard Algorithm (without \texttt{ALGTIME})
Memory management in multifrontal method

Standard Algorithm (without \texttt{ALGTIME})

![Diagram showing memory management in multifrontal method](imageurl)
Memory management in multifrontal method

Standard Algorithm (without \textit{AlgTime})
Memory management in multifrontal method

Standard Algorithm (without \textsc{AlgTime})

Factors
Active front
Contribution blocks

Shared workspace

9/43 MUMPS Users Group Meeting, May 29-30, 2013
Memory management in multifrontal method

Standard Algorithm (without \textit{ALGTIME})

![Diagram showing memory management in multifrontal method]

- **Factors**
- **Active front**
- **Contribution blocks**

Shared workspace

Private workspace 0

Private workspace 1
Memory management in multifrontal method

Standard Algorithm (without \textit{\textsc{AlgTime}})

- **Factors**
- **Active front**
- **Contribution blocks**

Shared workspace
Memory management in multifrontal method

Standard Algorithm (without $\text{AlgTime}$)

Factors  
Active front  
Contribution blocks

Shared workspace
Standard Algorithm (without ALGTIME)
Use of a private workspace per thread (with \texttt{ALGTime})

Factors
Active front
Contribution blocks

Shared workspace

Private workspace 0
Private workspace 1
Use of a private workspace per thread (with AlgTime)

Shared workspace

Factors
Active front
Contribution blocks

Private workspace 0

Private workspace 1
Use of a private workspace per thread (with $\text{ALGTime}$)
Use of a private workspace per thread (with $\text{ALGTIME}$)
Use of a private workspace per thread (with $\text{ALGTIME}$)
Use of a private workspace per thread (with ALGTIME)
Memory management in multifrontal method

Use of a private workspace per thread (with \texttt{ALGTIME})
Use of a private workspace per thread (with \textsc{AlgTime})
Use of a private workspace per thread (with AlgTime)
Use of a private workspace per thread (with $\text{ALGTIME}$)

Factors
Active front
Contribution blocks

$\mathcal{L}_\text{th}$

Shared workspace

Private workspace 0
Private workspace 1
Use of a private workspace per thread (with \textit{AlgTime})
Use of a private workspace per thread (with \textit{AlgTime})
Memory management in multifrontal method

Use of a private workspace per thread (with \text{AlgTime})

\[ L_{th} \]

Factors
Active front
Contribution blocks

Shared workspace

Private workspace 0
Private workspace 1
Use of a private workspace per thread (with ALGTime)
Use of a private workspace per thread (with \textsc{AlgTime})
Use of a private workspace per thread (with $\text{AlgTime}$)

- **Active front**
- **Factors**
- **Contribution blocks**
Use of a private workspace per thread (with $\text{AlgTime}$)

Memory management in multifrontal method
Memory management in multifrontal method

Use of a private workspace per thread (with \textsc{AlgTime})
Memory management in multifrontal method

Use of a private workspace per thread (with $\text{ALGTIME}$)

Factors
Active front
Contribution blocks

Shared workspace

Private workspace 0
Private workspace 1
Use of a private workspace per thread (with \texttt{ALGTime})
Multicore architectures: from SMP to NUMA

SMP

NUMA

Bus Interconnect
Memory policy affinities: localalloc and interleave

- Shared workspace
  - Memory page on node 0
  - Memory page on node 1
- Private workspace 0
- Private workspace 1

Shared workspace (default)
Shared workspace (interleave)

Private workspaces are naturally allocated on the local NUMA memory bank of their corresponding thread (localalloc).
We choose to allocate pages of the shared workspace on all NUMA memory banks in a round-robin fashion (interleave).
• Private workspaces are naturally allocated on the local NUMA memory bank of their corresponding thread (localalloc)
• Pages of the shared workspace are allocated, by default, on the memory of the first thread touching them (OS default)
• Private workspaces are naturally allocated on the local NUMA memory bank of their corresponding thread (localalloc)

• We choose to allocate pages of the shared workspace on all NUMA memory banks in a round-robin fashion (interleave)
Results on hidalgo (Intel Xeon Nehalem (8 cores))

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Speed-Up on 8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>3Dspectralwave</td>
<td>6</td>
</tr>
<tr>
<td>AUDI.rsa</td>
<td>6</td>
</tr>
<tr>
<td>G3_circuit</td>
<td>6</td>
</tr>
<tr>
<td>Haltere</td>
<td>6</td>
</tr>
<tr>
<td>QIMONDA07</td>
<td>6</td>
</tr>
<tr>
<td>ultrasound80</td>
<td>6</td>
</tr>
<tr>
<td>GeoAzur_3D_32_32_32</td>
<td>6</td>
</tr>
<tr>
<td>GeoAzur_3D_64_64_64</td>
<td>6</td>
</tr>
<tr>
<td>GeoAzur_2D_512_512</td>
<td>6</td>
</tr>
<tr>
<td>GeoAzur_2D_2048_2048</td>
<td>6</td>
</tr>
</tbody>
</table>

Graph showing speed-up on 8 cores for various datasets.

- 1 MPI + BLAS + OpenMP
- 1 MPI + BLAS + OpenMP + L0
- 1 MPI + BLAS + OpenMP + L0 + interleave
## Results on dude (AMD Istanbul (24 cores, NUMA))

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Serial reference</th>
<th>Threading BLAS + OpenMP</th>
<th>AlgTime algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Interleave</td>
<td>Interleave</td>
</tr>
<tr>
<td></td>
<td></td>
<td>off</td>
<td>on</td>
</tr>
<tr>
<td>AUDI</td>
<td>1535.78</td>
<td>231.76</td>
<td>225.47</td>
</tr>
<tr>
<td>conv3D64</td>
<td>3001.43</td>
<td>497.52</td>
<td>496.87</td>
</tr>
<tr>
<td>SERENA</td>
<td>7845.43</td>
<td>1081.42</td>
<td>1006.66</td>
</tr>
<tr>
<td>dielFilterV3real</td>
<td>324.50</td>
<td>68.42</td>
<td>69.91</td>
</tr>
<tr>
<td>HALTERE</td>
<td>867.47</td>
<td>133.61</td>
<td>135.20</td>
</tr>
<tr>
<td>mono_500Hz</td>
<td>42.39</td>
<td>11.09</td>
<td>11.19</td>
</tr>
<tr>
<td>G3_circuit</td>
<td>24.92</td>
<td>14.88</td>
<td>14.67</td>
</tr>
<tr>
<td>QIMONDA07</td>
<td>31.82</td>
<td>52.57</td>
<td>55.45</td>
</tr>
<tr>
<td>GeoAzur_3D_64_64_64</td>
<td>1774.57</td>
<td>221.73</td>
<td>225.45</td>
</tr>
<tr>
<td>GeoAzur_2D_2048_2048</td>
<td>309.64</td>
<td>96.26</td>
<td>436.65</td>
</tr>
</tbody>
</table>
Results on dude (AMD Istanbul (24 cores, NUMA))

<table>
<thead>
<tr>
<th></th>
<th>Node parallelism only</th>
<th>$\mathcal{L}_{th}$-based algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Threaded BLAS + OpenMP directives)</td>
<td>($\text{ALGTIME}$)</td>
</tr>
<tr>
<td>Time</td>
<td>without interleaving</td>
<td>with interleaving</td>
</tr>
<tr>
<td>Under $\mathcal{L}_{th}$</td>
<td>109.81</td>
<td>151.54</td>
</tr>
<tr>
<td>Above $\mathcal{L}_{th}$</td>
<td>121.95</td>
<td>73.93</td>
</tr>
<tr>
<td>Total</td>
<td>231.76</td>
<td>225.47</td>
</tr>
</tbody>
</table>

On large matrices (here AUDI), $\text{ALGTIME}$ and $\text{interleave}$ alone are not exceptional, but their combination brings a huge gain.
Limits of the approach

- $L_{th}$ layer represents a costly synchronization barrier for threads

- There is a brutal move from the use of sequential BLASs to fully multithreaded BLASs
• When a thread finishes its set of tasks under $\mathcal{L}_{th}$ . . .
  
  . . . it yields the cores at its disposal to the threads that needs them

• Between two BLAS calls, these threads will notice the newly available cores and will use them in future BLAS calls
• When a thread finishes its set of tasks under $L_{th}$ . . . 
  ...it yields the cores at its disposal to the threads that needs them
• Between two BLAS calls, these threads will notice the newly available cores and will use them in future BLAS calls
• When a thread finishes its set of tasks under $L_{th}$ . . .
  . . . it yields the cores at its disposal to the threads that needs them

• Between two BLAS calls, these threads will notice the newly available cores and will use them in future BLAS calls
Idle Core Recycling (ICR)

- When a thread finishes its set of tasks under $L_{th}$ ... 
  ...it yields the cores at its disposal to the threads that needs them
- Between two BLAS calls, these threads will notice the newly available cores and will use them in future BLAS calls
Idle Core Recycling (ICR)

- When a thread finishes its set of tasks under $\mathcal{L}_{th}$ ... 
  ...it yields the cores at its disposal to the threads that needs them
- Between two BLAS calls, these threads will notice the newly available cores and will use them in future BLAS calls
Idle Core Recycling (ICR)

- When a thread finishes its set of tasks under $\mathcal{L}_{th}$ . . .
  
  . . . it yields the cores at its disposal to the threads that needs them.

- Between two BLAS calls, these threads will notice the newly available cores and will use them in future BLAS calls.
Idle Core Recycling (ICR)

- When a thread finishes its set of tasks under $\mathcal{L}_{th} \ldots$
  
  \ldots it yields the cores at its disposal to the threads that needs them

- Between two BLAS calls, these threads will notice the newly available cores and will use them in future BLAS calls
**Idle Core Recycling (ICR)**

- When a thread finishes its set of tasks under $\mathcal{L}_{th}$ . . .
  
  . . . it yields the cores at its disposal to the threads that needs them

- Between two BLAS calls, these threads will notice the newly available cores and will use them in future BLAS calls
Robustness and gains

- **Robustness**
  ... against variations of $L_{th}$
  ... against numerical pivoting issues

- **Gains**
  ... due to a smoother use of multithreaded BLASs ...
  ... by intentionally unbalancing (elevating) $L_{th}$!
Introduction

1. Tree parallelism at thread level

2. Multithreaded dense factorization kernels

3. Distributed and hybrid dense factorization kernels

Conclusion
In shared-memory environments, the large TRSM and GEMM calls reach a good efficiency, but it is critical to have an efficient multithreaded factorization of the fully-summed rows.
In shared-memory environments, the large TRSM and GEMM calls reach a good efficiency, but it is critical to have an efficient multithreaded factorization of the fully-summed rows.
Shared-memory factorization algorithm in MUMPS

Structure of a frontal matrix in the multifrontal tree:

In shared-memory environments, the large TRSM and GEMM calls reach a good efficiency, but it is critical to have an efficient multithreaded factorization of the fully-summed rows.
In shared-memory environments, the large TRSM and GEMM calls reach a good efficiency, but it is critical to have an efficient multithreaded factorization of the fully-summed rows.
In shared-memory environments, the large TRSM and GEMM calls reach a good efficiency, but it is critical to have an efficient multithreaded factorization of the fully-summed rows.
Shared-memory factorization algorithm in MUMPS

Structure of a frontal matrix in the multifrontal tree:

In shared-memory environments, the large TRSM and GEMM calls reach a good efficiency, but it is critical to have an efficient multithreaded factorization of the fully-summed rows.
In shared-memory environments, the large TRSM and GEMM calls reach a good efficiency, but it is critical to have an efficient multithreaded factorization of the fully-summed rows.
Shared-memory factorization algorithm in MUMPS

Structure of a frontal matrix in the multifrontal tree:

In shared-memory environments, the large TRSM and GEMM calls reach a good efficiency, but it is critical to have an efficient multithreaded factorization of the fully-summed rows.
In shared-memory environments, the large TRSM and GEMM calls reach a good efficiency, but it is critical to have an efficient multithreaded factorization of the fully-summed rows.
Shared-memory factorization algorithm in MUMPS

Structure of a frontal matrix in the multifrontal tree:

In shared-memory environments, the large TRSM and GEMM calls reach a good efficiency, but it is critical to have an efficient multithreaded factorization of the fully-summed rows.
Factorization of the fully summed rows

**Blocked Right-looking**, Blocked Left-looking and Multi-Level blocking

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Factorization of the fully summed rows

**Blocked Right-looking**, Blocked Left-looking and Multi-Level blocking

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Factorization of the fully summed rows

**Blocked Right-looking**, Blocked Left-looking and Multi-Level blocking

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Factorization of the fully summed rows

**Blocked Right-looking**, Blocked Left-looking and Multi-Level blocking

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Factorization of the fully summed rows

**Blocked Right-looking**, Blocked Left-looking and Multi-Level blocking

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Factorization of the fully summed rows

**Blocked Right-looking**, Blocked Left-looking and Multi-Level blocking

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Factorization of the fully summed rows

**Blocked Right-looking**, Blocked Left-looking and Multi-Level blocking

![Diagram showing factorization](image)

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Factorization of the fully summed rows

**Blocked Right-looking**, Blocked Left-looking and Multi-Level blocking

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Factorization of the fully summed rows

**Blocked Right-looking**, Blocked Left-looking and Multi-Level blocking

![Diagram showing the factorization process]

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The Right-looking algorithm consists in updating all the trailing matrix after the factorization of each panel.
Factorization of the fully summed rows

Blocked Right-looking, **Blocked Left-looking** and Multi-Level blocking

The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
Factorization of the fully summed rows

Blocked Right-looking, **Blocked Left-looking** and Multi-Level blocking

The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
Factorization of the fully summed rows

Blocked Right-looking, **Blocked Left-looking** and Multi-Level blocking

The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
Factorization of the fully summed rows

Blocked Right-looking, **Blocked Left-looking** and Multi-Level blocking

The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
Blocked Right-looking, **Blocked Left-looking** and Multi-Level blocking

The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
Factorization of the fully summed rows

Blocked Right-looking, **Blocked Left-looking** and Multi-Level blocking

The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
Factorization of the fully summed rows

Blocked Right-looking, **Blocked Left-looking** and Multi-Level blocking

The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
Factorization of the fully summed rows

Blocked Right-looking, **Blocked Left-looking** and Multi-Level blocking

The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
Factorization of the fully summed rows

Blocked Right-looking, **Blocked Left-looking** and Multi-Level blocking

The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
Factorization of the fully summed rows

Blocked Right-looking, **Blocked Left-looking** and Multi-Level blocking

The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
Factorization of the fully summed rows

Blocked Right-looking, **Blocked Left-looking** and Multi-Level blocking

The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
Factorization of the fully summed rows

Blocked Right-looking, **Blocked Left-looking** and Multi-Level blocking

The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
Factorization of the fully summed rows

Blocked Right-looking, **Blocked Left-looking** and Multi-Level blocking

The Left-looking algorithm consists in updating only the next panel to be factorized using all the already factorized panels.
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and **Multi-Level blocking**

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, ...) during the factorization of another level of panels (blocks).
The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, ...) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, ... ) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and **Multi-Level blocking**

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, ...) during the factorization of another level of panels (blocks).
The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, ... ) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, . . . ) during the factorization of another level of panels (blocks).
The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, . . . ) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, . . . ) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and **Multi-Level blocking**

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, ...) during the factorization of another level of panels (blocks).
Blocked Right-looking, Blocked Left-looking and **Multi-Level blocking**

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, . . .) during the factorization of another level of panels (blocks).
The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, . . .) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, . . . ) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, . . . ) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, . . .) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, ...) during the factorization of another level of panels (blocks).
The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, . . . ) during the factorization of another level of panels (blocks).
Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, ...) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, ... ) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, ... ) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, . . . ) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, ... ) during the factorization of another level of panels (blocks).
The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, ...) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and **Multi-Level blocking**

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, . . .) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, . . . ) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, ...) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, ...) during the factorization of another level of panels (blocks).
Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, . . . ) during the factorization of another level of panels (blocks).
Factorization of the fully summed rows

Blocked Right-looking, Blocked Left-looking and Multi-Level blocking

The multi-level blocking algorithms consist in using a blocked factorization algorithm (Right-looking, Left-looking, recursive, ...) during the factorization of another level of panels (blocks).
### Results on fronts: Sequential case

<table>
<thead>
<tr>
<th>nfront</th>
<th>npiv</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>256</td>
<td>$1.52 \times 10^{-2}$</td>
<td>$1.42 \times 10^{-2}$</td>
<td>$1.40 \times 10^{-2}$</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>$3.86 \times 10^{-2}$</td>
<td>$3.55 \times 10^{-2}$</td>
<td>$3.50 \times 10^{-2}$</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>$1.00 \times 10^{-1}$</td>
<td>$9.04 \times 10^{-2}$</td>
<td>$8.98 \times 10^{-2}$</td>
</tr>
<tr>
<td>10000</td>
<td>2048</td>
<td>$2.96 \times 10^{+0}$</td>
<td>$2.62 \times 10^{+0}$</td>
<td>$2.58 \times 10^{+0}$</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>$1.03 \times 10^{+1}$</td>
<td>$9.30 \times 10^{+0}$</td>
<td>$9.14 \times 10^{+0}$</td>
</tr>
<tr>
<td></td>
<td>8192</td>
<td>$3.25 \times 10^{+1}$</td>
<td>$3.04 \times 10^{+1}$</td>
<td>$3.03 \times 10^{+1}$</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>$4.40 \times 10^{+1}$</td>
<td>$4.14 \times 10^{+1}$</td>
<td>$4.12 \times 10^{+1}$</td>
</tr>
</tbody>
</table>

Computation times (seconds) of partial factorizations using 1, 2 and 3 levels of blocking on 1 core of an Intel Sandy-Bridge processor
## Results on fronts: Multithreaded case

<table>
<thead>
<tr>
<th>nfront</th>
<th>npiv</th>
<th>Nb of blocking levels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>256</td>
<td>$1.31 \times 10^{-2}$</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>$1.86 \times 10^{-2}$</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>$3.10 \times 10^{-2}$</td>
</tr>
<tr>
<td>10000</td>
<td>2048</td>
<td>$9.41 \times 10^{-1}$</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>$2.49 \times 10^{+0}$</td>
</tr>
<tr>
<td></td>
<td>8192</td>
<td>$6.80 \times 10^{+0}$</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>$8.29 \times 10^{+0}$</td>
</tr>
</tbody>
</table>

Computation times (seconds) of partial factorizations using 1, 2 and 3 levels of blocking on 8 cores of an Intel Sandy-Bridge processor. On the 10000 front, the recursive algorithm takes 6.45 sec while the three-level blocking algorithm takes 6.16 sec.
• Implementation of double-blocking factorization in MUMPS, as the major gain due to multi-level factorization comes from double-blocking and as this keeps the code simple enough to operate with full MUMPS functionalities

• Useful both for Full-rank factorization and Low-rank approximation (talk by C. Weisbecker)
## Preliminary results in MUMPS

<table>
<thead>
<tr>
<th>Matrix</th>
<th>one level of blocking</th>
<th>two levels of blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>acoustic_pml_DF</td>
<td>428.94</td>
<td>430.20</td>
</tr>
<tr>
<td>acoustic_RC1</td>
<td>1242.43</td>
<td>1241.81</td>
</tr>
<tr>
<td>acoustic_RC3</td>
<td>634.66</td>
<td>622.45</td>
</tr>
<tr>
<td>TM_bypass</td>
<td>2311.29</td>
<td>1960.16</td>
</tr>
<tr>
<td>TM_inlet</td>
<td>7571.52</td>
<td>5950.51</td>
</tr>
<tr>
<td>VA_pl (*)</td>
<td>4179.85</td>
<td>4136.35</td>
</tr>
<tr>
<td>VA_RC4</td>
<td>105.90</td>
<td>103.84</td>
</tr>
</tbody>
</table>

- Factorization times (seconds) of MUMPS on 8 cores, Intel Sandy Bridge (no tree parallelism)
- Matrices were provided by Eveline Rosseel (FFT)
- High sensitivity of small fronts to panel sizes
- Good preliminary results on matrices with large fronts

(*) High cost of numerical pivoting
Introduction

1. Tree parallelism at thread level

2. Multithreaded dense factorization kernels

3. Distributed and hybrid dense factorization kernels

Conclusion
The distributed-memory factorization in MUMPS is a 1D pipelined asynchronous factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a 1D pipelined asynchronous factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement).
Distributed-memory factorization algorithm in MUMPS

The distributed-memory factorization in MUMPS is a **1D pipelined asynchronous** factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a **1D pipelined asynchronous** factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a 1D pipelined asynchronous factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a **1D pipelined asynchronous** factorization, with one **master** process factorizing the fully-summed part of the front and sending the successive panels to all other **slave** processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a **1D pipelined asynchronous** factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a 1D pipelined asynchronous factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a 1D pipelined asynchronous factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a 1D pipelined asynchronous factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement)
The distributed-memory factorization in MUMPS is a 1D pipelined asynchronous factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a **1D pipelined asynchronous** factorization, with one (**master**) process factorizing the fully-summed part of the front and sending the successive panels to all other (**slave**) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a 1D pipelined asynchronous factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a 1D pipelined asynchronous factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a **1D pipelined asynchronous** factorization, with one (**master**) process factorizing the fully-summed part of the front and sending the successive panels to all other (**slave**) processes which update the $L$ factors and contribution block (**Schur complement**).
The distributed-memory factorization in MUMPS is a 1D pipelined asynchronous factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a 1D pipelined asynchronous factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a **1D pipelined asynchronous** factorization, with one (**master**) process factorizing the fully-summed part of the front and sending the successive panels to all other (**slave**) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a 1D pipelined asynchronous factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement)
The distributed-memory factorization in MUMPS is a 1D pipelined asynchronous factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a 1D pipelined asynchronous factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a **1D pipelined asynchronous** factorization, with one (**master**) process factorizing the fully-summed part of the front and sending the successive panels to all other (**slave**) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a 1D pipelined asynchronous factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the L factors and contribution block (Schur complement)
The distributed-memory factorization in MUMPS is a 1D pipelined asynchronous factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement).
The distributed-memory factorization in MUMPS is a **1D pipelined asynchronous** factorization, with one (**master**) process factorizing the fully-summed part of the front and sending the successive panels to all other (**slave**) processes which update the $L$ factors and contribution block (Schur complement)
The distributed-memory factorization in MUMPS is a 1D pipelined asynchronous factorization, with one (master) process factorizing the fully-summed part of the front and sending the successive panels to all other (slave) processes which update the $L$ factors and contribution block (Schur complement).
• A Python simulator allowed us to better understand the algorithmic bottlenecks of the factorization
• It allowed us to simulate the behavior of the factorisation for several algorithmic variants and for different parameters:
  ◦ size of fronts (nfront)
  ◦ size of schur (ncb) or number of fully-summed variables (npiv)
  ◦ number of MPI processes (nproc)
  ◦ number threads per process (nth)
  ◦ GFlop rate of each CPU ($\alpha$)
  ◦ network bandwidth ($\gamma$)
  ◦ ...
Theoretical results

Communication buffer evolution

Influence of panel size

Time and Speed-Up

Influence of panel size

Time and Speed-Up

Theoretical results
Theoretical results

Figure: GANNT chart of master (top) and typical slave (bottom) process during 1D distributed factorization

- Right-looking algorithm has an inner limitation as the slaves have not enough work in the beginning of the computation and have too much at the end.
- On the contrary, the Left-looking algorithm does not suffer from these limitations as all processes have enough work at any time and all finish simultaneously.
Experimental results

The factorization of a front of size 10,000 on 8 MPI processes each with 1 thread shows that nearly all slaves start their computations very late, wasting lots of time.
A deeper analysis shows that there is no overlapping of computations and communications, as the slaves start receiving the panels only when the master reenter in the MPI layer.
By forcing the master process to enter the MPI layer as often as possible, things get better, and we can notice the real limitations of the algorithm.
Giving the master as many matrix rows as the slaves instead of balancing the work load between all processes improves the factorization
When implementing the Left-looking algorithm in MUMPS, we notice (unexpectedly) that the master tend to be slower than predicted so that it tends to delay the slaves. Giving it less work solves the problem.
When the master process is overloaded, slave processes spend their time being idle.
When the master process is overloaded, slave processes spend their time being idle.
When the master process is overloaded, slave processes spend their time being idle.
A solution consists in giving the master the amount of work such as to get best possible performance on the factorization
A solution consists in giving the master the amount of work such as to get best possible performance on the factorization.
A solution consists in giving the master the amount of work such as to get best possible performance on the factorization.
Then, the same operation is to be repeated on the remaining unfactorized piece of the front, either (standard mapping) by remapping this subfront on all processes, with the cost of an assembly operation (communication)
Then, the same operation is to be repeated on the remaining unfactorized piece of the front, either (standard mapping) by remapping this subfront on all processes, with the cost of an assembly operation (communication)
Then, the same operation is to be repeated on the remaining unfactorized piece of the front, either *(standard mapping)* by remapping this subfront on all processes, with the cost of an assembly operation (communication)
Then, the same operation is to be repeated on the remaining unfactorized piece of the front, either (standard mapping) by remapping this subfront on all processes, with the cost of an assembly operation (communication)
Then, the same operation is to be repeated on the remaining unfactorized piece of the front, or (*communication-free mapping*) by making the processes work on the exact same pieces of the front (losing one process)
Then, the same operation is to be repeated on the remaining unfactorized piece of the front, or *(communication-free mapping)* by making the processes work on the exact same pieces of the front (losing one process)
Then, the same operation is to be repeated on the remaining unfactorized piece of the front, or (*communication-free mapping*) by making the processes work on the exact same pieces of the front (losing one process)
Then, the same operation is to be repeated on the remaining unfactorized piece of the front, or (*communication-free mapping*) by making the processes work on the exact same pieces of the front (losing one process)
Then, the same operation is to be repeated on the remaining unfactorized piece of the front, or (*communication-free mapping*) by making the processes work on the exact same pieces of the front (losing one process)
This operation is called **splitting** and consists in transforming a front in the elimination tree into a **splitted chain**.
This operation is called **splitting** and consists in transforming a front in the elimination tree into a **splitted chain**
This operation is called **splitting** and consists in transforming a front in the elimination tree into a **splitted chain**.
Restarting

Standard mapping: remap at each node of the chain

Communication-free mapping: one process lost at each node of the chain
Restarting

Standard mapping: remap at each node of the chain

Communication-free mapping: one process lost at each node of the chain

Restart: intermediate strategy
Theoretical results

- Heuristics have been studied in order to decide, during the build phase of the splitted chain, whether to restart or not.
- Simulations have been achieved in order to estimate the potential of restarting, assuming efficient asynchronous panel broadcast during factorization (cf. talk by F-H.Rouet)
- The simulated speed-up of a typical partial factorization with:
  - $n_{\text{front}} = 100,000$
  - $n_{\text{piv}} = 50,000$
  - $n_{\text{proc}} = 64$ MPI
  - $n_{\text{thread}} = 8$ threads (per MPI)

  on a typical cluster is:
  - 56/64 with *standard* splitting
  - 49/64 with *communication-free* splitting
  - 60/64 with *restarting*
Introduction

1. Tree parallelism at thread level

2. Multithreaded dense factorization kernels

3. Distributed and hybrid dense factorization kernels

Conclusion
**Summary**

**Threaded tree parallelism implemented in factorization**
- Approach efficiently exploits NUMA and threaded BLAS
- Critical for matrices such as circuit-simulation and 2D meshes

**Numerical factorization kernels**
- Significant gains already observed
- Several algorithmic and low-level improvement issues identified and leveraged

**Availability at short term (MUMPS 5.0)**
- Improved unsymmetric multithreaded factorization kernels: two levels of blocking with tuned block sizes
- OpenMP directives to speed-up computations outside BLAS calls
- Speed-up communications with frequent entrance in the MPI layer
• Pursue optimizations (parallel cache assignment techniques, asynchronous 1D factorizations, tune symmetric kernels, . . . )
• Combine everything (dense kernels + tree parallelism in entire multifrontal factorization with modified dynamic scheduling)
• MPI vs. threads: 1 MPI per node or per socket? change number of threads per MPI process at runtime?
• Multithreaded solve algorithm – implement tree parallelism during solve
• Combine with other MUMPS features and evolutions, esp. memory-aware mappings/scheduling (talk by F.-H. Rouet)
• Make available in future MUMPS releases to solve huge problems on huge numbers of cores