Robust memory-aware mappings for parallel multifrontal factorizations

Joint work with P. R. Amestoy, E. Agullo, A. Buttari, A. Guermouche, J.-Y. L’Excellent

François-Henry Rouet, Lawrence Berkeley National Laboratory

MUMPS Users Group Meeting, May 29-30, 2013
Motivation

- Memory is often a bottleneck for direct solvers.
- Physical memory per core tends to decrease.
- Estimating memory consumption in a dynamic scheduling context is difficult (cf. the infamous error code “-9” in MUMPS).

Context

- We want to design mapping algorithms that enforce some memory constraints and provide better memory estimates.
- We focus on multifrontal methods (e.g., MUMPS).
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:
- Factors
- Active memory

Factors  
Active frontal matrix  
Stack of contribution blocks
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:

- Factors
- Active memory

\[
A = L + U - I =
\]

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
1 & & & & \\
2 & & & & \\
3 & & & & \\
4 & & & & \\
5 & & & & \\
\end{array}
\]
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:
- Factors
- Active memory
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:
- Factors
- Active memory
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:
- Factors
- Active memory
Storage is divided into two parts:
- Factors
- Active memory
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:

- Factors
- Active memory
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:

- Factors
- Active memory

A = L + U - I =

Active memory
Storage is divided into two parts:

- Factors
- Active memory
The multifrontal method [Duff & Reid '83]

Storage is divided into two parts:
- Factors
- Active memory
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:
- Factors
- Active memory
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:
- Factors
- Active memory
The multifrontal method [Duff & Reid ’83]

A = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 \\
5 & 4 & 3 & 2 & 1 \\
2 & 3 & 4 & 5 & 1 \\
3 & 4 & 5 & 1 & 2 \\
4 & 5 & 1 & 2 & 3
\end{bmatrix}

\rightarrow

L + U - I = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 \\
5 & 4 & 3 & 2 & 1 \\
2 & 3 & 4 & 5 & 1 \\
3 & 4 & 5 & 1 & 2 \\
4 & 5 & 1 & 2 & 3
\end{bmatrix}

Storage is divided into two parts:

- Factors
- Active memory
The multifrontal method [Duff & Reid ’83]

- Factors are incompressible and usually scale fairly; they can optionally be written on disk.
- In sequential, the traversal that minimizes active memory is known [Liu’86].
- In parallel, active memory becomes dominant.

Example: share of active storage on the AUDI matrix using MUMPS 4.10.0

1 processor: 11%
256 processors: 59%
The problem

Metric: memory efficiency

\[ e(p) = \frac{S_{seq}}{p \times S_{max}(p)} \]

We would like \( e(p) \approx 1 \), i.e. \( S_{seq}/p \) on each processor.

Common mappings/schedulings provide a poor memory efficiency:

- **Proportional mapping**: \( \lim_{p \to \infty} e(p) = 0 \) on regular problems.
- **MUMPS** relies primarily on a relaxed proportional mapping; typical efficiency: between 0.10 and 0.40.
  Memory estimates are unreliable.
Proportional mapping: top-down traversal of the tree, where processors assigned to a node are distributed among its children proportionally to the weight of their respective subtrees.
Proportional mapping: top-down traversal of the tree, where processors assigned to a node are distributed among its children proportionally to the weight of their respective subtrees.
Proportional mapping: top-down traversal of the tree, where processors assigned to a node are distributed among its children proportionally to the weight of their respective subtrees.
**Proportional mapping:** top-down traversal of the tree, where processors assigned to a node are distributed among its children proportionally to the weight of their respective subtrees.

- Targets run time but poor memory efficiency.
- Usually a **relaxed** version is used: more memory-friendly but **unreliable** estimates.
Proportional mapping: assuming that the sequential peak is 5 GB,

\[ S_{\text{max}}(p) \geq \max \left\{ \frac{4 \text{ GB}}{26}, \frac{1 \text{ GB}}{6}, \frac{5 \text{ GB}}{32} \right\} = 0.16 \text{ GB} \Rightarrow e(p) \leq \frac{5}{64 \times 0.16} \leq 0.5 \]
A more memory-friendly strategy...

All-to-all mapping: postorder traversal of the tree, where all the processors work at every node:
A more memory-friendly strategy...

All-to-all mapping: postorder traversal of the tree, where all the processors work at every node:

Optimal memory scalability \( S_{\text{max}}(p) = S_{\text{seq}}/p \) but no tree parallelism and prohibitive amounts of communications.
A class of “memory-aware” mappings

“Memory-aware” mapping [Agullo '08]: aims at enforcing a given memory constraint ($M_0$, maximum memory per processor):

1. Try to apply proportional mapping.
A class of “memory-aware” mappings

“Memory-aware” mapping [Agullo '08]: aims at enforcing a given memory constraint ($M_0$, maximum memory per processor):

1. Try to apply proportional mapping.
2. Enough memory for each subtree?
A class of “memory-aware” mappings

“Memory-aware” mapping [Agullo '08]: aims at enforcing a given memory constraint ($M_0$, maximum memory per processor):

1. Try to apply proportional mapping.
2. Enough memory for each subtree? If not, serialize them, and update $M_0$: processors stack equal shares of the CBs from previous nodes.
“Memory-aware” mapping [Agullo '08]: aims at enforcing a given memory constraint ($M_0$, maximum memory per processor):

- Ensures the given memory constraint and provides reliable estimates.
- Tends to assign many processors on nodes at the top of the tree \(\Rightarrow\) performance issues on parallel nodes.
A class of “memory-aware” mappings

A finer “memory-aware” mapping? Serializing all the children at once is very constraining: more tree parallelism can be found.

Find groups on which proportional mapping works, and serialize these groups. **Heuristic:** follow a given order (e.g. the serial postorder) and **form groups as large as possible.**
The algorithm maps each child $i$ on $p_i$ processors so that, on every processor working on $i$, two conditions are enforced:

(Cstk) there is enough memory to hold $P_{seq}(i)/p_i$.

(Casm) the assembly of the CB of $i$ into its parent is feasible.

while not all children collected do
  $i =$ current node
  if $i$ cannot be alone then
    Reset previous children to an all-to-all mapping (this should improve balance)
  end if
  Starting from $i$: collect as many nodes as possible as long as (Cstk) and (Casm) are ensured
  Use proportional mapping on the group, serialize with previous ones
end while
The algorithm maps each child \( i \) on \( p_i \) processors so that, on every processor working on \( i \), two conditions are enforced:

- **(Cstk)** there is enough memory to hold \( P_{seq}(i)/p_i \).
- **(Casm)** the assembly of the CB of \( i \) into its parent is feasible.

\[
\text{while not all children collected do}
\]
\[
i = \text{current node}
\]
\[
\text{if } i \text{ cannot be alone then}
\]
\[
\text{Reset previous children to an all-to-all mapping (this should improve balance)}
\]
\[
\text{end if}
\]
\[
\text{Starting from } i: \text{ collect as many nodes as possible as long as (Cstk) and (Casm) are ensured}
\]
\[
\text{Use proportional mapping on the group, serialize with previous ones}
\]
\[
\text{end while}
\]
The algorithm maps each child $i$ on $p_i$ processors so that, on every processor working on $i$, two conditions are enforced:

\begin{itemize}
  \item \textbf{(Cstk)} there is enough memory to hold $P_{seq}(i)/p_i$.
  \item \textbf{(Casm)} the assembly of the CB of $i$ into its parent is feasible.
\end{itemize}

\begin{verbatim}
while not all children collected do
    i = current node
    if i cannot be alone then
        Reset previous children to an all-to-all mapping (this should improve balance)
    end if
    Starting from $i$: collect as many nodes as possible as long as (Cstk) and (Casm) are ensured
    Use proportional mapping on the group, serialize with previous ones
end while
\end{verbatim}
The algorithm maps each child $i$ on $p_i$ processors so that, on every processor working on $i$, two conditions are enforced:

(Cstk) there is enough memory to hold $P_{seq}(i)/p_i$.

(Casm) the assembly of the CB of $i$ into its parent is feasible.

while not all children collected do
  $i =$ current node
  if $i$ cannot be alone then
    Reset previous children to an all-to-all mapping (this should improve balance)
  end if
  Starting from $i$: collect as many nodes as possible as long as (Cstk) and (Casm) are ensured
  Use proportional mapping on the group, serialize with previous ones
end while
The algorithm maps each child $i$ on $p_i$ processors so that, on every processor working on $i$, two conditions are enforced:

(Cstk) there is enough memory to hold $P_{seq}(i)/p_i$.

(Casm) the assembly of the CB of $i$ into its parent is feasible.

while not all children collected do
  $i$ = current node
  if $i$ cannot be alone then
    Reset previous children to an all-to-all mapping (this should improve balance)
  end if
  Starting from $i$: collect as many nodes as possible as long as (Cstk) and (Casm) are ensured
  Use proportional mapping on the group, serialize with previous ones
end while
Experiments

- **Matrix**: finite-difference model of acoustic wave propagation, 27-point stencil, $192 \times 192 \times 192$ grid; **Seiscope consortium**. $N = 7 \text{ M}$, $nnz = 189 \text{ M}$, factors=152 GB (METIS). Sequential peak of active memory: 46 GB.

- **Machine**: 64 nodes with two quad-core Xeon X5560 per node. We use 256 MPI processes.

- **Perfect memory scalability**: $46 \text{ GB} / 256 = 180 \text{ MB}$. 
### Experiments

<table>
<thead>
<tr>
<th>Prop Map</th>
<th>Memory-aware mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$M_0 = 225$ MB</td>
</tr>
<tr>
<td></td>
<td>$M_0 = 380$ MB</td>
</tr>
<tr>
<td></td>
<td>w/o groups</td>
</tr>
<tr>
<td>Max stack peak (MB)</td>
<td>1932</td>
</tr>
<tr>
<td>Avg stack peak (MB)</td>
<td>626</td>
</tr>
<tr>
<td>Time (s)</td>
<td>1323</td>
</tr>
</tbody>
</table>

The root node has 3 children that cannot be mapped using a proportional mapping:

\[ 46 \text{ GB} + 32 \text{ GB} + 30 \text{ GB} \succ 380 \text{ MB} \times 256 \]

- The “basic” memory-aware mapping serializes the three children.
- The variant with groups puts two children together and one alone.
Experiments

We assess how the different strategies map nodes, compared to the proportional mapping.

- Relaxing $M_0$ enhances tree parallelism.
- The variant with groups enhances tree parallelism.
Performance issues

The new mapping tends to assign many processes to nodes at the top of the tree. One of the communication patterns became a bottleneck.

Non-blocking broadcast of a block of factors from the master process.
A kind of ibcast.
The new mapping tends to assign many processes to nodes at the top of the tree. One of the communication patterns became a bottleneck.

Baseline implementation: a loop of non-blocking sends from the master to the processors. Communication speed is constant. Hyperion (CICT): 1.7 GB/s; Hopper (NERSC): 4.5 GB/s.
Performance issues

The new mapping tends to assign many processes to nodes at the top of the tree. One of the communication patterns became a bottleneck.

Baseline implementation: a loop of non-blocking sends from the master to the processors. Communication speed is constant. Hyperion (CICT): 1.7 GB/s; Hopper (NERSC): 4.5 GB/s.

Tree-based implementation: communication speed is almost proportional to the number of processors. Hyperion, 128 cores: 28 GB/s; Hopper, 768 cores: 122 GB/s.
MUMPS follows an (almost) fully dynamic scheduling scheme where a process can be asked on the fly to work on virtually any task.

**Main loop:**

```plaintext
while ... do
    Probe for message
    if message received then
        Read tag, execute associated task
        (e.g., Facto panel)
    end if
end while
```

**Facto panel:**

Do BLAS stuff on panel
Try to send panel to some processes:
if Send buffer is full then
    Call Main loop
end if
Free stuff
MUMPS follows an (almost) fully dynamic scheduling scheme where a process can be asked on the fly to work on virtually any task.

**Main loop:**
while ... do
  Probe for message
  if message received then
    Read tag, execute associated task
    (e.g., Facto panel)
  end if
end if
end while

**Facto panel:**
Do BLAS stuff on panel
Try to send panel to some processes:
if Send buffer is full then
  Call Main loop
end if
Free stuff
MUMPS follows an (almost) fully dynamic scheduling scheme where a process can be asked on the fly to work on virtually any task.

**Main loop:**

```plaintext
while ... do
    Probe for message
    if message received then
        Read tag, execute associated task
        (e.g., Facto panel)
    end if
end while
```

**Facto panel:**

Do BLAS stuff on panel
Try to send panel to some processes:

```plaintext
if Send buffer is full then
    Call Main loop
end if
Free stuff
```
MUMPS follows an (almost) fully dynamic scheduling scheme where a process can be asked on the fly to work on virtually any task.

**Main loop:**
while ... do
    Probe for message
    if message received then
        Read tag, execute associated task
        (e.g., Facto panel)
    end if
end while

**Facto panel:**
Do BLAS stuff on panel
Try to send panel to some processes:
if Send buffer is full then
    Call Main loop
end if
Free stuff
Non-blocking broadcast – implementation

MUMPS follows an (almost) fully dynamic scheduling scheme where a process can be asked on the fly to work on virtually any task.

Main loop:
while ... do
  Probe for message
  if message received then
    Read tag, execute associated task
    (e.g., Facto panel)
  end if
end while

Facto panel:
Do BLAS stuff on panel
Try to send panel to some processes:
if Send buffer is full then
  Call Main loop
end if
Free stuff
MUMPS follows an (almost) fully dynamic scheduling scheme where a process can be asked on the fly to work on virtually any task.

**Main loop:**
while ... do
  Probe for message
  if message received then
    Read tag, execute associated task
    (e.g., Facto panel)
  end if
end while

**Facto panel:**
Do BLAS stuff on panel
Try to send panel to some processes:
if Send buffer is full then
  Call Main loop
end if
Free stuff
MUMPS follows an (almost) fully dynamic scheduling scheme where a process can be asked on the fly to work on virtually any task.

**Main loop:**

```plaintext
while ... do
    Probe for message
    if message received then
        Read tag, execute associated task
        (e.g., Facto panel)
    end if
end while
```

**Facto panel:**

Do BLAS stuff on panel
Try to send panel to some processes:
if Send buffer is full then
    Call Main loop
end if
Free stuff
MUMPS follows an (almost) fully dynamic scheduling scheme where a process can be asked on the fly to work on virtually any task.

**Main loop:**
while ... do
  Probe for message
  if message received then
    Read tag, execute associated task
    (e.g., **Facto panel**)
  end if
end while

**Facto panel:**
Do BLAS stuff on panel
Try to send panel to some processes:
if Send buffer is full then
  Call **Main loop**
end if
Free stuff
MUMPS follows an (almost) fully dynamic scheduling scheme where a process can be asked on the fly to work on virtually any task.

Main loop:
while ... do
    Probe for message
    if message received then
        Read tag, execute associated task  
        (e.g., Facto panel)
    end if
end while

Facto panel:
Do BLAS stuff on panel
Try to send panel to some processes:
if Send buffer is full then
    Call Main loop
end if
Free stuff
MUMPS follows an (almost) fully dynamic scheduling scheme where a process can be asked on the fly to work on virtually any task.

**Main loop:**
while ... do
  Probe for message
  if message received then
    Read tag, execute associated task
    (e.g., Facto panel)
  end if
end while

**Facto panel:**
Do BLAS stuff on panel
Try to send panel to some processes:
if Send buffer is full then
  Call Main loop
end if
Free stuff
MUMPS follows an (almost) fully dynamic scheduling scheme where a process can be asked on the fly to work on virtually any task.

**Main loop:**

```plaintext
while ... do
    Probe for message
    if message received then
        Read tag, execute associated task (e.g., Facto panel)
    end if
end while
```

**Facto panel:**

Do BLAS stuff on panel
Try to send panel to some processes:
if Send buffer is full then
    Call Main loop
end if
Free stuff
Non-blocking broadcast – implementation

Difficulties:

- Messages representing different blocks of factors can overtake.
- Deadlocks...

Example of messages overtaking (broadcast tree is a chain):

$P_0$ receives the second panel before the first one!
Experiments on a single node of size 64000 with 64 processors:

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-way threaded BLAS</td>
</tr>
<tr>
<td>Baseline</td>
<td>596</td>
</tr>
<tr>
<td>Tree-based</td>
<td>401</td>
</tr>
</tbody>
</table>

Better overlapping between communications and computations ⇒ more potential for exploiting multithreaded BLAS / faster cores.

Quite critical in a memory-aware context since the algorithm tends to increase the number of processes per subtree.
Back to experiments with the memory-aware mapping; matrices:

<table>
<thead>
<tr>
<th>Matrix name</th>
<th>Order (millions)</th>
<th>Entries (millions)</th>
<th>Factors (GB)</th>
<th>$S_{seq}$ (GB)</th>
<th>Description; origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>cage13</td>
<td>0.4</td>
<td>7.5</td>
<td>30.7</td>
<td>17.9</td>
<td>Directed weighted graph; Utrecht U.</td>
</tr>
<tr>
<td>as-Skitter</td>
<td>1.7</td>
<td>23.9</td>
<td>17.7</td>
<td>6.2</td>
<td>Internet topology graph; SNAP</td>
</tr>
<tr>
<td>HV15R</td>
<td>2.0</td>
<td>283.1</td>
<td>366.4</td>
<td>87.8</td>
<td>CFD, 3D engine fan; FLUOREM</td>
</tr>
<tr>
<td>MORANSYS1</td>
<td>2.7</td>
<td>81.3</td>
<td>63.5</td>
<td>19.1</td>
<td>Model Order Reduction; CADFEM</td>
</tr>
<tr>
<td>meca_raff6</td>
<td>3.3</td>
<td>130.2</td>
<td>63.5</td>
<td>8.8</td>
<td>Thermo-mechanical coupling; EDF</td>
</tr>
</tbody>
</table>
### Memory-aware mapping vs. default strategy (MUMPS 4.10.0):

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Mapping</th>
<th>$S_{max}$ (MB)</th>
<th>$S_{avg}$ (MB)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cage13</td>
<td>Default</td>
<td>1069</td>
<td>727</td>
<td>285</td>
</tr>
<tr>
<td></td>
<td>MA, $M_0 = 380$ MB</td>
<td>305</td>
<td>302</td>
<td>498</td>
</tr>
<tr>
<td>as-Skitter</td>
<td>Default</td>
<td>1484</td>
<td>584</td>
<td>578</td>
</tr>
<tr>
<td></td>
<td>MA, $M_0 = 130$ MB</td>
<td>135</td>
<td>70</td>
<td>285</td>
</tr>
<tr>
<td>HV15R</td>
<td>Default</td>
<td>9063*</td>
<td>8773*</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>MA, $M_0 = 2600$ MB</td>
<td>2225</td>
<td>1803</td>
<td>7169</td>
</tr>
<tr>
<td>MORAN SYS1</td>
<td>Default</td>
<td>1246</td>
<td>834</td>
<td>373</td>
</tr>
<tr>
<td></td>
<td>MA, $M_0 = 380$ MB</td>
<td>379</td>
<td>305</td>
<td>651</td>
</tr>
<tr>
<td>meca_raff6</td>
<td>Default</td>
<td>1078</td>
<td>796</td>
<td>324</td>
</tr>
<tr>
<td></td>
<td>MA, $M_0 = 320$ MB</td>
<td>329</td>
<td>209</td>
<td>367</td>
</tr>
</tbody>
</table>
## Conclusion

- The memory-aware mapping ensures a given constraint...
- ...and its variant with groups tries to add more parallelism.
- Increasing the number of processes per node led to performance problems ⇒ new (dense!) algorithms.

## Future work

- Memory-aware mapping with groups: assess some heuristics.
- We have performance models that provide the optimal number of processors to use on a node, and we are working on how to inject these models into the mapping.
- More dynamic scheduling.
- Compressing CBs with low-rank approximation techniques (cf work by Clément Weisbecker and the MUMPS team).
Thank you for your attention!

Any questions?

Acknowledgements

- Matrices: Stéphane Operto (SEISCOPE), Olivier Boiteau (EDF).
- Computational resources: Nicolas Renon, CICT, Toulouse.